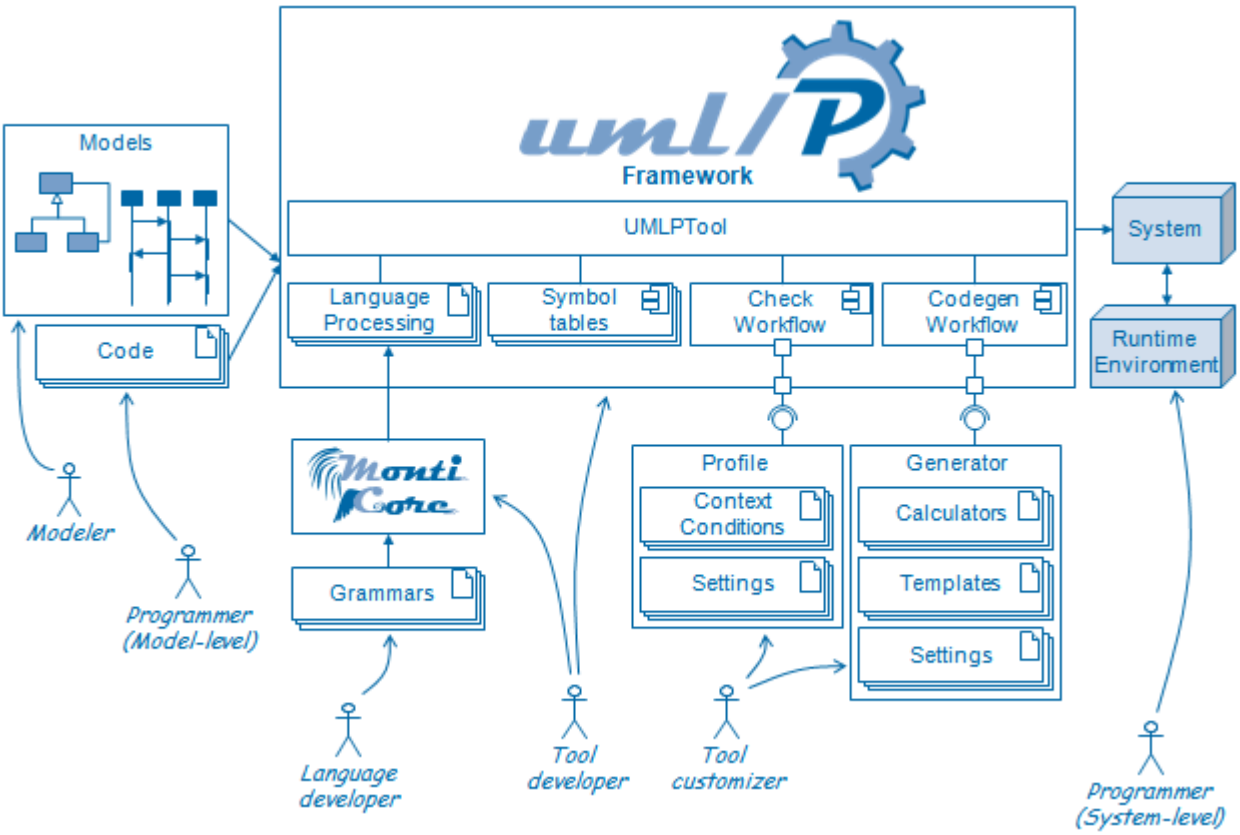
Martin Schindler

Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P



Aachener Informatik-Berichte,
Software Engineering

Band 11

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

[Sch12] M. Schindler
Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P
Shaker Verlag, ISBN 978-3-8440-0864-7.
Aachener Informatik-Berichte,
Software Engineering Band 11. 2012.
www.se-rwth.de/publications

Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P

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

Diplom-Informatiker
Martin Schindler

aus Dortmund

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Albert Zündorf

Tag der mündlichen Prüfung: 23. Dezember 2011

Die Druckfassung dieser Dissertation ist unter der ISBN 978-3-8440-0864-7 erschienen.

"It's not the strongest who survive, nor the most intelligent, but the ones most adaptable to change."

Charles Darwin

Kurzfassung

Einer der vielversprechendsten Ansätze, um der steigenden Komplexität von Softwareentwicklungsprojekten zu begegnen, ist die Abstraktion durch Modellierung, für die sich in den letzten Jahren die UML etabliert hat. Diese eher schwergewichtige und in Teilen redundante Familie von Modellierungssprachen wurde mit der UML/P auf wesentliche Kernnotationen reduziert und hinsichtlich der Anforderungen agiler, modellgetriebener Entwicklungsprozesse optimiert. In Kombination mit Java als Aktionssprache erlaubt die UML/P eine vollständige Modellierung und Qualitätssicherung von Softwaresystemen, deren technische Realisierung mittels Generatoren aus den Modellen abgeleitet wird.

Die praktische Anwendung der agilen Konzepte und Modellierungstechniken der UML/P wird mit dieser Arbeit erstmalig durch eine Werkzeuginfrastruktur unterstützt. Zusammengefasst sind die wichtigsten Ergebnisse wie folgt:

- Die *Modellierung* von Struktur, Verhalten und Qualitätssicherung von Softwaresystemen wird durch eine an Java angelehnte textuelle Notation der UML/P unterstützt. Diese bietet eine kompakte, effiziente und werkzeugunabhängige Modellerstellung, eine Strukturierung der Modelle in Pakete, eine Unterstützung von Sprachprofilen sowie explizite Modellbeziehungen durch qualifizierte Namensreferenzen und Importe.
- Die *Kontextanalyse* erlaubt eine modell- und sprachübergreifende Konsistenzsicherung der Modelle und kann flexibel an Sprachprofile oder unterschiedliche Modellabstraktionen angepasst und erweitert werden. Dabei wird insbesondere Java als abstrakte Aktionssprache auf Modellebene angehoben und interpretiert.
- Die *Codegenerierung* wird mit Templates als zentrales Artefakt für Entwicklung, Konfiguration und Ausführung von Generatoren modular bezüglich Sprachen und Modellen umgesetzt. Das Tracing von Generat und dessen Quellen, die Optimierung auf lesbare Templates, die automatisierte Ableitung des Datenmodells und die Anbindung von Java vereinfacht die Anpassung und Erweiterung der Generierung im Projektkontext. Dazu wird eine Methodik zur schrittweisen Ableitung von Generatoren aus exemplarischem Quellcode definiert. Konzepte für die Generierung von Produktiv- und Testcode aus UML/P-Modellen sowie ein Verfahren zur sprachunabhängigen Messung der Testüberdeckung auf Modellen ergänzen diesen Ansatz und demonstrieren, wie aus abstrakten Modellen eine kompositionale, erweiterbare und testbare Architektur für das Zielsystem abgeleitet werden kann.
- Die *Werkzeuginfrastruktur* wird als modulares und leichtgewichtiges Framework umgesetzt. Dabei wird jede Sprache als eigenständige Komponente realisiert, die jeweils die grammatikbasierte Sprachdefinition, Sprachverarbeitung, Kontextbedingungen und Generatoren unabhängig von anderen Sprachen enthält. Dies erlaubt eine flexible Erweiterung und variable Zusammensetzung der Sprachen der UML/P nach dem Baukastenprinzip.

Insgesamt ist damit eine umfassende Werkzeuginfrastruktur zur agilen modellgetriebenen Entwicklung mit der UML/P entstanden, die sich flexibel erweitern und an Technologien sowie Domänen anpassen lässt.

Abstract

One of the most promising approaches for handling the increasing complexity of software development projects is the abstraction using modeling for which the UML has been established during the last years. With the UML/P this rather heavyweight and partly redundant family of modeling languages got reduced to essential core notations and optimized according to the requirements of agile, model-driven development processes. Adding Java as action language, the UML/P allows entire modeling and quality assurance of software systems whereas the technical realization is derived from the models using generators.

With this work the practical application of the agile concepts and modeling techniques of the UML/P are supported by a tooling infrastructure for the first time. Summarized the main results are as follows:

- The *modeling* of structure, behavior, and quality assurance of software systems is supported by a Java-like textual notation for the UML/P. This offers a compact, efficient, and tooling-independent model creation, a structuring of models in packages, a support of language profiles as well as explicit model relations using qualified name references and imports.
- The *context analysis* allows model- and language-crossing consistency checks of the models and is flexibly adaptable and extendable for language profiles or different model abstractions. Particularly the context analysis interprets Java against the models so that Java is raised to the model-level as abstract action language.
- The *code generation* is realized using templates as central artifact for development, configuration, and execution of generators in a modular manner concerning languages and models. Tracing of generated code and its sources, optimizations for readable templates, automatic derivation of the data model, and invoking of Java code simplifies the adaption and extension of the generators within a development project. In addition, a method for a stepwise development of generators from exemplary source code is defined. The approach is complemented by concepts for production and test code generation from UML/P-models as well as by a technique for a language-independent calculation of test coverage on models. This demonstrates how a compositional, extendable, and testable architecture can be derived from models for the target system.
- The *tooling infrastructure* is implemented as a modular and lightweight framework. Each language is realized as a self-contained component, which combines the grammar-based language definition, language processing, context conditions and generators independently from other languages. This allows a flexible extension and variable composition of the UML/P languages in a modular manner.

To conclude, this work offers a comprehensive tooling infrastructure for the agile model-driven development with the UML/P which is flexibly extendable and adoptable to technology as well as to domains.

Danksagung

Während meiner Promotion haben mich viele liebe Menschen begleitet und unterstützt und so zum Erfolg dieser Dissertation beigetragen. Dafür möchte ich mich hier herzlich bedanken.

Ganz besonderer Dank gebührt meinem Doktorvater Prof. Dr. Bernhard Rumpe, auf dessen Habilitation diese Arbeit aufbaut. Die gemeinsamen Diskussionen, seine konstruktiven und weitsichtigen Anregungen sowie seine Erfahrung haben maßgeblich zu dieser Arbeit beigetragen. Darüber hinaus habe ich im Rahmen meiner Lehrtätigkeit, der Betreuung des Journals "Software and Systems Modeling" (SoSyM) sowie den zahlreichen Projekten im Industrie- und Forschungsumfeld in dieser Zeit viel von ihm über praxisrelevante Techniken und Methoden des Software Engineerings gelernt. Für diese vielschichtige Erfahrung und das sich daraus ergebende abwechslungsreiche und spannende Arbeitsumfeld möchte ich ihm ebenfalls danken.

Prof. Dr. Albert Zündorf danke ich herzlich für sein Interesse an meiner Arbeit und die Übernahme des Zweitgutachtens. Eine ganz besondere Herausforderung war die Organisation der Promotionsprüfung einen Tag vor Weihnachten. Möglich wurde dies erst durch den Einsatz und der Bereitschaft von Prof. Dr. Dr.h.c. Wolfgang Thomas und Prof. Dr. Stefan Kowalewski, selbst an diesem Tag noch die Prüfung durchzuführen.

Für das hervorragende Arbeitsklima, die intensiven Diskussionen und Anregungen, den Ideenaustausch, die Zusammenarbeit in gemeinsamen Projekten und Papieren sowie für die vielen gemeinsamen Erlebnisse möchte ich meinen Kollegen des Instituts für Software Systems Engineering der TU Braunschweig und des Lehrstuhls für Software Engineering der RWTH Aachen danken. Hervorzuheben sind hier insbesondere Dr. Hans Grönniger, Arne Haber, Dr. Holger Krahn, Claas Pinkernell, Dr. Steven Völkel und Ingo Weisemöller, deren Arbeiten rund um MontiCore, Sprachentwicklung und -semantik zu gegenseitigen Anregungen, Impulsen und Synergieeffekten geführt und so eine gemeinsame Forschung sowie kooperatives Arbeiten möglich gemacht haben. Inspiriert wurde ich dabei immer wieder von Holgers fachlicher Kompetenz und Stevens pragmatischer Herangehensweise. In der Endphase meiner Implementierung haben mich darüber hinaus Marita Breuer und Galina Volkova bei der Umsetzung der Symboltabellen und Kontextbedingungen besonders unterstützt. Danken möchte ich auch Ingo, Steven, Hans und meiner Frau Daniela für die kritische Durchsicht meiner Ausarbeitung, sowie Claas Oppitz für die Gestaltung der TripleLogo-Tiere. Auch wenn namentlich nicht genannt, so haben doch alle meine Kollegen dazu beigetragen, dass mir meine Dissertation als eine spannende und schöne Zeit in Erinnerung bleiben wird. Dabei sorgten u.a. gemeinsame Kickerturniere für den sportlichen Ausgleich, bei denen mir insbesondere Steven regelmäßig meine Grenzen aufgezeigt hat.

Nicht zuletzt gilt mein Dank meiner Familie und Freunden für ihre Unterstützung und ihr Verständnis, wenn ich mich wieder mal nicht von meiner Arbeit losreißen konnte. Meinen Eltern möchte ich darüber hinaus für ihre stete Unterstützung, den Rückhalt und das Vertrauen in meinen Lebensweg danken. Ebenfalls danke ich meinen Schwiegereltern, dass auch sie immer für mich da sind. Mein besonderer Dank gebührt schließlich meiner Frau Daniela für ihre uneingeschränkte Unterstützung, Liebe und die unglaubliche Geduld, ohne die diese Arbeit nicht möglich gewesen wäre.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Sprachen dienen als Träger und Übermittler von Informationen. In dieser Funktion werden sie für vielfältige Zwecke eingesetzt. Dazu gehören Kommunikation, Speicherung und Weitergabe von Wissen oder die Beschreibung von Beobachtungen, Vorstellungen und Ideen. Auch in der Softwareentwicklung spielen Sprachen für Planung, Entwurf und Implementierung von Softwaresystemen eine wichtige Rolle. Werden Anforderungen an ein Softwaresystem oft noch in einer natürlichen Sprache festgehalten, werden im Laufe des Entwicklungsprozesses verschiedene formale Sprachen eingesetzt, um diese Anforderungen in eine auf Maschinen ausführbare Form zu übertragen. Die Abstraktionsmechanismen dieser formalen Sprachen zu verbessern und so die Effizienz der Softwareentwicklung zu steigern, ist eine der wesentlichen Forschungsziele der Informatik. Für den praktischen Einsatz essentiell ist dabei eine geeignete Werkzeuginfrastruktur, die eine komfortable und produktive Verwendung einer Sprache erst ermöglicht.

In dieser Arbeit wird aufbauend auf der grundlegenden Einführung in die UML/P und ihren Nutzungsmöglichkeiten in [Rum04a, Rum04b] eine Werkzeuginfrastruktur für die UML/P vorgestellt. Die UML/P setzt sich aus mehreren Einzelsprachen zusammen, die gemeinsam eine abstrakte Beschreibung (Modellierung) von Softwaresystemen und deren Qualitätssicherung in Form von Testfällen erlauben. Auf diese Weise entstehen Modelle des zu entwickelnden Softwaresystems, die mit Hilfe der Werkzeuginfrastruktur automatisiert verarbeitet werden können. So ist es möglich, die Modelle auf Konsistenz zu überprüfen oder weitere Systemartefakte daraus abzuleiten. Insbesondere können die Modelle mittels Generatoren in eine ausführbare Programmiersprache übersetzt werden, so dass im Idealfall das vollständige Softwaresystem aus den Modellen erzeugt werden kann. Dabei steuern die Generatoren die von den Modellen abstrahierten technischen Details für die Ausführung bei. Darüber hinaus ist die Werkzeuginfrastruktur hochgradig konfigurier- und erweiterbar, so dass Sprache, Analysen und Generatoren flexibel an neue Anforderungen angepasst werden können.

Die folgenden Abschnitte dienen der Einführung in diese Arbeit. In Abschnitt 1.1 wird die Entwicklung der Werkzeuginfrastruktur motiviert und thematisch eingeordnet. Abschnitt 1.2 behandelt die grundlegenden Forschungsfragen und -ziele, die mit dieser Arbeit verfolgt wurden. Abschließend wird in Abschnitt 1.3 der weitere Aufbau der Arbeit beschrieben.

1.1 Motivation und Kontext

Software durchdringt sämtliche Bereiche des modernen Lebens. Die meisten Industrien und Dienstleistungen sind ohne den Einsatz von Software nicht mehr denkbar [BJN+06]. Auch viele Produkte und Innovationen basieren mittlerweile direkt oder indirekt auf Software. So verzeichnet etwa der Softwareanteil in Automobilen in den letzten Jahren einen rasanten Anstieg [PBKS07, Bro06]. Schon heute ist für über 80% der Innovationen und für rund 40% der Produktionskosten im Automobilbereich Software verantwortlich [BKPS07, Bro06]. Eine kostengünstige und effiziente Entwicklung von qualitativ hochwertiger und zuverlässiger Software wird dadurch zu einem wettbewerbsentscheidenden Faktor [BJN+06].

Trotz dieser großen Bedeutung scheitern immer noch viele, insbesondere großangelegte IT-Projekte [BM11]. Als einer der Gründe wird neben Fehlern in der Projektplanung sowie sich ändernden oder unklaren Anforderungen oft die Komplexität der Software selbst oder deren zugrunde liegenden technischen Umgebung angeführt. Hintergrund ist nicht zuletzt die schnelle technologische Entwicklung im IT-Kontext, die nur schwer vorhersehbar ist und dazu führt, dass Wissen und Software bereits nach kurzer Zeit veraltet. Allein in den letzten 10 Jahren haben sich Mobiltelefone, Digitalkameras, Notebooks und Navigationsgeräte zu Massenprodukten entwickelt. Neue Gerätekategorien wie Tablet-PCs, Smartphones und E-Reader sind entstanden und scheinen eine ähnliche Entwicklung zu nehmen. Gemeinsam mit neuen Technologien wie Cloud Computing, Servicearchitekturen, Multi-Prozessor-Systemen oder mobilen Anwendungen sowie neuen Standards wie HDCP[1], EPUB[2] oder LTE[3] führt dies zu einer ständigen Änderung und Weiterentwicklung der technologischen und konzeptionellen Basis von Softwaresystemen. Gleichzeitig steigen die Anforderungen an die Software hinsichtlich Flexibilität, Funktionsumfang, Portabilität und Einbettung in diesen technologischen Kontext [BR07].

Eine Möglichkeit, die Komplexität von Technologien und den Entwicklungsaufwand von Softwaresystemen zu reduzieren, bilden Frameworks. Diese stellen ein Funktionsgerüst für einen bestimmten Problem- oder Anwendungsbereich zur Verfügung, in das über Programmierschnittstellen (engl. *application programming interface* - API) konkrete Implementierungen eingebunden werden können. Auf diese Weise lässt sich ein Softwaresystem in Module zerlegen, die jeweils leichter austauschbar sind - ein Prinzip, das auf [Par72] zurück geht. Die Modularität und Austauschbarkeit wird in diesem Fall allerdings durch das Framework bestimmt, so dass ein Wechsel durchaus mit hohem Aufwand verbunden sein kann. Darüber hinaus sind die Frameworks selbst Änderungen unterworfen, die Wartung und Weiterentwicklung eines darauf aufbauenden Softwaresystems erschweren können. Erst kürzlich hat etwa die Firma Google angekündigt, künftig eine Reihe von APIs zu seinen Technologien und Frameworks nicht mehr zu unterstützen[4].

[1] High-bandwidth Digital Content Protection: 2003 entwickeltes Kopierschutzsystem für Audio-/Videoübertragung

[2] Electronic PUBlication: 2007 veröffentlichter offener Standard für elektronische Bücher (E-Books)

[3] Long Term Evolution: in der Entwicklung befindlicher Mobilfunkstandard für höhere Datenraten

[4] "Frühjahrsputz bei Google-API", Heise News vom 29.05.2011, http://heise.de/-1251953

Ein grundlegendes Problem, dass sich hier zeigt, ist die Verbindung von Systemspezifikation und technischer Realisierung eines Softwaresystems, die sich unabhängig voneinander weiter entwickeln können. Auch Frameworks können in dieser Hinsicht als technische Plattform angesehen werden, an die sich ein Softwaresystem durch die Aufrufe der Schnittstellen bindet. Eine stärkere Trennung und damit weitreichendere Abstraktionsmechanismen bieten hingegen generative Ansätze wie die modellgetriebene Entwicklung (engl. *model-driven engineering* - MDE). Diese beschreiben ein System mit Hilfe von Modellen, die im Idealfall vollständig von der technischen Realisierung abstrahieren. Die für die Ausführung notwendigen technischen Details sind hingegen in Form von Generatoren gekapselt, die schließlich auf Basis der Modelle die Implementierung in einer allgemeinen Programmiersprache (engl. *general purpose language* - GPL) wie Java oder C++ erzeugen. Dadurch wird die technische Realisierung allein durch einen Wechsel der Generatoren austauschbar, ohne dass davon die Systemspezifikation betroffen ist. Gleichzeitig ist es durch den Einsatz verschiedener Generatoren möglich, eine Systemspezifikation auf unterschiedlichen Plattformen zu realisieren. Neben der Codegenerierung erleichtern die Modelle aufgrund der höheren Abstraktion das Verständnis und die Entwicklung der Systemspezifikation und eignen sich damit insbesondere auch zur Analyse, Dokumentation und Kommunikation.

Die Modelle können in verschiedenen, sowohl textuellen als auch graphischen Sprachen spezifiziert werden. Domänenspezifische Sprachen (engl. *domain-specific language* - DSL) [KT08, MHS05] bieten eine auf ein bestimmtes fachliches oder technisches Anwendungsgebiet (Domäne) optimierte Notation. Innerhalb dieser Domäne erlauben DSLs eine sehr abstrakte und damit kompakte Spezifikationen von Lösungen, die dadurch im Allgemeinen von den Experten der Domäne selbst entwickelt werden können. Universeller anwendbar sind hingegen allgemeine Modellierungssprachen (engl. *general purpose modeling language* - GPML) wie die UML [OMG10d, OMG10c], die nicht auf eine bestimmte Domäne festgelegt sind und so eine Modellierung von beliebigen Softwaresystemen erlauben.

Trotz der Vorteile existieren bisher nur wenige Berichte, dass MDE in der Praxis erfolgreich eingesetzt wurde [WWM+07, Rai05, KR05]. In den letzten Jahren zeichnet sich jedoch eine Tendenz für ein zunehmendes Interesse nicht nur in der Forschung, sondern auch in der Industrie ab. Erst kürzlich wurden die Ergebnisse einer Umfrage im Bereich der Softwareentwicklung veröffentlicht [GF10], laut der 50% der Befragten innerhalb der letzten drei Jahre generative Techniken eingesetzt haben. Andererseits wenden weiterhin rund 30% derjenigen, die bereits theoretische Kenntnisse in diesem Bereich besitzen, diese in der Praxis nicht an, obwohl es sich bei fast 90% der Befragten um Entwickler und Architekten gehandelt hat. Neben sozialen und organisatorischen Faktoren [HWRK11, HRW11] werden nach wie vor technische Gründe gegen den praktischen Einsatz von MDE angeführt. Dazu gehören:

- hohe Komplexität der Werkzeuge und der dadurch bedingte Einarbeitungsaufwand [Sta06]
- fehlende Integration von Werkzeugen oder Legacy Code [MRA05, Sta06, MD08]
- unzureichende Skalierung und Reifegrad der Werkzeuge im Praxiseinsatz [BLW05, MD08]

Wie [GF10] zeigt, werden Modelle weiterhin auch nur zu Entwurfs- und Dokumentationszwecken verwendet. Ein solch informeller Einsatz von Modellen jenseits von Werkzeugunterstützung und generativen Ansätzen bringt jedoch erhebliche Nachteile mit sich:

- Eine manuelle Implementierung ist arbeitsintensiv und fehleranfällig.
- Informelle Modelle bieten keine Möglichkeit der automatisierten Analyse.
- Es besteht die Gefahr einer unterschiedlichen Interpretation der Modelle.
- Die Konsistenz zwischen Modellen und Code ist nicht sichergestellt.
- Bei Änderungen entsteht Mehraufwand durch die Anpassung von Modellen und Code.

Zeit- und Kostendruck führen deshalb in der Praxis oft dazu, dass eine Modellierung nur zu Beginn eines Projektes erfolgt oder vollständig unterbleibt. Darüber hinaus wird insbesondere die UML oft als zu schwergewichtig und zu komplex angesehen [Hen05]. In [Rum04a, Rum04b] wurde deshalb mit der UML/P eine auf die wesentlichen Kernnotationen reduzierte Variante der UML entwickelt und mit einer agilen Vorgehensweise [BBB+] zu einem leichtgewichtigen Ansatz für die Entwicklung von Softwaresystemen kombiniert. Dabei kann die UML/P sowohl als Modellierungs-, Test- und Implementierungssprache eingesetzt werden. Gemeinsam mit der agilen Methodik ist so eine evolutionäre und generative Entwicklung von Softwaresystemen auf Modellbasis möglich. Dieser Ansatz wurde bisher in [Rum04a, Rum04b] nur theoretisch behandelt. Mit der vorliegenden Arbeit wird dieser nun erstmalig in einem Werkzeug umgesetzt und damit praktisch anwendbar. Die sich daraus ergebenden Forschungsfragen und -ziele behandelt der folgende Abschnitt.

1.2 Forschungsfragen, -ziele und -herausforderungen

Das Kernziel dieser Arbeit lässt sich mit folgender Forschungsfrage zusammenfassen:

> *Lassen sich die in [Rum04a, Rum04b] entwickelten Konzepte und der Modellierungsansatz der UML/P praktisch in ein Werkzeug zur agilen, modularen und modellgetriebenen Softwareentwicklung umsetzen?*

Damit eine Sprache von einem Werkzeug verarbeitet werden kann, muss diese in einer maschinell verarbeitbaren Form vorliegen. Hierbei ist zwischen textuellen und graphischen Ansätzen zu unterscheiden. Darüber hinaus setzt sich die UML/P aus mehreren Sprachen zusammen. Werden diese als einzelne, abgeschlossene Komponenten entwickelt und erst dann komponiert, können diese später separat weiterentwickelt und ausgetauscht werden. Bezieht sich diese Kompositionalität auf alle Aspekte der Werkzeuginfrastruktur, kann diese und damit die UML/P leichter an zukünftige Technologien und Anforderungen angepasst werden.

Insgesamt setzt sich ein Werkzeug für die modellgetriebene Entwicklung mit der UML/P aus folgenden Teilaspekten zusammen:

- **Modellerstellung**: Notation (konkrete Syntax) und Editoren für die Modellierung.
- **Modellverwaltung**: Speicherung und Versionierung von Modellen, insbesondere auch im Hinblick auf die Unterstützung der Modellierung im Team und verteiltes Arbeiten.
- **Modellverarbeitung**: Einlesen von Modellen und Umsetzung in eine interne Struktur (abstrakte Syntax), die Modellinhalte für die Automatisierung von Analysen und Transformationen geeignet zur Verfügung stellt.
- **Modellanalysen**: Überprüfung der Modelle auf inhaltliche Konsistenz. Darüber hinaus sind weitere Analysen denkbar, die jedoch nicht im Fokus dieser Arbeit liegen.
- **Modelltransformationen**: Erzeugung weiterer Artefakte auf Basis der Modelle. Im Fokus steht hier die automatisierte und vollständige Generierung des Produktivsystems sowie dessen Qualitätssicherung in Form von Testfällen aus den Modellen.

Dabei zeichnet sich die UML/P insbesondere durch folgende Eigenschaften und Möglichkeiten aus, die die UML/P von einer GPL abgrenzen und in einem Werkzeug entsprechend zu berücksichtigen sind:

- **Unterspezifikation**: Auslassung von Details in Modellen in frühen Phasen des Entwicklungsprozesses.
- **Syntaktische Erweiterungen**: Anpassung der UML/P an domänen-, unternehmens- oder projektspezifische Anforderungen.
- **Semantische Variabilität**: Berücksichtigung des auf dem Modellcharakter basierenden Interpretationsspielraums und dessen Festlegung für bestimmte Anwendungsgebiete.

Trotz dieses konzeptionellen und funktionellen Umfangs soll ein möglichst leichtgewichtiges Werkzeug entstehen, das eine einfache Nutzung, Anpassung und Erweiterung für unterschiedliche Kontexte und Zieltechnologien im Rahmen von Softwareentwicklungsprojekten erlaubt. Hierbei ist insbesondere eine Konfigurier- und Erweiterbarkeit der Analysen und Generatoren wichtig, um Modellierung und Generierung hinsichtlich projektspezifischer Anforderungen optimieren zu können. Insgesamt ergeben sich aus diesen Anforderungen eine Reihe weiterer Fragen und Problemstellungen, die ebenfalls zu klären sind:

- Ist im Sinne einer verteilten und agilen Entwicklung mit der UML/P eine graphische oder textuelle Notation besser geeignet? Falls letzteres, wie lässt sich die UML/P in eine für den Menschen gut nutzbare textuelle Notation übertragen?
- Können die Sprachen der UML/P und deren Werkzeuginfrastruktur hinsichtlich Sprachverarbeitung, Konsistenzprüfung und Codegenerierung kompositional entwickelt werden?
- Welche Schnittstellen müssen zwischen den einzelnen Sprachen der UML/P bestehen, damit sich aus mehreren Modellen ein für Mensch und Werkzeuginfrastruktur konsistentes und nachvollziehbares Gesamtsystem ergibt?

- Können Modelle für die Konsistenzsicherung und Codegenerierung modular verarbeitet werden?
- Wie lassen sich les- und wartbare Codegeneratoren entwickeln?
- Wie können Sprache und Werkzeuginfrastruktur flexibel an spezifische Anforderungen von Domänen, Unternehmen oder Projekten angepasst werden?

1.3 Aufbau der Arbeit

Der Aufbau dieser Arbeit gliedert sich wie folgt:

Kapitel 1: *Einleitung*
Gibt einen Überblick über Motivation, Kontext und Ziele dieser Arbeit sowie deren Aufbau.

Kapitel 2: *Grundlagen modellbasierter Softwareentwicklung*
Klärt grundlegende Begriffe und Konzepte in Bezug auf diese Arbeit. Darüber hinaus wird in existierende Werkzeuge innerhalb der modellbasierten Softwareentwicklung sowie in die Entwicklung textueller Sprachen und Werkzeuge für deren Verarbeitung eingeführt.

Kapitel 3: *Die Sprache UML/P*
Beschreibt die einzelnen Sprachen und Modellierungskonzepte der UML/P. Der Fokus liegt dabei auf der textuellen Notation und deren Vergleich mit der in [Rum04a, Rum04b] verwendeten graphischen Darstellung.

Kapitel 4: *Kontextbedingungen*
Beschreibt und diskutiert die Bedingungen, die innerhalb der Modellanalyse geprüft werden, um die Konsistenz der Modelle sicher zu stellen. Dabei wird zwischen sprachinternen und sprachübergreifenden sowie modellinternen und modellübergreifenden Bedingungen unterschieden.

Kapitel 5: *Templatebasierte Codegenerierung*
Führt in grundlegende Technologien der Codegenerierung ein. Darauf aufbauend wird ein Ansatz für eine templatebasierte Codegenerierung vorgestellt, bei dem die Templates Ausgangspunkt und zentrales Artefakt für die Definition von Generatoren bilden.

Kapitel 6: *Konzepte der Codegenerierung*
Beschreibt und diskutiert verschiedene Ansätze, wie Modelle auf einer objektorientierten Zielplattform modular in Code umgesetzt werden können. Dabei wird der Umgang mit Unterspezifikation und projektspezifischer Variabilität, die Generierung von Produktiv- und Testcode sowie ein sprachunabhängiger Ansatz für die Messung von Testüberdeckungskriterien auf Modellen behandelt.

Kapitel 7: *Das UML/P-Framework*
Beschreibt Aufbau, Nutzung und die wichtigsten technischen Details der in dieser Arbeit entwickelten Werkzeuginfrastruktur für die modellgetriebene Softwareentwicklung mit der UML/P.

Kapitel 8: *Diskussion und verwandte Arbeiten*
Diskutiert die Ergebnisse dieser Arbeit und vergleicht diese mit anderen Ansätzen im kommerziellen und wissenschaftlichen Bereich.

Kapitel 9: *Zusammenfassung und Ausblick*
Fasst die wichtigsten Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Forschungsziele sowie Weiterentwicklungen und Verwendungen der UML/P und der Werkzeuginfrastruktur.

Anhang: Führt die vollständigen Grammatiken der Sprachen der UML/P auf und enthält eine Übersicht der Templates der in Kapitel 6 diskutierten Generierungsansätze, eine Liste der Diagramm- und Quellcodemarkierungen, einen Lebenslauf sowie ein Abkürzungs-, Abbildungs-, Tabellen-, Quellcode-, Literatur- und Stichwortverzeichnis.

Kapitel 2

Grundlagen modellbasierter Softwareentwicklung

Codegenerierung und Automatisierung sind keine neuen Ideen modellbasierter Ansätze, sondern werden fast seit Anbeginn des Computerzeitalters für die Entwicklung von Softwaresystemen eingesetzt. Wurden die ersten Programme noch Hardware-spezifisch auf Lochkarten und in Maschinensprache implementiert, entstanden im Laufe der Zeit über Assembler, prozedurale und objektorientierte Sprachen immer abstraktere Beschreibungsformen für Programme. Diese werden mittels Compilern in Maschinensprache übersetzt [ASU86]. Bei dieser Übersetzung, die oftmals in mehreren Schritten erfolgt, handelt es sich im Grunde um nichts anderes als eine Codegenerierung, um die höhere Abstraktion gegenüber der Maschinensprache zu erreichen. Erleichtert wird die Übersetzung durch die Verwendung von Zwischenschichten, wie Betriebssystemen, Frameworks und Middleware, die mehrere Anweisungen zu komplexen Befehlen bündeln und über Schnittstellen zur Verfügung stellen. Ein weiterer Ansatz sind sogenannte virtuelle Maschinen, die über einen Satz von Befehlen von der eigentlichen Hardware abstrahieren. Gemeinsam mit der technischen Umgebung bilden diese die Basis für die Ausführung von Software, die allgemein auch als Plattform bezeichnet wird. Zwar können verschiedene Compiler für eine Sprache existieren, um etwa unterschiedliche Plattformen zu berücksichtigen. Anpassbar durch den Anwendungsprogrammierer sind sie im Allgemeinen jedoch nicht.

Modellierung und darauf beruhende modellgetriebene Ansätze bieten das Potential, noch weiter von der technischen Realisierung zu abstrahieren und bereits früh im Entwicklungsprozess zum Einsatz zu kommen. Aus diesem Grund ist immer wieder die Vermutung anzutreffen, dass diese Ansätze die nächste Abstraktionsstufe gegenüber heutigen GPLs darstellen und irgendwann der daraus generierten Code nicht mehr von Interesse ist, wie dies auch bei den Compileransätzen der Fall ist. In diesem Fall wären jedoch Modellierungssprachen nichts anderes als eine neue Programmiersprache und damit nicht flexibel an zukünftige Anforderungen anpassbar. Dave Thomas hat dies in [Tho04] wie folgt formuliert:

> *"The term executable specification is an oxymoron - if the specification were truly executable, it would actually be 'the thing'. Otherwise, it would merely model 'the thing', which is by definition partial and incomplete."*

Andere Arbeiten sehen die Zukunft in domänenspezifischen Sprachen (DSLs) oder domänenspezifischen Modellierungssprachen (DSMLs). Der hohe Abstraktionsgrad und die damit verbundene Produktivitätssteigerung bis zu einem Faktor von 10 gegenüber herkömmlicher Programmierung [KT08] wird durch die Kenntnis der Domäne und eine entsprechend darauf abgestimmte Codegenerierung erreicht. Allerdings wird auch hier oft der Eindruck vermittelt, dass Sprache und Generatoren nur einmal entwickelt werden müssen und danach die Generatoren im Sinne von Compilern agieren.

In dieser Arbeit wird hingegen die These vertreten, dass die Stärke von Modellierung und modellgetriebenen Ansätzen gerade in der Anpassbarkeit liegt. Eine abgeschlossene Lösung nach dem Compilerprinzip wäre zu unflexibel, um den komplexen und sich ändernden Anforderungen heutiger und zukünftiger Technologien begegnen zu können. Stattdessen werden anpassbare Sprachen sowie gut lesbare und einfach zu implementierende Generatoren benötigt, die projekt-, unternehmens-, plattform- oder domänenspezifischen Anforderungen angepasst werden können. Dieses Ziel wird mit der in dieser Arbeit entwickelten Werkzeuginfrastruktur für die modellgetriebene Entwicklung mit der UML/P verfolgt.

Im Folgenden wird ein kurzer Einblick in Entwicklung, Hintergründe und heutigen Stand der modellbasierten Softwareentwicklung gegeben. Dazu werden im Abschnitt 2.1 grundlegende Begriffe und Ziele sowie in Abschnitt 2.2 existierende Konzepte und Werkzeuge diskutiert. Abschließend erläutert Abschnitt 2.3 Verfahren für die Entwicklung von Sprachen und Werkzeugen.

2.1 Grundlegende Begriffe und Ziele

Innerhalb der Informatik ist die Bedeutung des Begriffs *Modell* nicht eindeutig geklärt [Küh05, Sei03], so dass sich in der Literatur unterschiedlichste Definitionen finden lassen:

- "*Ein Modell ist seinem Wesen nach eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems.*" [Sta73]
- "*A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.*" [Küh06]
- "*A model is a set of statements about some system under study.*" [Sei03]
- "*A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.*" [BG01]

Einige Ansichten gehen soweit, dass im Grunde alles als ein Modell betrachtet werden kann [Béz04, Béz05a]. Dies führt dazu, dass selbst Code in einer höheren Programmiersprache als ein Modell des Systems angesehen wird, da dieser den vom Compiler erzeugten Maschinencode

abstrahiert. Einer solch breiten Auslegung fehlt jedoch die abgrenzende Funktion, die einem Begriff zugrunde liegen sollte [Küh05]. Es stellt sich somit die Frage, welche Merkmale ein Modell ausmachen. Aus den obigen Definitionen lassen sich drei Hauptmerkmale von Modellen ableiten, die bereits in [Sta73] formuliert wurden:

- **Abbildungsmerkmal**: Basiert auf einem Original
- **Verkürzungsmerkmal**: Abstraktion von Eigenschaften des Originals
- **Pragmatisches Merkmal**: Schaffung für einen Zweck

Darauf aufbauend definiert [Sel03] fünf Schlüsselkriterien, die ein Modell bieten muss, um für die modellgetriebene Entwicklung von Softwaresystemen nützlich und effektiv zu sein. In Klammern sind die von [Sel03] verwendeten Begriffe angegeben:

1. **Abstraktion** (engl. "abstraction"): Auslassen oder Verstecken von Details, um die Komplexität des Zielsystems zu beherrschen.
2. **Verständlichkeit** (engl. "understandability"): Reduzierung des intellektuellen Aufwands, um das System zu verstehen. Dies hängt mit der Ausdruckskraft des Modells sowie der Sprache zusammen, aber auch mit einer gut lesbaren und intuitiven Notation.
3. **Exaktheit** (engl. "accuracy"): Die Modelle müssen im Rahmen der Abstraktion das Zielsystem zutreffend wiedergeben.
4. **Vorhersagekraft** (engl. "predictiveness"): Evaluierung von Eigenschaften des Zielsystem anhand der Modelle, die über das Offensichtliche hinausgehen.
5. **Kosteneffektivität** (engl. "inexpensive"): Einfachere Erstellung und Analyse des Modells gegenüber einer direkten Umsetzung des Zielsystems.

In dieser Arbeit wird die Ansicht vertreten, dass ein Modell nur dann als ein solches betrachtet werden kann, wenn es eine echte Abstraktion im Hinblick auf Detailauslassungen gegenüber dem Original bietet. Bei einer GPL ist dies nicht der Fall, da der Code schließlich sämtliche Details enthält, um ausführbar zu sein. Konzepte der UML/P wie Unterspezifikation oder semantische Variationspunkte sind hier nicht möglich. Für die modellgetriebene Softwareentwicklung bedeutet dies, dass ein Modell nicht sämtliche Aspekte eines Systems spezifiziert, sondern dass weitere durch die verwendeten Generatoren bestimmt werden und erst auf diese Weise ein lauffähiges Softwaresystem entsteht. Dadurch wird die Auswahl, Konfiguration, Implementierung oder Erweiterung von Generatoren Teil der Softwareentwicklung mit modellgetriebenen Ansätzen. Entsprechend wichtig ist ein einfacher und effizienter Ansatz für die Entwicklung von modifizierbaren Generatoren.

Ebenfalls nicht eindeutig geklärt ist die Granularität eines Modells gegenüber dem System [Rum04a]. So können mehrere Artefakte das Modell eines Systems bilden oder jedes Artefakt für sich als Modell bezeichnet werden. In dieser Arbeit wird letzteres angenommen, wobei jedes Modell als eine separate Datei abgelegt wird (siehe Abschnitt 3.8).

Modelle können in der Softwareentwicklung sowohl zur Spezifikation und Qualitätssicherung eines zu entwickelnden Systems (*Forward Engineering*), als auch zur Beschreibung existierender Systeme eingesetzt werden (*Reverse Engineering*). Beim *Round-trip Engineering* werden beide Vorgehen kombiniert. Dies wird angewendet, wenn Änderungen auf Modellebene und im daraus erzeugten Systemcode erfolgen können, dabei aber beide Sichten synchron gehalten werden. Weniger eindeutig ist der Begriff des *Metamodells* (vgl. [Küh05]), unter dem jedoch im Allgemeinen ein Modell verstanden wird, das eine Sprache zur Formulierung von Modellen definiert. In diesem Sinne wird der Begriff auch in dieser Arbeit verwendet.

Uneinigkeit herrscht auch bei den Begriffen *modellbasiert* und *modellgetrieben*, die in der Literatur teilweise synonym verwendet werden. Darauf aufbauend sind diverse andere Bezeichnungen zu finden, die durch das Anhängen von *E* für *Engineering* und *SE* für *Software Engineering* bzw. *D* für *Entwicklung* (engl. "*Development*") und *SD* für *Software Entwicklung* entstehen. Teilweise steht dabei das *S* auch für *System*, was dann aber ebenfalls im Sinne von Softwaresystemen verstanden wird. Welche der Bezeichnungen jeweils verwendet wird, scheint dabei eher von den Vorlieben der Autoren abzuhängen, als von einer klaren Abgrenzung der Begriffe. Im Sinne einer einheitlichen Begriffsbildung werden diese Konzepte in dieser Arbeit wie folgt klassifiziert:

- **modellbasierte Ansätze** (engl. "model-based"): MBE - MBSE - MBD - MBSD
  Ansätze, die Modelle nicht nur zu Dokumentationszwecken, sondern auch für die Softwareentwicklung selbst einsetzen. Im einfachsten Fall kann es sich dabei um informelle Modelle für die Erhebung von Anforderungen oder den Systementwurf handeln, aus denen durch manuelle Implementierung das Softwaresystem entsteht.
- **modellgetriebene Ansätze** (engl. "model-driven"): MDE - MDSE - MDD - MDSD
  Hierbei handelt es sich um eine Spezialisierung modellbasierter Ansätze, bei denen formale Modelle als ein zentrales und essentielles Artefakt in der Softwareentwicklung eingesetzt werden[1]. Die formale Spezifikation erlaubt es, die Modelle automatisiert zu verarbeiten, zu analysieren und deren Konsistenz zum Softwaresystem sicher zu stellen. Letzteres kann durch Analysen, Generierung von automatisierten Testfällen oder Generierung des Softwaresystems selbst erfolgen.

Der Begriff *Engineering* hebt den Ingenieur-Charakter des jeweiligen Ansatzes stärker hervor, ist aber im Wesentlichen synonym zu *Entwicklung* zu sehen. Durch das Hinzufügen von *System* bzw. *Software* wird hingegen ein stärkerer Fokus auf die Entwicklung von Softwaresystemen gesetzt, während die kürzeren Bezeichnungen prinzipiell auch Businessabläufe oder Hardwareanteile mit einschließen und somit als Oberbegriffe fungieren.

Insgesamt ist es das Ziel modellbasierter Ansätze, die Entwicklung von Softwaresystemen mit Hilfe der höheren Abstraktion der Modelle zu vereinfachen. Dabei sollen die Modelle helfen, auf die problemrelevanten Aspekte der Software hinsichtlich der Anforderungen des Fach-

[1] Dieser Fokus ist konform zu [Sel03, Sel06], wo die Behandlung von Modellen als primäres Produkt der Softwareentwicklung als die "definierende Charakteristik" modellgetriebener Ansätze angesehen wird.

oder Anwendungsgebiets zu fokussieren. Dieses Gebiet wird im Folgenden als *Problemdomäne* bezeichnet. GPLs sind hingegen eher auf die *Implementierungsdomäne* ausgerichtet, die für die technische Umgebung der lauffähigen Software steht [Sel03, Sel06]. Die Konzepte rund um modellgetriebene Ansätze dienen schließlich dazu, den Nutzen der Modelle durch Automatisierung von Analysen bis hin zur Generierung des lauffähigen Softwaresystems zu erhöhen. Insbesondere kapseln die Generatoren dabei das Wissen, das für die Überbrückung der Problem- hin zur Implementierungsdomäne notwendig ist. Aus diesem Grund wird den modellgetriebenen Ansätzen eine hohe Bedeutung für die Effizienzsteigerung in der Softwareentwicklung zugeschrieben [Sel03].

2.2 Existierende Konzepte und Werkzeuge

Die Idee der modellbasierten Entwicklung von Softwaresystemen geht auf das *CASE-Konzept* (Computer-Aided Software Engineering) der frühen 80er Jahre zurück [Béz05b]. Davor wurde Modellierung im Wesentlichen nur als Instrument zur Dokumentation und Planung von Softwaresystemen gesehen [Dem79, GS79]. Mit CASE wurde hingegen erstmalig versucht, formale graphische Modelle in den Entwicklungsprozess zu integrieren, um frühzeitige Analysen zu ermöglichen und zumindest Teile des Zielsystems daraus automatisch abzuleiten. Aufgrund diverser Probleme insbesondere der Werkzeugunterstützung hatte dieser Ansatz auf die kommerzielle Softwareentwicklung jedoch nur einen geringen Einfluss [Sch06]:

- erschwerte Fehlerbehebung und Weiterentwicklung der generierten Systeme aufgrund unausgereifter und nicht anpassbarer Transformationstechniken
- unzureichende Skalierbarkeit von Modellierungstechniken und Werkzeugunterstützung für die Entwicklung großer Softwaresysteme
- keine Unterstützung für verteilte Entwicklung im Team
- Einschränkung des generierten Systems auf bestimmte Plattformen durch Verwendung proprietärer Zwischenschichten
- fehlende Erweiter- oder Anpassbarkeit der Modellierungssprachen

Die Sprachen und Werkzeuge waren somit insbesondere zu unflexibel, um sie für die Entwicklung von Softwaresystemen für unterschiedliche Domänen oder Plattformen einsetzen zu können. Darüber hinaus sind in diesem Kontext viele neue Modellierungssprachen wie [RBP+91, SM91] entstanden, die den Überblick über die Werkzeuge und deren Möglichkeiten weiter erschwerte.

Anfang der 90er begann die Object Management Group (OMG), eine weltweite Vereinigung verschiedener Organisationen zur Erarbeitung von Standards in der IT-Branche, diese große Bandbreite von Modellierungssprachen in einer einheitlichen Sprachfamilie zu vereinen. Die Version 1.0 dieser "*Unified Modeling Language*" (UML) wurde 1997 veröffentlicht [UML97, BRJ97] und seitdem kontinuierlich überarbeitet und erweitert.

Auf dieser Basis bauen die meisten heutigen modellbasierten und -getriebenen Ansätze auf. Auch wenn diese als eine Weiterentwicklung der frühen CASE-Konzepte gesehen werden

können [FR07], wird dieser Begriff aufgrund seiner negativen Assoziation mittlerweile vermieden [Fow03]. Dennoch können auch heutige Ansätze noch von den Erfahrungen und Fehlern der ehemaligen CASE-Forschung profitieren.

Neben Konzepten wie [TB03] oder [Amb04] hat insbesondere ein modellgetriebener Ansatz in den letzten Jahren besonders viel Aufmerksamkeit erregt: die *Model Driven Architecture* (MDA, [OMG03, KWB03]) der OMG. Deren Kernkonzept besteht in der Modellierung auf unterschiedlichen Abstraktionsstufen, indem abstrakte Modelle nahe der Problemdomäne durch Transformationen in Modelle bezogen auf eine bestimmte Zielplattform konkretisiert werden - eine Idee, die auf die Shlaer-Mellor-Methode [MS91] zurückgeht. Insbesondere werden dabei folgende Modelle unterschieden[2]:

- **PIM** (Platform Independent Model): Das Plattform-unabhängige Modell spezifiziert die fachlichen Aspekte eines Softwaresystems, ohne dabei auf technische Details einer spezifischen Zielplattform für die lauffähige Software einzugehen. Dadurch ist es näher an der Problemdomäne als das PSM.
- **PSM** (Platform Specific Model): Das Plattform-spezifische Modell entsteht aus dem PIM durch Transformationen, indem technische Aspekte, die für die Umsetzung der Software auf einer bestimmten Plattform notwendig sind, hinzugefügt werden. Im Vergleich zum PIM ist es somit näher an der Implementierungsdomäne.

Dieses konzeptionelle Framework baut neben der UML als Modellierungssprache auf einer Reihe von weiteren Standards und Spezifikationen der OMG auf. Dazu gehören[3]:

- **MOF** (Meta Object Facility, [OMG06]):
  An Klassendiagramme angelehnte Sprache zur Beschreibung von Metamodellen. Diese wird u.a. für die Spezifikation des UML-Metamodells verwendet.
- **XMI** (XML Metadata Interchange, [OMG07]):
  Ein auf XML[4] basierendes textuelles Format für die Speicherung von MOF-basierten Modellen und deren Austausch zwischen Werkzeugen.
- **QVT** (Query, View, and Transformation, [OMG11a]):
  Kombination aus drei Sprachen für die Spezifikation von Modelltransformationen, wobei die Modelle für die Anwendung in XMI vorliegen müssen:
  - *Q*: Sprache zur Ermittlung von Informationen aus den Modellen
  - *V*: Sprache zur Beschreibung von Sichten auf Modellen
  - *T*: Sprache zur Beschreibung von Transformationen zwischen den Sichten

[2] Ebenfalls in [OMG03] erwähnt wird CIM (Computation Independent Model), das für die Erfassung von Anforderungen im Vokabular der Problemdomäne gedacht ist, ohne bereits auf die Struktur des Softwaresystems einzugehen, und PM (Platform Model), das eine technische Plattform und deren Anforderungen beschreibt.

[3] Die Referenzen verweisen jeweils auf die aktuell vorliegende finale Spezifikation.

[4] Extensible Markup Language: Sprache für die Spezifikation von textuellen hierarchischen Dokumentstrukturen

Die Bezeichnungen PIM und PSM sind abhängig vom Kontext. So kann sich die erste Transformation auf eine allgemeinere Plattform wie CORBA [OMG08a] beziehen. Dabei handelt es sich um eine Programmiersprachen-unabhängige Spezifikation einer objektorientierten Middleware, die in einer weiteren Transformation auf eine konkrete Zieltechnologie übertragen werden muss. Das CORBA-spezifische Modell dient dieser zweiten Transformation somit wiederum als PIM. Auf diese Weise kann die Verfeinerung der Modelle auch in mehreren Schritten erfolgen, wobei der MDA-Standard nicht festlegt, ob diese automatisiert, teil-automatisiert oder manuell durchgeführt werden.

Die Modelltransformationen sind Regeln, die Elemente eines Quellmodells auf Elemente eines Zielmodells abbilden. Sie basieren auf den Metamodellen der jeweiligen Modellierungssprachen, um sie unabhängig von den konkreten Modellen spezifizieren zu können. Dies ist keine neue Idee der OMG, sondern geht auf [Lem98] zurück. Darauf aufbauend sind seitdem eine Vielzahl an Transformationsansätzen [Kön05, TEG+05, LG08] und -sprachen [KBC05, LS06, JABK08, KPP08a] entstanden. Formal basieren fast alle diese Ansätze auf Graph Grammatiken und insbesondere den Triple Graph Grammatiken [Sch95], mit denen Beziehungen zwischen zwei unterschiedlichen Typen von Modellen dargestellt werden können. Auch QVT lässt sich in diesem Kontext betrachten [GK10]. Die Sprachen für die Spezifikation der Transformationsregeln sind im Allgemeinen textuell oder graphisch. Darüber hinaus existieren auch rein API-basierte Ansätze [ABE+06]. Eine allgemeine Kategorisierung von Modelltransformationen wurde in [CH03, CH06] aufgestellt. [JUH10] bietet einen Überblick und Kategorisierung der Modelltransformationsansätze im Rahmen der MDA.

Die Implementierung ist in den Standards der OMG nicht vorgegeben, so dass deren Umsetzung von den Werkzeugentwicklern abhängt. Dass dies aufgrund der Komplexität und des Umfangs der Standards keine leichte Aufgabe ist, zeigt eine umfassende Studie über die Eigenschaften aktueller UML-Modellierungswerkzeuge und deren Konformität zum UML-Standard [EES09]. Danach erreichen nur 4 von 68 untersuchten Werkzeugen eine akzeptable Bewertung, wobei keines der Werkzeuge die UML-Spezifikation vollständig erfüllt. Genauso schwierig und fehleranfällig ist damit aber auch das Verständnis und die Nutzung der UML für den Modellierer und motiviert nicht zuletzt die deutliche Vereinfachung, die die UML/P mit sich bringt. Darüber hinaus zeigt die Studie, dass der Markt der UML-Werkzeuge noch sehr starken Veränderungen unterworfen ist. So wurden über 60% der ursprünglich 200 identifizierten Werkzeuge aufgrund fehlender Weiterentwicklung oder Verfügbarkeit in der Studie nicht berücksichtigt.

Die Unterstützung für modellgetriebene Ansätze setzt sich bei den Werkzeugen nur langsam durch. So unterstützen viele nur eine rudimentäre Codegenerierung von Klassen- und Methodenrümpfen. Verhalten wird in diesen Fällen im Allgemeinen manuell als Annotation an den Modellen oder im generierten Code ergänzt. Für letzteres sehen die Werkzeuge oft geschützte Regionen im generierten Code vor, um die manuellen Ergänzungen bei einer erneuten Generierung zu erhalten. Diese Form der Verhaltensimplementierung bezieht sich jedoch auf das generierte System und damit auf die Implementierungsdomäne. Dadurch muss bei einem Wechsel der Plattform dieser Code entsprechend angepasst werden.

Andere Werkzeuge wie iUML [iUML] oder Fujaba [Fuj] ermöglichen hingegen auch die Codegenerierung aus abstrakten Verhaltensbeschreibungen [RFW+04, KNNZ99]. Für die Modellierung von Verhaltens wird in iUML eine eigene Aktionssprache verwendet. Fujaba bietet hierfür Story Diagramme, die mit den Aktivitätsdiagrammen der UML verwandt sind. Tabelle 2.1 führt diese und einige weitere der bekanntesten und weit verbreitetsten Werkzeuge zur modellbasierten Softwareentwicklung auf.

Produkt	*Version*	*Vertrieb*	*Codegenerierung*	
			Architektur	*Verhalten*
ArgoUML [AU]	0.32	Open-Source	✓	-
Artisan Studio [AS]	7.2	kommerziell	✓	✓
Enterprise Architect [EA]	9.0	kommerziell	✓	✓
Fujaba [Fuj]	5.0	Open-Source	✓	✓
Gentleware Apollo [GA]	2.0	kommerziell	✓	-
IBM Rational Rhapsody [RR]	7.6	kommerziell	✓	✓
iUML [iUML]	2.4	kommerziell	✓	✓
MagicDraw UML [MD]	17.0	kommerziell	✓	-
objectiF [obj]	7.1	kommerziell	✓	-
Papyrus [Pap]	0.9	Open-Source	-	-
Poseidon for UML [Pos]	8.0	kommerziell	✓	-
Together [Tog]	2008	kommerziell	✓	-

Tabelle 2.1: Verbreitete UML-Werkzeuge

Für die Modellierung bieten die Werkzeuge graphische Editoren an. Dabei erfolgt die Eingabe textueller Anteile wie Namen, Annotationen oder anderer Eigenschaften von Modellelementen im Allgemeinen formularbasiert. Dasselbe gilt für die Ausführung und Parametrisierung der Codegenerierung. Daneben unterstützen insbesondere die kommerziellen Werkzeuge oft Reverse- und Round-trip Engineering im Rahmen der Architekturmodellierung. Die Definition eigener Generatoren ist dagegen nur selten möglich.

Einige Werkzeuge beschränken sich allein auf die Modellierung. Die fehlende Unterstützung der Codegenerierung kann bei vorhandenem XMI-Export durch andere Werkzeuge wie MOFScript [MS] oder AndroMDA [AM] nachgerüstet werden. Basierend auf den XMI-Daten erlauben diese die Definition und Ausführung von Codegeneratoren. MOFScript nutzt für den Modellimport das Eclipse Modeling Framework (EMF, [EMF, SBPM08]), das mit Ecore ein zum MOF-Standard kompatibles Metamodell für die Entwicklungsumgebung Eclipse [Ecl] zur Verfügung stellt. Die XMI-Daten der Modelle werden so in eine programminterne Darstellung überführt und sind über eine API für die mit MOFScript definierten Codegeneratoren nutzbar. Viele weitere Werkzeuge etwa für Modellanalysen oder -transformationen nutzen EMF und Ecore auf die gleiche Weise, so dass auf dieser Basis theoretisch eine umfangreiche Werkzeuginfrastruktur für

die modellgetriebene Softwareentwicklung aufgebaut werden kann [Gro09]. Allerdings ist bisher bei den meisten Werkzeugen der Im- und Export nicht vollständig konform zum XMI-Standard, so dass ein werkzeugübergreifender Austausch von Modellen oft noch nicht problemlos möglich ist [EES09, LLPM06]. Darüber hinaus besteht laut [GF10] weiteres Verbesserungspotenzial insbesondere bei der Versionierung, Zusammenführung und Validierung von Modellen, sowie bei der Bedienbarkeit der Editoren.

2.3 Entwicklung von Sprachen und Werkzeugen

Die modellbasierte Softwareentwicklung ist nicht auf graphische Modelle beschränkt. Auch textuelle Sprachen können die in Abschnitt 2.1 beschriebenen Eigenschaften von Modellen erfüllen und somit für die abstrakte Beschreibung von Softwaresystemen eingesetzt werden. Im Allgemeinen werden graphische und textuelle Elemente gemeinsam verwendet. So können graphische Modelle textuelle Anteile etwa zur Darstellung von Namen oder Eigenschaften enthalten. Auch der gemeinsame Einsatz graphischer und textueller Modelle ist möglich. In der UML ist z.B. mit der OCL eine textuelle Sprache für die Beschreibung von Bedingungen vorgesehen (siehe auch Abschnitt 3.6).

Letztendlich handelt es sich bei der Frage nach der Notation nur um eine Darstellungsform. So können die einem Modell zugrunde liegenden Konzepte und Informationen sowohl graphisch als auch textuell ausgedrückt werden. Ein Beispiel hierfür ist XMI, das die Speicherung von UML-Modellen als textuelle Dateien erlaubt (siehe Abschnitt 2.2). Die XMI-Darstellung wird bei größeren Modellen jedoch schnell unübersichtlich, so dass eine Modellierung auf dieser Basis nicht sinnvoll ist.

[Kra10] stellt ein Vorgehensmodell für die Entwicklung formaler Sprachen und darauf aufbauender Werkzeuge vor, das sich an dem Vorgehen bei der Entwicklung von Softwaresystemen orientiert. So sind in einer Analysephase zuerst die Anforderungen an die Sprache und deren erforderlichen Eigenschaften zu ermitteln. Dies bildet die Basis für die Sprachdefinition und die Implementierung entsprechender Werkzeuge, die gegenüber den Anforderungen validiert und iterativ weiter entwickelt werden können. Dabei setzt sich die Definition einer formalen Sprache aus folgenden Bestandteilen zusammen [HR00, HR04b]:

- **Konkrete Syntax**: Notation der Sprache (textuell und/oder graphisch)
- **Abstrakte Syntax**: Programminterne Repräsentation der Sprache
- **Kontextbedingungen**: Regeln für zulässige Ausdrücke einer Sprache
- **Semantik**: Bedeutung der Sprache

Aus Sicht der Modellierungssprachen werden die Modelle somit in der konkreten Syntax verfasst und im Rahmen der Sprachverarbeitung in die abstrakte Syntax überführt. Diese bildet die Basis für die gesamte weitere Werkzeuginfrastruktur und wird somit für die Implementierung von automatisierten Analysen, Transformationen oder Codegeneratoren eingesetzt. Einen Teil der

Analysen bilden die Kontextbedingungen, die fehlerhafte Konstrukte und mögliche Inkonsistenzen in den Modellen spezifizieren. Diese werden auf Basis der abstrakten Syntax programmiert oder anhand des Metamodells in einer Bedingungssprache wie OCL beschrieben. Die Semantik einer Sprache kann schließlich durch eine sogenannte semantische Abbildung der Sprachelemente auf eine Menge von bekannten Elementen definiert werden [HR04b]. Letztere wird als semantische Domäne bezeichnet. Für viele Sprachen besteht die semantische Abbildung aus einer Beschreibung in einer natürlichen Sprache. Es existieren aber auch formale Ansätze [Rum96, HR04b]. Die Semantik ist für ein präzises und eindeutiges Verständnis der Modelle von Bedeutung und muss insbesondere bei der Umsetzung von Werkzeugen erhalten bleiben.

Die Entwicklung von Werkzeugen zur Erstellung und Verarbeitung von Modellen unterscheidet sich hinsichtlich graphischer und textueller Sprachen vor allem in Syntaxdefinition und Sprachverarbeitung:

- *Graphische Sprachen* setzen für die Verwendung einen entsprechenden Editor voraus, der die Notation unterstützt. Dieser realisiert auch die Abbildung der Modellelemente auf die abstrakte Syntax. Theoretisch können Editor und abstrakte Syntax manuell implementiert werden. Es existieren jedoch auch diverse Werkzeuge, die die Entwicklung graphischer Sprachen unterstützen. Einige der bekanntesten sind MetaEdit+ [KT08], Microsoft DSL Tools [CJKW07], GME (Generic Modeling Environment, [LMB$^+$01]) und openArchitectureware [oAW], das seit kurzem Teil des Eclipse Modeling Projects ist [Gro09]. Die Sprachdefinition erfolgt in diesen Werkzeugen im Allgemeinen in Form eines an Klassendiagramme angelehnten Metamodells. Dieses kann auf MOF oder Ecore basieren, während andere wie MetaEdit+, Microsoft DSL Tools und GME auf ein eigenes Metaformat setzen. Indem Teilen des Metamodells graphische Elemente zugeordnet werden, kann ein Editor und die abstrakte Syntax aus dem Metamodell generiert werden. Die Editoren sind nur innerhalb der Werkzeuge selbst, als eigenständige Applikation oder wie bei openArchitectureware innerhalb einer bestimmten Entwicklungsumgebung wie Eclipse lauffähig.
- *Textuelle Sprachen* sind für die Nutzung nicht an einen bestimmten Editor gebunden, sondern werden über einen Lexer und Parser verarbeitet. Der Lexer identifiziert in der lexikalischen Analyse die Wörter innerhalb des Textes. Darauf folgt die syntaktische Analyse durch den Parser, der schließlich die textuellen Modelle in die abstrakte Syntax überführt [ASU86].

  Für die Definition der konkreten Syntax haben sich kontextfreie Grammatiken etabliert, die die zulässigen Wörter und den Aufbau der Sprache beschreiben. Die Grammatiken werden in Sprachen wie EBNF (Extended Backus-Naur-Form, [EBNF]) formal spezifiziert. Aus dieser Spezifikation können sogenannte Parser-Generatoren wie ANTLR [PQ95] die Infrastruktur für die Sprachverarbeitung automatisch erzeugen. Die abstrakte Syntax wird entweder separat spezifiziert oder aus der Grammatik abgeleitet. Andere Werkzeuge wie EMFText [ET] oder das in openArchitectureware integrierte XText [XT] bieten darüber hinaus zusätzliche Infrastruktur für die Entwicklung von textuellen Sprachen. Diese erlauben

u.a. auch die automatische Erstellung von Editoren, die die Nutzung der Sprache durch Syntaxhervorhebung oder Autovervollständigung unterstützen.

Vertiefende Details zur Entwicklung und Nutzung von Sprachen finden sich in [FP10, MHS05]. Darüber hinaus wurden in [KKP+09] Richtlinien für die methodische Entwicklung formaler Sprachen aufgestellt.

Im Hinblick auf die UML/P, die sich aus mehreren Sprachen zusammensetzt, ist insbesondere eine kompositionale Entwicklung dieser Sprachen von Interesse. Dabei wird jede Sprache als einzelne Komponente entwickelt und erst dann zu einem Gesamtwerkzeug integriert. Dies ermöglicht eine getrennte Wiederverwendung sowie eine unabhängige Austausch- und Weiterentwickelbarkeit der Sprachen. Eine solche kompositionale Sprachentwicklung wird bisher von den Werkzeugen nicht oder nur in einem sehr geringem Umfang unterstützt. Aus diesem Grund wurde parallel zu dieser Arbeit das MontiCore-Framework entwickelt, für das die UML/P eines der wesentlichen Architekturtreiber war [GKRS06, GKR+06, GKR+08, Kra10, Völ11].

Die UML/P ist in [Rum04a, Rum04b] weitgehend graphisch konzipiert. Textuelle Sprachen bringen jedoch diverse Vorteile für die Sprachentwicklung und -nutzung mit sich, die bereits in [GKR+07] ausführlich diskutiert wurden. Während graphische Modelle im Allgemeinen einen leichteren und intuitiveren Überblick bieten und damit vor allem zur Kommunikation und Dokumentation geeignet sind, überwiegen die Vorteile von textuellen Notationen im Hinblick auf die modellgetriebene Entwicklung von Softwaresystemen:

- *Fokussierung auf fachliche Inhalte*, während bei graphischen Notationen oft viel Aufwand in die Formatierung fließt, die sich bei textuellen Notationen zudem leicht automatisieren lässt.
- *Schnellere und kompaktere Erstellung*, ohne einen zeitaufwändigen Wechsel zwischen verschiedenen Formularen, Menüstrukturen und Fenstern. Gleichzeitig ist damit die Modellerstellung nicht an einen bestimmten Editor gebunden.
- *Nahtlosere Integration von mehreren Sprachen* als dies bei einer Kombination von textuellen und graphischen Notationen möglich ist.
- *Unterstützung von Teamarbeit* durch konventionelle Versionsverwaltungssysteme wie CVS und SVN, während ähnlich ausgereifte Verfahren für graphische Modelle insbesondere im Hinblick auf das Anzeigen und automatische Zusammenführen von Änderungen bisher nicht existieren (siehe auch Abschnitt 8.4).

Aus diesen und weiteren in [GKR+07] erläuterten Gründen wurde MontiCore für die Entwicklung von textuellen Sprachen konzipiert. Auch für die UML/P wurde eine textuelle Notation entwickelt. Diese und die darauf aufbauende Werkzeuginfrastruktur für eine modellgetriebene Entwicklung wird in den folgenden Kapiteln vorgestellt.

Kapitel 3

Die Sprache UML/P

Von der OMG (Object Management Group) als standardisierte Sprachfamilie[1] für die Entwicklung von Softwaresystemen entworfen, hat sich die UML (Unified Modeling Language [OMG10c, OMG10d]) in den letzten Jahren zu einem etablierten Standard in Forschung und Industrie entwickelt. Die UML bietet dabei ein Portfolio hauptsächlich graphischer aber auch einiger weniger textueller Sprachen, die sich für die Planung und Modellierung von Softwaresystemen eignen. Trotz oder gerade wegen dieser Vielzahl an Sprachen, deren Anwendungsbereiche sich teilweise überschneiden, erscheint die UML jedoch zu schwergewichtig, um auch eine agile Umsetzung von in Umfang und Zielsetzung stark voneinander abweichende Projekte gleich gut zu unterstützen. Vor diesem Hintergrund wurde die UML/P entwickelt.

Die UML/P ist ein Sprachprofil der UML, die es sich zum Ziel gesetzt hat, modellbasierte Ansätze und eine agile Vorgehensweise miteinander zu vereinen [Rum04a, Rum04b]. Hierzu wurde die UML auf einen Kern von Modellierungssprachen und Sprachkonstrukten reduziert, die sich für eine Modellierungs-, Test- und Implementierungsbeschreibung besonders eignen. Diese Teilmenge der UML wurde darüber hinaus in ihrer Semantik präzisiert und mit der Programmiersprache Java kombiniert. Das Ergebnis ist ein Profil, mit dem ein zu entwickelndes Produktionssystem nicht nur geplant und modelliert, sondern auch vollständig implementiert werden kann. Die UML/P lässt sich damit als High-Level-Programmiersprache einsetzen.

Einen Überblick über die Einsatzzwecke der einzelnen Diagrammarten bzw. Sprachen gibt die Abbildung 3.1. Danach umfasst die UML/P die folgenden Sprachen:

- **Klassendiagramme (CD)** beschreiben die Architektur des Produktionssystems und bilden damit das Grundgerüst des Systems, auf das alle anderen Diagramme und Sprachen aufbauen.
- **Objektdiagramme (OD)** ermöglichen die Definition exemplarischer Daten, die sich etwa für die Spezifikation von Testfällen eignen.
- **Statecharts (SC)** beschreiben das Verhalten des Produktionssystems.

[1] Eine Sprachfamilie fasst mehrere Sprachen für eine gemeinsame Verwendung zusammen.

- **Sequenzdiagramme (SD)** modellieren exemplarische Abläufe, woraus sich - häufig zusammen mit Objektdiagrammen - vollständige Testfälle ableiten lassen.
- **Object Constraint Language (OCL/P)** wird in Kombination mit den Diagrammen genutzt, um zusätzliche Bedingungen oder Invarianten auszudrücken.
- **Java (Java/P)** kann an verschiedenen Stellen in den Diagrammen genutzt werden, etwa um Methodenrümpfe in Klassendiagrammen zu spezifizieren. Theoretisch ist aber auch der Einsatz beliebiger anderer Sprachen wie C++ denkbar.

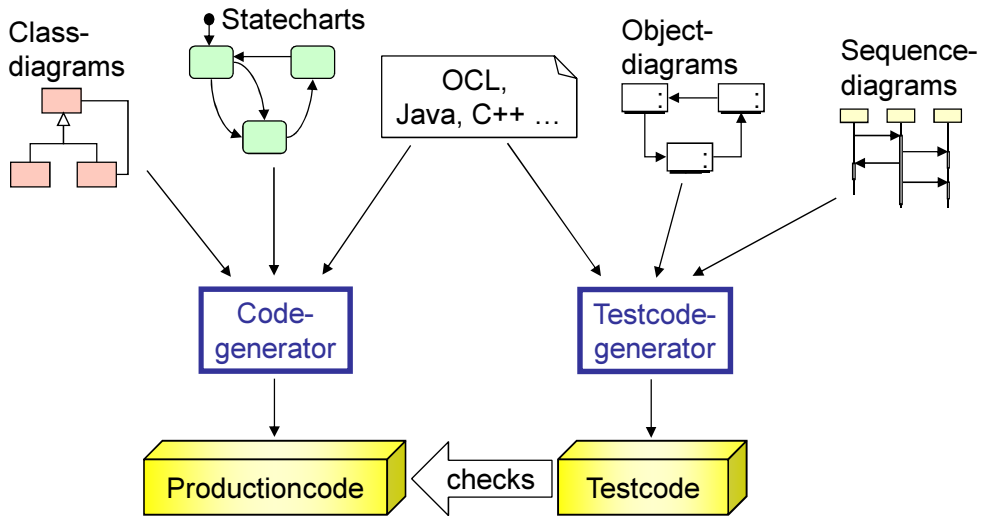


Abbildung 3.1: UML/P-Übersicht (adaptiert aus [Rum04a])

Im Rahmen der vorliegenden Arbeit wurde mit Hilfe des MontiCore-Frameworks [Kra10, GKR+06] eine textuelle Fassung der UML/P entworfen. Diese basiert auf den Grammatiken im Anhang C und wird in den Abschnitten 3.1-3.7 anhand eines durchgehenden Beispiels vorgestellt. Abschnitt 3.5 behandelt dabei eine Sprache zur Spezifikation von Testfällen, die zusätzlich zu den oben aufgeführten Sprachen eingeführt wurde, um modellbasierte Tests auf Basis von Objekt- und Sequenzdiagrammen sowie Java/P zu ermöglichen. Die gemeinsame Bedeutung der Sprachen wird im Abschnitt 3.8 behandelt. Darauf aufbauend entstand eine Werkzeuginfrastruktur, die die praktische Nutzung der UML/P für die Planung und Entwicklung von Softwaresystemen erlaubt. Obwohl dabei von Anfang an auf eine größtmögliche Konformität zu [Rum04a, Rum04b] geachtet wurde, haben sich im Laufe der Zeit durch Entwicklungen in der Softwaretechnik sowie durch die Transformation in eine textuelle Syntax einige Änderungen zur ursprünglichen UML/P ergeben, die abschließend in Abschnitt 3.9 zusammengefasst werden.

3.1 Klassendiagramme

Klassendiagramme (CD) ermöglichen die Beschreibung der Struktur eines Softwaresystems. Diese bildet die Basis, auf der alle anderen Diagramme und Sprachen der UML/P aufbauen. Damit

sind Klassendiagramme die wichtigste Notation innerhalb der UML/P. Im Folgenden werden die Sprachelemente und -konzepte anhand der Beispielarchitektur einer kleinen Applikation mit dem Namen "TripleLogo" erläutert, die die Idee der funktionalen Programmiersprache Logo erweitert.

Logo wurde 1967 von Wally Feurzeig und Seymour Papert als einfache Programmiersprache für die Lehre entwickelt [FL72, LL77]. Die Sprache ermöglicht es, mit Hilfe weniger Kommandos eine Schildkröte innerhalb einer graphischen Oberfläche zu bewegen und dabei Linien zu zeichnen. Kontrollstrukturen wie Schleifen und Bedingungen, sowie Funktionen und Prozeduren erlauben die Definition von komplexen Abläufen. Durch die Einfachheit der Sprache sowie die direkte visuelle Rückmeldung eignet sich Logo besonders für die Vermittlung von Konzepten der Informatik für Programmieranfänger.

Die im Folgenden entwickelte Variante "TripleLogo" weicht insofern von dem ursprünglichen Logo ab, als dass anstatt einem insgesamt drei Tiere mit jeweils spezifischen Eigenschaften angesteuert werden können (siehe Abbildung 3.3). Dabei unterscheiden sich die Tiere in der Art der gezeichneten Linie sowie dem Verhalten beim Verlassen des Zeichenfeldes (siehe Tabelle 3.2).

Tier	*Linienart*	*Randkontakt*
Schildkröte (engl. "Turtle")	gerade	180° Drehung
Schlange (engl. "Snake")	geschwungen	90° Rechtsdrehung
Vogel (engl. "Bird")	gestrichelt	zufällige Position innerhalb des Zeichenfeldes

Tabelle 3.2: TripleLogo: Verhalten der Tierarten

Auch wenn Verhalten für die Strukturbeschreibung in Klassendiagrammen keine direkte Rolle spielt, so haben diese Anforderungen doch Einfluss auf Designentscheidungen, die bei der Modellierung der Systemarchitektur getroffen werden müssen. Klassendiagramme orientieren sich dazu an den Konzepten objektorientierter Programmiersprachen, die ein System als Menge von Objekten betrachten. Diese kapseln wiederum Zustände und Funktionen des Systems in Form von Attributen und Methoden. Die gültige Struktur der Objekte beschreiben Klassen, Interfaces und Enumerationen, die die Kernelemente innerhalb von Klassendiagrammen darstellen und im Folgenden zusammenfassend als Typen bezeichnet werden. Interfaces bilden dabei reine Schnittstellenbeschreibungen für Klassen und Enumerationen, die Attribute und Signaturen von Methoden umfassen, ohne aber interne Zustände oder Funktionalitäten von Objekten darstellen zu können. Auch Enumerationen besitzen keine internen Zustände, sondern nur eine festgelegte, endliche Menge von sortierten Werten, können aber im Gegensatz zu Interfaces Funktionen implementieren. Klassen hingegen bieten sowohl Methodenimplementierungen als auch interne Zustände, und stellen damit den Bauplan für konkrete Objekte eines Systems dar. Die Erzeugung von Objekten (Instanzen) aus einer Klasse wird als Instanziierung bezeichnet. Dies ist bei Interfaces und Enumerationen nicht möglich. Neben diesen selbst definierten Typen können in der UML/P auch eine Reihe sogenannter primitiver Datentypen wie z.B. `boolean` für Wahrheitswerte

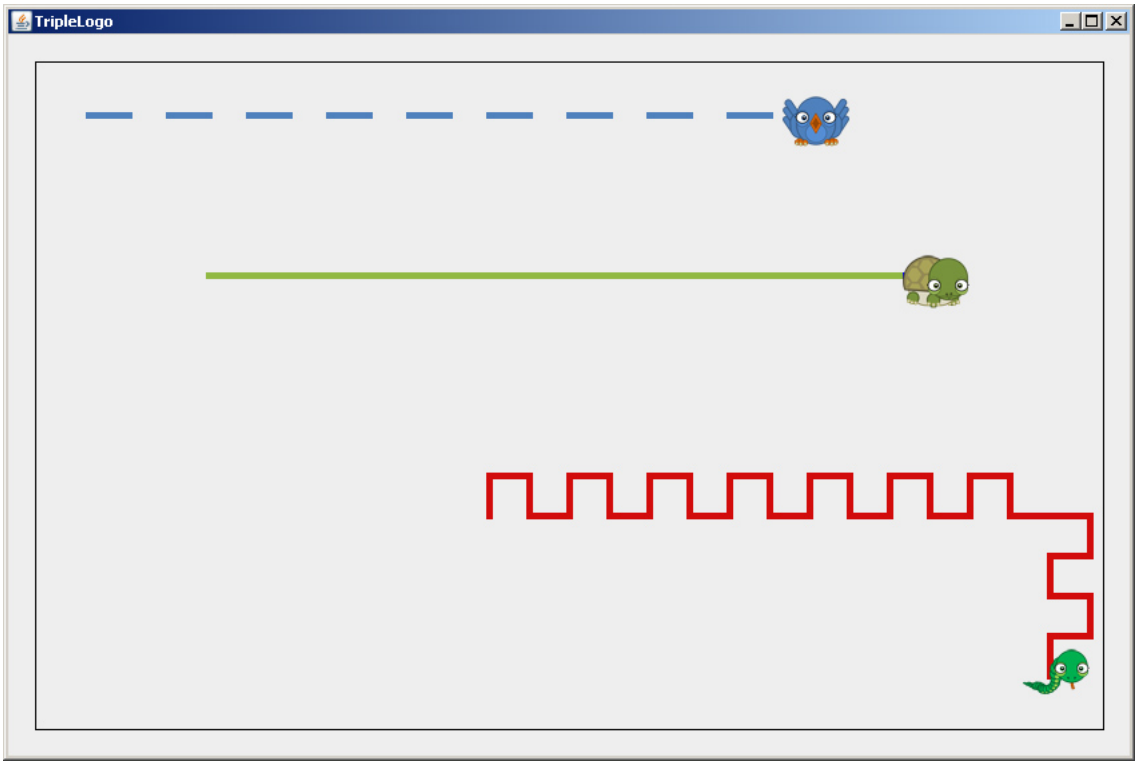


Abbildung 3.3: Screenshot der TripleLogo-Anwendung

oder `int` für natürliche Zahlen genutzt werden, die der Programmiersprache Java entlehnt wurden.

Abbildung 3.5 zeigt die vollständige Architektur der TripleLogo-Anwendung als graphisches UML/P-Klassendiagramm, das weitgehend der textuellen Fassung im Quellcode 3.4 entspricht. Letztere basiert auf der Grammatik im Anhang C.4, die in Quellcode 3.6 in komprimierter Form dargestellt ist. Neben den genannten Typarten, die in der textuellen Fassung durch die Schlüsselwörter `class`, `interface` und `enum`, gefolgt von einem eindeutigen Namen eingeleitet werden, sind in dem Klassendiagramm verschiedene Formen von Beziehungen zwischen diesen Elementen definiert, die ein weiteres wichtiges Strukturierungsmittel in der UML/P darstellen. So stehen die drei Tierarten `Turtle`, `Bird` und `Snake` in einer Vererbungsbeziehung mit der Klasse `Animal`. Jede der in diesem Fall auch als Subklassen oder Subtypen bezeichneten Tierarten erhalten dadurch die Attribute und Methoden der Superklasse (bzw. des Supertyps) `Animal`, ohne dass diese nochmal explizit aufgeführt werden müssen. Subklassen haben dabei die Möglichkeit, die geerbten Eigenschaften zu erweitern oder zu überschreiben. Neben Klassen sind auch Interfaces als Supertypen erlaubt, wobei man in diesem Fall von einer Interface-Implementierung spricht. Auf der anderen Seite dürfen Interfaces nur andere Interfaces erweitern sowie Enumerationen Interfaces implementieren. Andere Vererbungsbeziehungen sind nicht erlaubt.

Die textuelle Notation der UML/P orientiert sich im Sprachstil an der Programmiersprache Java. Vererbungsbeziehungen werden deshalb durch die Schlüsselwörter `extends` bzw. `imple-`

```
package mc.tl;

import java.awt.Color;
import java.lang.String;

classdiagram TripleLogo {

  class Controller {
    public void run();
    public boolean check(Animal a);
    public void update(Animal a);
    public void place(Animal a);
  }

  class Position {
    public int x;
    public int y;
  }

  abstract class Animal {
    public boolean visible;
    public Color color;
    protected readonly String species;

    public void meet(Animal a) {}
    public void turn(Rotation r);
    protected void step();
    public abstract void move(int distance);
    public abstract void reverse();
  }

  class Turtle extends Animal {
    protected readonly String species = "Turtle";
  }

  class Bird extends Animal {
    protected readonly String species = "Bird";
  }

  class Snake extends Animal {
    protected readonly String species = "Snake";
  }

  enum Rotation {
    LEFT, RIGHT;
  }

  enum Orientation {
    NORTH, SOUTH, EAST, WEST;
  }

  association Controller <-> Animal [0..3];
  association Animal -> Orientation;
  association Animal -> Position;

}
```

Quellcode 3.4: Architektur des TripleLogo-Beispiels (textuell)

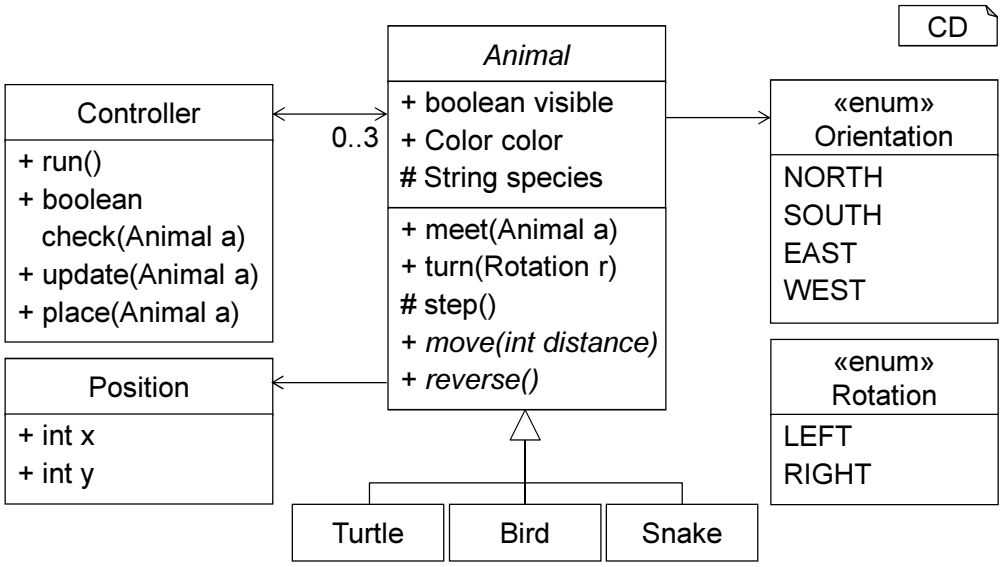


Abbildung 3.5: Architektur des TripleLogo-Beispiels (graphisch)

`ments` für den Spezialfall der Interface-Implementierung direkt an den Subtypen notiert. Dabei können jeweils mehrere Klassen oder Interfaces als Supertypen angegeben werden. Auch die Notation der Attribute und Methoden ist an Java angelehnt, indem anders als im UML-Standard der OMG [OMG10d] der Attributtyp vor dem Attributnamen bzw. der Rückgabetyp vor dem Methodennamen und den -parametern angegeben oder ein fehlender Rückgabetyp durch das Schlüsselwort `void` gekennzeichnet wird.

Neben der Vererbung bilden Assoziationen eine weitere Form, Beziehungen zwischen Typen auszudrücken. Während es sich bei der Vererbung um eine Weitergabe und Erweiterung von Eigenschaften handelt, stellt die Assoziation eine Delegationsbeziehung zwischen entsprechenden Instanzen der Typen dar, bei der deren Eigenschaften über Methoden- oder Attributzugriffe genutzt werden. In der TripleLogo-Anwendung wurden insgesamt drei Assoziationen definiert, die in der textuellen Notation durch das Schlüsselwort `association` eingeleitet werden (vgl. Quellcode 3.4, Zeile 53-55). Da diese genauso wie Methoden und Attribute vererbt werden, wird in diesem Fall festgelegt, dass jedes Tier seine Orientierung und Position im Raum kennt. In diesen beiden Fällen ist die Navigierbarkeit durch die Pfeildarstellung eingeschränkt, so dass andersherum z.B. über eine konkrete Position nicht abgefragt werden kann, welchem Tier diese zugeordnet ist. Die Aufgabe der Klasse `Controller` ist es hingegen, die Anzeigefunktionalitäten und Bewegungsstrategien zu kapseln. Der Kontroller gibt den Tieren somit Bewegungsrichtungen vor und muss andererseits von diesen über Positionsänderungen informiert werden, weshalb zwischen diesen beiden Klassen eine bidirektionale Assoziation besteht. Fehlt eine Angabe der Navigationsrichtung, ist diese unterspezifiziert und es wird vorerst von einer bidirektionalen Assoziation ausgegangen. Darüber hinaus kann für jede Seite über die Angabe einer in eckigen Klammern eingefassten Kardinalität die erlaubte Anzahl der assoziierten Objekte angegeben

```
grammar CD extends mc.umlp.common.Common {

  abstract CDElement;
  abstract CDType extends CDElement;
  abstract CDCallable;

  external Value, Body;

  CDDefinition =
      Completeness? Stereotype? "classdiagram" Name
      "{" ( CDClass | CDInterface | CDEnum | CDAssociation | Invariant )* "}";

  CDClass astextends CDType =
      Completeness? Modifier? "class" Name TypeParameters?
      ( "extends" superclasses:ReferenceType ("," superclasses:ReferenceType)* )?
      ( "implements" interfaces:ReferenceType ("," interfaces:ReferenceType)* )?
      ( ";" | "{" (CDAttribute | CDConstructor | CDMethod)* "}" );

  CDInterface astextends CDType =
      Completeness? Modifier? "interface" Name TypeParameters?
      ( "extends" interfaces:ReferenceType ("," interfaces:ReferenceType)* )?
      ( ";" | "{" (CDAttribute | CDMethod)* "}" );

  CDEnum astextends CDType =
      Completeness? Modifier? "enum" Name
      ( "implements" interfaces:ReferenceType ("," interfaces:ReferenceType)* )?
      ( ";" | "{" ( CDEnumConstant ("," CDEnumConstant)* ";"
                    (CDAttribute | CDConstructor | CDMethod)* )? "}" );

  CDEnumConstant  = Name ( "(" CDEnumParameter ("," CDEnumParameter)* ")" )?;
  CDEnumParameter = Value;

  CDMethod astextends CDCallable =
      Modifier? TypeParameters? ReturnType Name
      "(" ( CDParameter ("," CDParameter)* )? ")"
      ( "throws" exceptions:QualifiedName ("," exceptions:QualifiedName)* )?
      ( ";" | Body );

  CDConstructor astextends CDCallable =
      Modifier? TypeParameters? Name
      "(" ( CDParameter ("," CDParameter)* )? ")"
      ( "throws" exceptions:QualifiedName ("," exceptions:QualifiedName)* )?
      ( ";" | Body );

  CDParameter = Type Name;

  CDAttribute = Modifier? Type Name ("=" Value)? ";";

  CDQualifier = Name | Type;

  CDAssociation astextends CDElement =
      Stereotype? ( ["association"] | ["aggregation"] | ["composition"] )
      Derived:["/"]? Name?
      leftModifier:Modifier? leftCardinality:Cardinality?
      leftReferenceName:QualifiedName
      ( "[" leftQualifier:CDQualifier "]")? ("(" leftRole:Name ")" )?
      ( leftToRight:["->"]|rightToLeft:["<-"]|bidirectional:["<->"]|simple:["--"] )
      ( "(" rightRole:Name ")")? ("[" rightQualifier:CDQualifier "]")?
      rightReferenceName:QualifiedName
      rightCardinality:Cardinality? rightModifier:Modifier? ";";
}
```

MC

Quellcode 3.6: Grammatik für Klassendiagramme (komprimierte Darstellung)

werden, wobei "`*`" für beliebig viele steht. Im vorliegenden Beispiel verwaltet ein Kontroller also maximal drei Tiere. Fehlt eine solche Angabe, entspricht dies einer Kardinalität von "`1`". Bei Enumerationen, die im Gegensatz zu Klassen nicht instanziierbar sind, wird ein konkreter Wert der Enumeration assoziiert, bzw. mehrere Werte bei mengenwertigen Assoziationen. Interfaces stehen hingegen für Instanzen entsprechender Subklassen.

Die vorliegende Fassung der UML/P erlaubt drei spezielle Formen von Assoziationen: Aggregation, Komposition und qualifizierte Assoziation. Die ersten beiden Varianten werden in der textuellen Fassung durch die Schlüsselwörter `aggregation` bzw. `composition` anstelle von `association` definiert und drücken aus, dass sich der Typ auf der linken Seite[2] aus den Typen der rechten Seite zusammensetzt. Die Stärke der Bindung zwischen dem Ganzen und seinen Teilen wird im OMG-Standard für die Aggregation nicht näher festgelegt [OMG10d], was zu unterschiedlichen Auslegungen dieser Beziehung in der Literatur führt. Unter anderem ist es laut [RQZ07] erlaubt, dass Teile gleichzeitig mehreren Ganzen angehören können. Damit unterscheidet sich eine Aggregation aber im Grunde kaum von einer gewöhnlichen Assoziation. Für die UML/P wird deshalb die Teile-Ganzes-Beziehung einer Aggregation in sofern präzisiert, dass ein aggregiertes Objekt nur Teil eines Ganzen ist, dieses aber im Laufe seines Lebenszyklus wechseln kann (siehe dazu auch Abschnitt 3.9). Bei der Komposition besteht darüber hinaus eine Existenzabhängigkeit zwischen den beteiligten Objekten, so dass diese einmal als Einheit erstellt, auch nur in dieser Form existieren können. Eine qualifizierte Assoziation spezifiziert hingegen, dass jedes assoziierte Objekt über einen eindeutigen Qualifikator selektiert werden kann. Wie eine Abwandlung der TripleLogo-Architektur in Abbildung 3.7 und Quellcode 3.8 zeigt, handelt es sich dabei häufig um ein Attribut des assoziierten Typs, dessen Name innerhalb der Assoziation in eckigen Klammern hinter dem Namen des assoziierenden Typs angegeben wird. Anstelle eines Attributnamens kann auch ein beliebiger Typ als Qualifikator verwendet werden. Darüber hinaus ist es bei bidirektionalen Assoziationen erlaubt, auf jeder Seite einen Qualifikator anzugeben.

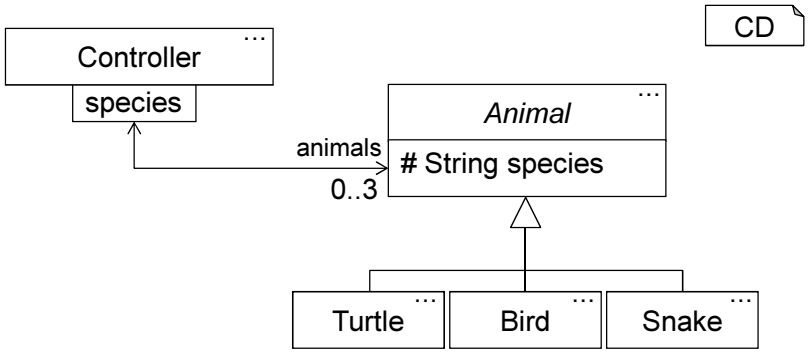


Abbildung 3.7: Beispiel für eine qualifizierte Assoziation (graphisch)

[2] in der graphischen Notation ist dies das mit einer Raute markierte Ende

```
1  package example;
2
3  classdiagram TripleLogoVariant {
4
5    (...) class Controller;
6    (...) abstract class Animal {
7      protected readonly String species;
8    }
9    (...) class Turtle extends Animal;
10   (...) class Bird extends Animal;
11   (...) class Snake extends Animal;
12
13   association Controller [species] <-> (animals) Animal [0..3];
14
15 }
```

Quellcode 3.8: Beispiel für eine qualifizierte Assoziation (textuell)

Neben den bisher beschriebenen Angaben kann für Assoziationen ein Name sowie für jede Seite jeweils ein Rollenname in runden Klammern spezifiziert werden (siehe Quellcode 3.8, Zeile 13). Die Reihenfolge dieser Angaben orientiert sich an der graphischen Darstellung und lässt sich aus den Grammatiken in Quellcode 3.6 bzw. Anhang C.4 ablesen. Diese Namen spielen bei der Navigation entlang der Assoziationen eine Rolle. In der UML/P werden Assoziationen wie Attribute der beteiligten Typen behandelt, wobei der Rollenname dem Attributnamen entspricht. Ist kein Rollenname angegeben, wird stattdessen der Assoziationsname verwendet und fehlt auch dieser, dient der kleingeschriebene Typ als Rollenname. Der interpretierte Attributtyp hingegen ist abhängig von der Kardinalität. Ist diese `<=1`, kann der Typ direkt verwendet werden. In allen anderen Fällen wird von einem Container für den entsprechenden Typ ausgegangen[3], wobei eine qualifizierte Assoziation als Map, die den Qualifikator auf das entsprechende Objekt abbildet, und alle anderen Assoziationen als Menge oder Liste behandelt werden. Tabelle 3.9 führt eine Reihe von Methoden auf, die in der UML/P für diese beiden Containerarten zur Verfügung stehen. Dabei wird wie in Java und der OCL/P der englische Begriff “Collection” als gemeinsamer Supertyp von Mengen und Listen verwendet. Entsprechend Quellcode 3.8 kann für einen Kontroller `c` über `c.animals.size()` die Anzahl der assoziierten Tiere ermittelt werden. Der Zugriff auf ein bestimmtes Tier wie die Schildkröte erfolgt hingegen bei dieser qualifizierten Assoziation über das Attribut `species` mit dem Ausdruck `c.animals.get("Turtle")`.

Für jedes Element eines Klassendiagramms können verschiedene Eigenschaften wie Sichtbarkeit oder Änderbarkeit über davor angegebene Modifikatoren gesteuert werden, wobei es für einige eine Lang- und Kurzbezeichnung gibt, die alternativ verwendet werden kann (siehe Tabelle 3.10). Hervorzuheben ist hier der Modifikator `local`, der im Rahmen dieser Arbeit eingeführt wurde. Dieser schränkt die Sichtbarkeit auf das Modell ein, so dass die darüber gekennzeichneten Elemente außerhalb des Modells mit privaten Elementen gleichzusetzen sind und nicht referenziert

[3]In Java entspricht dies einer parametrisierten `Collection`

Methoden für qualifizierte Assoziationen	*sonstige Assoziationen*	*Bedeutung*
`boolean put(K key, V obj)`	`boolean add(V obj)`	Fügt ein Objekt hinzu
`Collection<V> getValues()`	`Iterator iterator()`	Liefert alle Objekte
`Collection<K> getKeys()`	-	Liefert alle Schlüssel
`Object get(K key)`	-	Liefert das zum Schlüssel passende Objekt
`boolean contains(V obj)`	`boolean contains(V obj)`	Prüft, ob das Objekt enthalten ist
`boolean containsKey(K key)`	-	Prüft, ob der Schlüssel gesetzt ist
`boolean isEmpty()`	`boolean isEmpty()`	Prüft, ob der Container Elemente enthält
`int size()`	`int size()`	Berechnet die Objektanzahl
`boolean remove(K key)`	`boolean remove(V obj)`	Löscht ein Objekt
`void clear()`	`void clear()`	Löscht alle Objekte

Tabelle 3.9: Methoden für den Zugriff auf assoziierte Objektmengen (K steht für den Typ des Schlüssels/Qualifikators, V für den assoziierten Objekttyp)

oder genutzt werden können. Dies schließt die Vererbung von Attributen und Methoden mit ein. Eine lokale Methode kann demnach nur innerhalb des Modells an Subklassen vererbt und überschrieben werden, ist aber außerhalb des Modells nicht aufrufbar. Darüber hinaus werden Attribute und Methoden eines lokalen Typs ebenfalls als lokal angesehen.

Während das Überschreiben von Eigenschaften durch das abermalige Aufführen von Attributen und Methoden in der ursprünglichen UML nur angedeutet werden kann, bietet die UML/P detailliertere Spezifikationsmöglichkeiten durch die Besonderheit der Einbettung von Java-Code an verschiedenen Stellen innerhalb der einzelnen Diagramme. In Klassendiagrammen wird dies genutzt, um Methodenrümpfe oder Werte von Attributen zu spezifizieren. Ein Beispiel hierfür ist das Attribut `species` der Klassen `Turtle`, `Bird` und `Snake` in Quellcode 3.4. Darüber hinaus können Java-Typen direkt innerhalb des Klassendiagramms verwendet werden, etwa als Supertypen oder Typen innerhalb von Attribut- und Methodenspezifikationen. Neben der Angabe des voll-qualifizierten Typnamens, können in solchen Fällen mit Hilfe von `import`-Anweisungen die Namen dieser Java-Typen bekannt gemacht werden, so dass als Typreferenz der einfache Name ausreicht, wie Quellcode 3.4 am Beispiel der Klassen `Color` und `String` zeigt. Äquivalent zu Java ist es ebenfalls erlaubt, alle Typen eines Pakets über die `*`-Notation zu importieren. Auf ähnliche Weise können auch Typen und Assoziationen aus anderen Klassendiagrammen wiederverwendet werden, wobei sich der voll-qualifizierte Name aus den durch Punkte getrennten Paket-, Klassendiagramm- und Typnamen bzw. (Rollen-)Namen der Assoziation zusammensetzt. Der voll-qualifizierte Name

Modifikator	*Beschreibung*	*Anwendung*
`public/+`	Die Sichtbarkeit ist nicht eingeschränkt.	Attribute, Methoden, Assoziationen, Typen
`protected/#`	Die Sichtbarkeit gilt nur innerhalb des Typs oder dessen Subtypen, sowie für Instanzen innerhalb desselben Pakets.	Attribute, Methoden, Assoziationen
`private/-`	Die so gekennzeichneten Elemente werden nicht vererbt und sind nur innerhalb des Typs selbst sichtbar.	Attribute, Methoden, Assoziationen
`readonly/?`	Die Werte von Attributen mit diesem Modifikator sind nur innerhalb des definierenden Typs und dessen Subtypen modifizierbar. Die Lesbarkeit ist hingegen unbeschränkt.	Attribute, Assoziationen
`local`	Die Sichtbarkeit ist auf das definierende Modell beschränkt.	Attribute, Methoden, Assoziationen, Klassen
`final`	Finale Elemente können nicht verändert oder überschrieben werden. Bei Typen bedeutet dies insbesondere, dass keine Subtypen gebildet werden können.	Attribute, Methoden, Assoziationen, Klassen, Enumerationen
`abstract`	Abstrakte Methoden und Typen können nur über Subtypen verwendet werden. Insbesondere müssen abstrakte Methoden in Subtypen implementiert werden.	Methoden, Typen
`derived//`	Die Werte von abgeleiteten Attributen und Assoziationen ergeben sich aus anderen Eigenschaften der definierenden Typen.	Attribute, Assoziationen
`static`	Statische Elemente können ohne eine Instanz genutzt werden.	Attribute, Methoden, Assoziationen

Tabelle 3.10: Modifikatoren in der UML/P

der Klasse `Animal` aus Quellcode 3.4 lautet demnach `mc.tl.TripleLogo.Animal`. Ein Typimport importiert gleichzeitig die im selben Klassendiagramm angegebenen Assoziationen zu diesem Typen. Um alle Typen und Assoziationen aus dem Klassendiagramm zu importieren, kann die Anweisung `import mc.tl.TripleLogo.*;` verwendet werden.

Insbesondere durch die Einbettung von Java-Statements für Attribute und Java-Codeblöcken für Methodenrümpfe sind komplexe Spezifikationen möglich, womit UML/P-Klassendiagramme Turing-vollständig und damit von der Ausdrucksmächtigkeit mit Programmiersprachen gleichzusetzen sind.

3.2 Objektdiagramme

Die in Abschnitt 3.1 vorgestellten Klassendiagramme ermöglichen die Beschreibung der Architektur eines Softwaresystems, der zur Laufzeit des Systems potentiell unendlich viele Objektstrukturen entsprechen können. Diese genauer zu beschreiben ist Aufgabe der Objektdiagramme, die Objekte, deren Zustände und Beziehungen untereinander darstellen können. Somit modelliert ein Objektdiagramm eine Momentaufnahme des Systems, die je nach Einsatzzweck eine geforderte, verbotene oder beispielhafte Situation darstellt. Quellcode 3.12 (bzw. dessen graphische Entsprechung in Abbildung 3.11) zeigt eine solche Objektstruktur, die auf der TripleLogo-Architektur in Quellcode 3.4 basiert. Die zugehörige MontiCore-Grammatik findet sich im Anhang C.5. Quellcode 3.13 zeigt eine komprimierte Fassung.

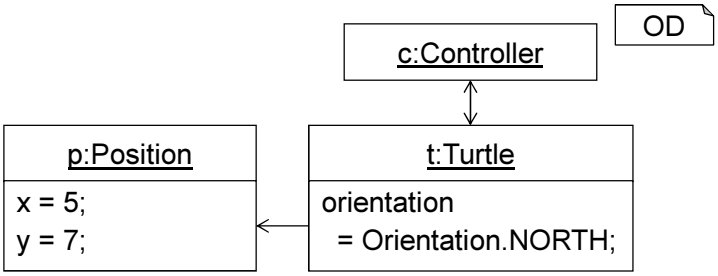


Abbildung 3.11: Beispiel für ein Objektdiagramm (graphisch)

```
1  package mc.tl.test;
2
3  import mc.tl.TripleLogo.*;
4
5  objectdiagram Turtle1 {
6
7    c:Controller;
8
9    t:Turtle {
10     orientation = Orientation.NORTH;
11   }
12
13   p:Position {
14     x = 5;
15     y = 7;
16   }
17
18   link c <-> t;
19   link t -> p;
20
21 }
```

Quellcode 3.12: Beispiel für ein Objektdiagramm (textuell)

Objekte, die die Instanz eines Typs beschreiben, werden über einen eindeutigen Namen, gefolgt von einer durch einen Doppelpunkt angeführten Typangabe dargestellt, wobei eine von beiden

```
1  grammar OD extends mc.umlp.common.Common {
2
3    abstract ODElement;
4
5    external Value;
6
7    ODDefinition =
8        Completeness? Stereotype?
9        "objectdiagram" Name
10       "{" ( ODObject | ODLink | Invariant )* "}";
11
12   ODObject astextends ODElement =
13       Completeness? Modifier?
14       ( Name (":" type:ReferenceType)? | ":" type:ReferenceType )
15       ( ";" | "{" ODAttribute* "}" );
16
17   ODAttribute =
18       Modifier? Type? Name ("=" Value)? ";";
19
20   ODLinkQualifier =
21       Name | Value;
22
23   ODLink astextends ODElement =
24       Stereotype? ( ["link"] | ["aggregation"] | ["composition"] )
25       Derived:["/"]? Name?
26       leftModifier:Modifier?
27       leftReferenceNames:QualifiedName ("," leftReferenceNames:QualifiedName)*
28       ( "[" leftQualifier:ODLinkQualifier "]")? ("(" leftRole:Name ")" )?
29       (
30         leftToRight:["->"]
31         | rightToLeft:["<-"]
32         | bidirectional:["<->"]
33         | simple:["--"]
34       )
35       ( "(" rightRole:Name ")")? ("[" rightQualifier:ODLinkQualifier "]" )?
36       rightReferenceNames:QualifiedName ("," rightReferenceNames:QualifiedName)*
37       rightModifier:Modifier? ";";
38 }
```

MC

Quellcode 3.13: Grammatik für Objektdiagramme (komprimierte Darstellung)

Angaben auch weggelassen werden kann. Dabei werden namenlose Objekte auch als anonyme Objekte bezeichnet. Anstelle des exakten Typs ist alternativ die Angabe eines Supertyps möglich, so dass auch Interfaces oder abstrakte Klassen als Typangabe dienen können. Genauso wie viele Programmiersprachen bieten Objektdiagramme damit eine Abstraktion von den Instanztypen. Im Gegensatz zur graphischen Notation, die sich auf die Nutzung einfacher Typnamen beschränkt, erlaubt die textuelle Darstellung die Angabe von voll-qualifizierten Typnamen sowie entsprechende Import-Anweisungen, so dass die eindeutige Referenzierung von Java-Typen und Typen aus Klassendiagrammen möglich ist (siehe dazu auch Abschnitt 3.9). Quellcode 3.12 verwendet insgesamt die vier Typen `Controller`, `Turtle`, `Position` und `Orientation` aus der TripleLogo-Architektur, die über die Anweisung in Zeile 3 importiert werden.

Der Zustand eines Objekts wird durch dessen Attributwerte ausgedrückt, die in der textuellen Notation in geschweiften Klammern angegeben werden. Diese Angabe entspricht der Notation von Attributen im Klassendiagramm, wobei Typangaben und Modifikatoren optional sind, da

sich diese aus der entsprechenden Deklaration des Objekttyps ablesen lassen. Als ebenfalls optionaler Attributwert ist ein beliebiges Java-Statement erlaubt. Neben den direkten Attributen des Objekttyps können auch geerbte und abgeleitete sowie Klassenattribute aufgeführt werden.

Eine Beziehung zwischen Objekten wird über Links dargestellt (siehe Quellcode 3.12, Zeile 18 und 19), die als Instanz einer Assoziation im Klassendiagramm zu verstehen ist. Im Gegensatz zu den Assoziationen, die Beziehungen zwischen zwei Typen darstellen, erlauben Links jedoch auf jeder Seite eine Komma-separierte Liste von Objekten eines Typs, um auf diese Weise mehrere Links durch eine Zeile auszudrücken. Jedes Objekt der linken ist in einem solchen Fall mit allen Objekten der rechten Seite verlinkt und umgekehrt. Die Referenzierung der Objekte geschieht entweder über den Objektnamen oder -typ, solange sich letzterer eindeutig einem Objekt zuordnen lässt. Ähnlich den Attributen können bei einem Link ebenfalls optional alle Angaben wie Rollenname oder Modifikatoren der entsprechenden Assoziation wiederholt werden. Auch die Navigationsrichtung kann konform zur Assoziation oder unterspezifiziert sein. Die zugehörige Assoziation zu einem Link muss sich anhand der angegebenen Rollen- oder Assoziationsnamen eindeutig bestimmen lassen. Fehlen diese, wie in den Links des obigen Beispiels, muss diese Zuordnung anhand der Objekttypen möglich sein. So ist im Klassendiagramm in Quellcode 3.4 jeweils nur eine Assoziation spezifiziert, die den beiden Links des Beispiels zugeordnet werden kann. Da in beiden Fällen keine Assoziation für den Typ `Turtle` existiert, Assoziationen aber genauso wie Attribute vererbt werden, passen hier nur die Assoziationen der Superklasse `Animal`, also zwischen `Controller` und `Animal` sowie `Animal` und `Position` in den Zeilen 52 und 54 in Quellcode 3.4.

Aggregations- und Kompositionslinks können genauso wie im Klassendiagramm durch die Schlüsselwörter `aggregation` bzw. `composition` eingeleitet werden. Bei Links einer qualifizierten Assoziation kann der Wert des Qualifikators direkt angegeben werden oder indirekt, indem der Wert eines Attributs des verlinkten Objekts über den Attributnamen referenziert wird. In beiden Fällen muss dieser Wert innerhalb des Objektdiagrams für alle Links derselben qualifizierten Assoziation eindeutig sein.

Wie das obige Beispiel zeigt, müssen Objektstrukturen nicht vollständig beschrieben werden, sondern der Modellierer wählt eine geeignete Abstraktionsstufe, die dem Einsatzziel des Objektdiagramms genügt. Deshalb wird in [Rum04a] zwischen Objekten im System und Objekten im Modell klar unterschieden. Letztere werden als "prototypische Objekte" bezeichnet. Diese Unterscheidung ist notwendig, da die in Objektdiagrammen beschriebenen Strukturen nicht notwendigerweise im System auftreten müssen, oder andererseits auch mehrmals auf unterschiedliche Teilstrukturen im laufenden System zur gleichen Zeit angewendet werden können, wobei auch Überlappungen erlaubt sind. So ist in Quellcode 3.12 etwa keine Aussage darüber getroffen, welche Werte die Attribute `color` oder `visible` des Objekts `t` haben. Eine konkrete Instanz der Klasse `Turtle` entspricht also diesem prototypischen Objekt, solange es nach Norden gerichtet ist, unabhängig von dessen aktueller Farbe oder Sichtbarkeit. Ein Objektdiagramm stellt daher einen auf die wesentlichen Aspekte reduzierten Systemausschnitt mit einer zeitlich limitierten Gültigkeit dar. Diese Momentaufnahme eignet sich insbesondere, um bestimmte, festgelegte Situationen

in einem System zu beschreiben, wie z.B. die initiale Objektstruktur beim Systemstart oder bei der Erzeugung von komplexen Aggregations- bzw. Kompositionsstrukturen. Darüber hinaus lassen sich Objektdiagramme auch als Vor- und Nachbedingungen von Tests einsetzen (siehe Abschnitt 3.5). Trotz oder gerade wegen ihres exemplarischen Charakters ist ihr Nutzen nicht auf eine bestimmte Phase in der Softwareentwicklung beschränkt. So können Objektdiagramme der Erhebung von Anforderungen, als Vorstufe oder begleitend zur Architekturmodellierung, zur Dokumentation oder zur Darstellung fehlerhafter Situationen in der Wartung dienen.

3.3 Statecharts

Das Verhalten eines Objekts wird durch dessen Zustand und Methoden bestimmt. Auch wenn durch Einbettung von Java-Code in Klassendiagrammen diese im Prinzip in der Lage sind, eine vollständige Verhaltensbeschreibung zu liefern (siehe Abschnitt 3.1), ist dies jedoch nur in Ausnahmefällen zu empfehlen, da sonst leicht die Übersichtlichkeit der Architekturbeschreibung leidet. Eine weitere Möglichkeit, Verhalten zu beschreiben, sind die in diesem Abschnitt vorgestellten Statecharts.

Der Zustand eines Objekts setzt sich aus zwei Faktoren zusammen [BCGR09a, BCGR09b]. Zum einen sind dies die aktuellen Werte der Attribute, wozu auch die assoziierten Objekte zählen, die den *Datenzustand* darstellen. Darüber hinaus können Objekte Nachrichten in Form von Methodenaufrufen empfangen, sodass sich ein Objekt in einem bestimmten Bearbeitungszustand einer erhaltenen Nachricht befinden kann. Dieser sich aus dem Programmzähler ergebende zweite Faktor wird als *Kontrollzustand* bezeichnet. Das Verhalten eines Objekts manifestiert sich letztendlich in den Methoden, deren Ausführung und Ergebnis vom aktuellen Zustand des Objekts abhängen. Darauf aufbauend bieten Statecharts die Möglichkeit, dieses zustandsbasierte Verhalten für ein einzelnes Objekt oder eine Methode zu beschreiben. Letztere werden auch als Methodenstatecharts bezeichnet. Während die Objektdiagramme aus Abschnitt 3.2 den Datenzustand von Objekten zu einem bestimmten Zeitpunkt darstellen können, ermöglichen Statecharts somit die Modellierung des gesamten Zustandsraumes eines Objekts sowie dessen Übergänge bzw. das Antwortverhalten einer einzelnen Methode bei deren Aufruf. Dabei abstrahiert das Statechart vom potentiell unendlichen Objektzustandsraum, indem es diesen mit einer endlichen Menge von (Diagramm-)Zuständen modelliert [Rum96, Bro97].

Statecharts wurden erstmalig von David Harel in [Har87] eingeführt. Ihr Aufbau aus Zuständen und deren Übergängen, den Transitionen, basiert auf der Automatentheorie und stellt im Wesentlichen eine Weiterentwicklung der endlichen erkennenden Automaten um hierarchische Zustände sowie Ausgaben sowohl bei Zuständen als auch den Transitionen dar. Vereinfacht ausgedrückt, handelt es sich bei Statecharts also um eine Kombination aus hierarchischen Mealy- und Moore-Automaten. Genauere Grundlagen zur Automatentheorie würden an dieser Stelle allerdings zu weit führen. Eine gute Einführung findet sich in [HMU02, Rum96].

In der TripleLogo-Applikation (siehe Abschnitt 3.1) wird das spezifische Verhalten von Objekten der Klasse `Turtle` aus Tabelle 3.2 über das Statechart in Quellcode 3.15 spezifiziert,

das der graphischen Repräsentation in Abbildung 3.14 entspricht. Der Objekttyp wird in der textuellen Fassung direkt nach dem Namen des Statecharts notiert, wobei der voll-qualifizierte Typname entweder direkt angegeben wird oder, wie in diesem Fall, über einen Import bereits bekannt ist. Das Statechart selbst enthält eine Menge von Zuständen und Transitionen. Die textuelle Notation erlaubt darüber hinaus noch freie Codeblöcke, die etwa zur Initialisierung von innerhalb des Statecharts genutzten Variablen verwendet werden können (siehe Zeile 23-25 in Quellcode 3.15).

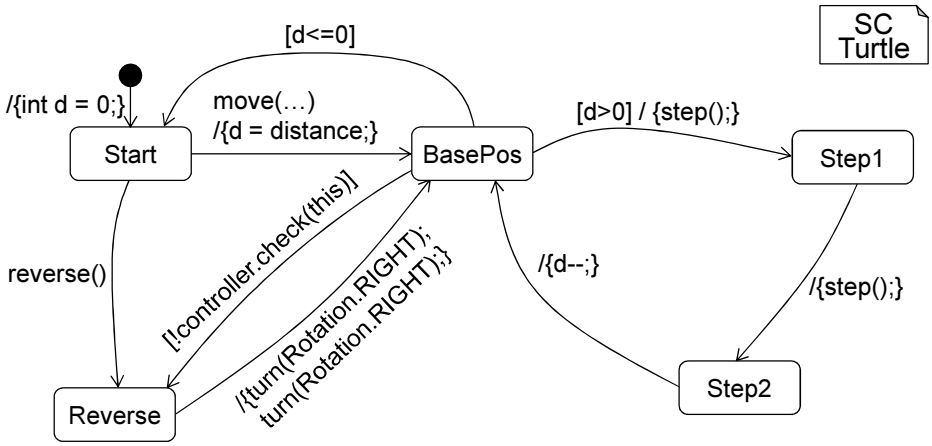


Abbildung 3.14: Statechart für das Verhalten der Klasse `Turtle` (graphisch)

Ein Zustand wird über das Schlüsselwort `state` eingeleitet, gefolgt von einem innerhalb des Statecharts eindeutigen Namen. Die noch vor dem Schlüsselwort angegebenen Modifikatoren `initial` und `final` markieren Start- bzw. Endzustände, also die möglichen Einstiegs- und Endpunkte der Statechart-Ausführung. Dabei kann ein Zustand gleichzeitig initial und final sein. Wie schon an anderen Stellen orientiert sich die UML/P auch bei Statecharts an den Konzepten von Java. So wird davon ausgegangen, dass nicht mehr benötigte Objekte automatisch entfernt werden und damit ein expliziter Endzustand bei der Modellierung von Objektverhalten nicht notwendig ist. Der Startzustand symbolisiert hingegen den Zustand, in dem sich ein Objekt zu Beginn seines Lebenszykluses befindet.

Transitionen referenzieren immer genau einen Quell- und einen Zielzustand innerhalb des Diagramms. Darauf folgt nach einem Doppelpunkt jeweils optional eine Vorbedingung, ein Stimulus, einen Menge von Aktionen, die durch einen führenden Schrängstrich ("/") eingeleitet werden, und schließlich eine Nachbedingung. Letztere kann nur auftreten, wenn auch eine Aktion definiert wurde. Ein Zustandsübergang entlang einer Transition wird bei Auftreten des Stimulus ohne Unterbrechung ausgeführt ("run-to-completion" entsprechend [CCD+02]), wenn sich das Statechart im entsprechenden Quellzustand befindet und die Vorbedingung gilt. Weitere Stimuli werden sequentiell abgearbeitet. Dabei werden Aufrufe innerhalb von Transitionen im betrachteten Statechart nicht als Stimuli interpretiert und führen somit nicht gleichzeitig zur Schaltung weiterer Transitionen desselben Objekts. Es sind auf diese Art demnach keine rekursiven Aufrufe oder

```
1  package mc.tl;
2
3  import mc.tl.TripleLogo.Turtle;
4
5  statechart TurtleBehavior Turtle {
6
7    initial state Start;
8    state Reverse;
9    state BasePos;
10   state Step1;
11   state Step2;
12
13   Start -> BasePos: move(...) / {d = distance;}
14   Start -> Reverse: reverse();
15   BasePos -> Reverse: [!controller.check(this)];
16   Reverse -> BasePos: /{turn(Rotation.RIGHT); turn(Rotation.RIGHT);}
17   BasePos -> Start: [d<=0];
18
19   BasePos -> Step1: [d>0]/{step();}
20   Step1   -> Step2: /{step();}
21   Step2   -> BasePos: /{d--;}
22
23   code {
24     int d = 0;
25   }
26
27 }
```

Quellcode 3.15: Statechart für das Verhalten der Klasse `Turtle` (textuell)

Nebenläufigkeiten vorgesehen. Nach Ausführung einer Transition muss die Nachbedingung erfüllt sein. Andernfalls stellt dies einen Fehler im Modell dar, da das Statechart in einem solchen Fall nicht sinnvoll interpretiert werden kann [Rum96].

Bei einem Stimulus handelt es sich um eine an das im Statechart modellierte Objekt gerichtete Nachricht. In der UML/P werden dabei vier verschiedene Arten von Stimuli unterschieden [Rum04a]:

- Methodenaufruf oder asynchrone Nachricht
- Empfang eines Ergebnisses (aufgrund eines vorherigen Methodenaufrufs)
- Abfangen und Bearbeiten eines aufgetretenen Fehlers (Exception)
- spontane Transition (auch als ϵ-Transition bezeichnet)

Bei der spontanen Transition handelt es sich um einen Übergang ohne Stimuli. Im Quellcode 3.15 sind dies die Transitionen in den Zeilen 15-21. Hier hängt es nur von der Vorbedingung ab, ob die Transition ausgeführt wird, sobald sich das Statechart im entsprechenden Quellzustand befindet. Teilweise dienen diese Transitionen auch nur einer besseren Strukturierung, wie etwa die Zeilen 19-21, die die Zwischenschritte zum Aufbau der unterschiedlichen Bewegungsmuster der Tiere darstellen (vgl. Abbildung 3.3). Methodenaufrufe und Exceptions können zusätzlich anhand der übergebenen Parameterwerte differenziert werden. Die Notation dieser Stimuli erfolgt dabei

ähnlich einem Methodenaufruf in Java durch die Angabe des Methoden bzw. Exception-Namens gefolgt von den Parameterwerten in runden Klammern. Sollen die Parameter hingegen für die Ausführung einer Transition keine Rolle spielen, können stattdessen drei Punkte angegeben werden (siehe Quellcode 3.15, Zeile 13). Dasselbe gilt für die erwarteten Rückgabewerte von Methoden, die ebenfalls in runden Klammern eingefasst und durch das vorangestellte Schlüsselwort `return` gekennzeichnet sind.

Wie schon in Klassen- und Objektdiagrammen, wird auch in Statecharts an verschiedenen Stellen Code eingebettet. Als die in eckigen Klammern eingefassten Vor- und Nachbedingungen können beliebige OCL/P- oder Java/P-Ausdrücke dienen. Über letztere werden auch konkrete Rückgabe- oder Parameterwerte für Stimuli angegeben. Darüber hinaus finden Java-Blockstatements für die Aktionen und die bereits oben beschriebenen Code-Blöcke Verwendung. Diese Code-Stücke können so formuliert werden, als ob sie sich innerhalb des modellierten Objekttyps befinden. Somit haben sie Zugriff auf dessen Attribute, Methoden und assoziierten Objekte, aber auch auf die im Statechart gegebenenfalls definierten lokalen Variablen sowie auf die Argumente des bearbeiteten Stimulus. So bezeichnet zum Beispiel `this` in Zeile 15 das modellierte Objekt vom Typ `Turtle`, das der Methode `check()` des assoziierten Objekts `controller` übergeben wird (vgl. Quellcode 3.4).

Wie bereits eingangs erwähnt, können Statecharts nicht nur das Verhalten eines Objekts, sondern auch den Kontrollfluss einzelner Methoden modellieren. Dabei unterscheiden sich die beiden Formen in der textuellen Notation auf den ersten Blick durch die Angabe nach dem Statechart-Namen. Bei einem Methodenstatechart wird hier neben dem Objekttyp zusätzlich die modellierte Methode inklusive etwaiger Parameter genannt. Die graphische Notation macht diese Angabe in der Diagramm-Marke und sieht bei Methodenstatecharts den zusätzlichen Stereotyp «`method`» vor, auf den in der textuellen Notation verzichtet werden kann. Die angegebene Methode kann als Stimulus verstanden werden, der die Ausführung des Statecharts anstößt. Da während der Ausführung einer Methode keine weiteren Methoden von außen auf einem Objekt aufgerufen werden können, sind innerhalb eines Methodenstatecharts nur spontane Transitionen oder Transitionen mit erwarteten Rückgabewerten erlaubt.

Abbildung 3.16 und Quellcode 3.17 zeigen ein Methodenstatechart für die Methode `step()` der Klasse `Animal`. Das Verhalten dieser Methode ist für alle Subtypen dieser abstrakten Klasse in Quellcode 3.4 gleich und modelliert einen einzelnen Schritt, aus denen sich die komplexeren Bewegungsmuster der Tiere zusammensetzen. Dies zeigt auch, wie einzelne Statecharts unterschiedlichen Typs zusammen arbeiten können. So modelliert das Statechart in Quellcode 3.15 zwar das Verhalten von Objekten des Typs `Turtle`, lässt dabei aber u.a. die Spezifikation der Methode `step()` offen, indem diese nur in verschiedenen Aktionen, nicht aber als Stimulus genutzt wird. Auf diese Weise ergibt sich zusammen mit dem Methodenstatechart für die Methode `step()` das Gesamtverhalten des Typs `Turtle`.

Wie das obige Beispiel zeigt, wird am Anfang der Methode `step()` zuerst bestimmt, in welcher Orientierung sich das Tier aktuell befindet. Da das Attribut `orientation` nur einen von vier Werten annehmen kann, sind die Vorbedingungen der Transitionen in Zeile 15-18 im

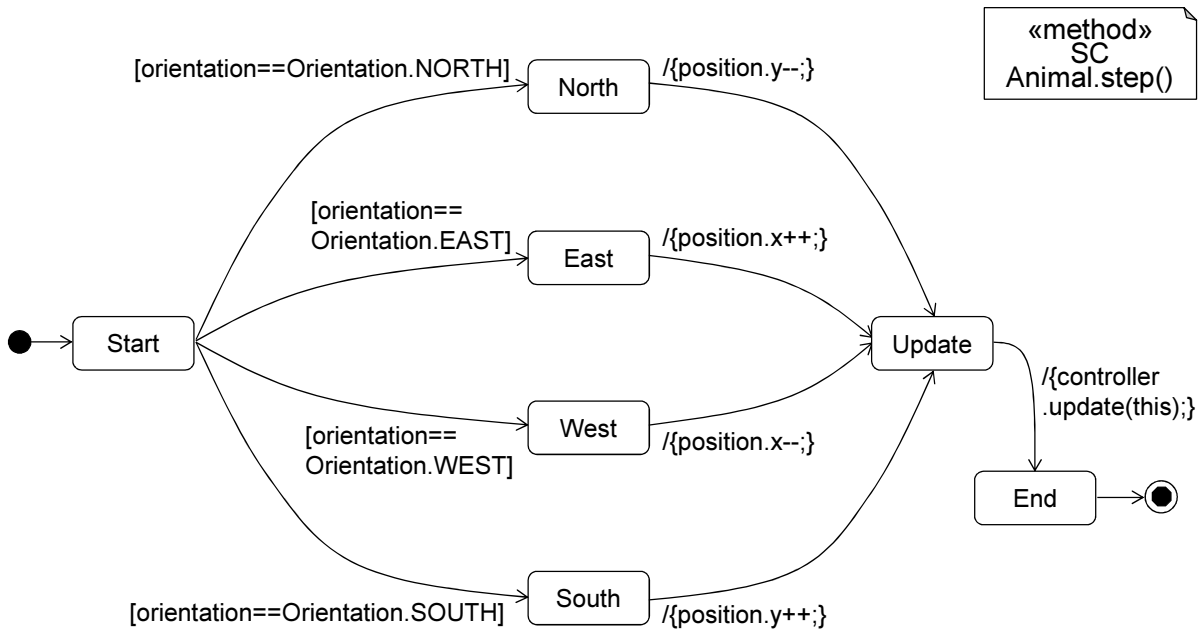


Abbildung 3.16: Mehodenstatechart für die Methode `step()` der Klasse `Animal` (graphisch)

```
1  package mc.tl;
2
3  import mc.tl.TripleLogo.Animal;
4
5  statechart AnimalStepBehavior Animal.step() {
6
7    initial state Start;
8    state North;
9    state East;
10   state West;
11   state South;
12   state Update;
13   final state End;
14
15   Start -> North: [orientation==Orientation.NORTH];
16   Start -> East:  [orientation==Orientation.EAST];
17   Start -> West:  [orientation==Orientation.WEST];
18   Start -> South: [orientation==Orientation.SOUTH];
19
20   North -> Update: /{position.y=position.y--;}
21   East  -> Update: /{position.x=position.x++;}
22   West  -> Update: /{position.x=position.x--;}
23   South -> Update: /{position.y=position.y++;}
24
25   Update -> End: /{controller.update(this);}
26
27 }
```

Quellcode 3.17: Mehodenstatechart für die Methode `step()` der Klasse `Animal` (textuell)

Quellcode 3.17 disjunkt und vollständig. Ausgehend vom initialen Zustand `Start` kann also immer nur genau eine Transition schalten. Gilt dies für alle Zustände des Statecharts, ist dieses deterministisch und vollständig. Umgekehrt spricht man von Nichtdeterminismus, falls mehrere Transitionen schaltbereit sein können[4], oder von Unvollständigkeit, wenn Stimuli in einem Zustand auftreten können, für die keine Transition schaltet. Da sowohl ein Nichtdeterminismus durch die Hinzunahme disjunkter Vorbedingungen, als auch eine Unvollständigkeit durch zusätzliche Transitionen behoben werden kann, handelt es sich in beiden Fällen um eine Unterspezifikation des Systems, die verschiedene Gründe haben kann [Rum04a]:

- Abstraktion aufgrund einer noch frühen Entwicklungsphase oder als vereinfachte Darstellung
- Entscheidung wird dem Compiler/Generator oder System überlassen

Ein Beispiel für einen Nichtdeterminismus findet sich in Quellcode 3.15 bei den ausgehenden Transitionen des Zustands `BasePos` (Zeilen 15, 17 und 19). Danach kann die Transition in Zeile 15 gleichzeitig mit einer der beiden anderen aktiv sein, da das Ergebnis der `check()`-Methode des Controllers unabhängig vom Zähler `d` ist. Dieser Nichtdeterminismus ließe sich durch Hinzunahme von "`&& controller.check(this)`" in der Vorbedingung von Zeile 17 und 19 auflösen. Neben dieser manuellen Auflösung führt [Rum04a] mehrere weitere Verfahren auf, wie solche Unterspezifikationen behoben werden können. So können nicht behandelte Stimuli in einem unvollständigen Statechart dieses in einen expliziten Fehlerzustand oder einen zufälligen Zustand überführen, oder ohne Zustandsänderung eine Fehlermeldung hervorrufen bzw. einfach ignoriert werden. Ein Nichtdeterminismus kann hingegen auch ein gewolltes zufälliges Verhalten des Systems modellieren oder durch einen intelligenten Generator aufgelöst werden, der eine geeignete Transition anhand bestimmter Kriterien oder Konventionen auswählt. In all diesen Fällen wird das entsprechende Verhalten durch den Codegenerator hinzugefügt, wobei die Auswahl des Verfahrens durch eine entsprechende Parametrisierung des Generators erfolgen kann (siehe Kapitel 6).

Bisher wurde nur zwischen einfachen, Start- und Finalzuständen unterschieden. Darüber hinaus erlaubt die UML/P bei Zuständen die Modellierung eines komplexen inneren Aufbaus, der im Folgenden nur kurz beschrieben werden soll. Die genaue Syntax ist der Grammatik in Anhang C.6 oder dessen komprimierte Fassung in Quellcode 3.18 zu entnehmen.

Ähnlich der Schreibweise von Klassen, wird der innere Aufbau eines Zustands in geschweiften Klammern nach dem Zustandsnamen notiert. Da Statecharts in der UML/P hierarchisch sind, können hier innere Zustände und Transitionen angegeben werden. Diese hierarchischen Zustände erlauben eine abstrakte Sicht auf das System, die in den Subzuständen verfeinert wird. Letztere können dabei wiederum hierarchisch sein.

Transitionen können Subzustände einer Hierarchieebene miteinander verbinden, aber diese Zustandshierarchien auch überwinden. Da keine global eindeutigen Zustandsnamen gefordert sind, sich also nur die Namen einer Hierarchieebene innerhalb eines Zustands bzw. eines Diagramms

[4]Dies gilt insbesondere auch dann, wenn mehrere Startzustände vorhanden sind.

MC

```
grammar SC extends mc.umlp.common.Common {

  abstract SCElement;
  abstract SCEvent;

  external Statements, Expression;

  SCDefinition =
      Completeness? Stereotype?
      "statechart" Name ( SCMethod | className:ReferenceType )?
      "{" ( SCState | SCTransition | SCCode )* "}";

  SCMethod =
      name:QualifiedName "(" ( SCParameter ("," SCParameter)* )? ")";

  SCParameter = Type Name;

  SCAction =
      preCondition:Invariant?
      ( ";" | "/" Statements (postCondition:Invariant ";")? );

  SCDoAction    = "do" SCAction;
  SCEntryAction = "entry" SCAction;
  SCExitAction  = "exit" SCAction;

  SCModifier =
      Stereotype? ( ["initial"] | ["final"] | ["local"] )*;

  SCState astextends SCElement =
      Completeness? SCModifier? "state" Name
      (";" | "{" (Invariant ";")?
                  SCEntryAction?
                  SCDoAction?
                  SCExitAction?
                  ( SCState | SCTransition | SCCode | SCInternTransition )* "}"
      );

  SCInternTransition = Stereotype? "->" ":"? SCTransitionBody;

  SCTransition astextends SCElement =
      Stereotype? sourceName:QualifiedName "->" targetName:QualifiedName
      ( ";" | ":" SCTransitionBody );

  SCTransitionBody =
      preCondition:Invariant? SCEvent?
      ( ";" | "/" Statements (postCondition:Invariant ";")? );

  SCMethodOrExceptionCall extends SCEvent =
      name:QualifiedName SCArguments?;

  SCReturnStatement extends SCEvent =
      "return" ( incomplete:["..."] | "(" Expression ")" )?;

  SCArguments =
      incomplete:["(...)"] | "(" ")" | "(" Expression ("," Expression)* ")";

  SCCode astextends SCElement =
      "code" Statements;
}
```

Quellcode 3.18: Grammatik für Statecharts (komprimierte Darstellung)

unterscheiden müssen, erlauben Transitionen die Referenzierung von voll-qualifizierten Namen, die sich bei Zuständen aus den durch Punkten getrennten Namen der Superzustände zusammensetzen (siehe Abbildung 3.19 und Quellcode 3.20). Es wird jedoch empfohlen, möglichst eindeutige Namen zu verwenden, um Missinterpretationen des Statecharts zu vermeiden.

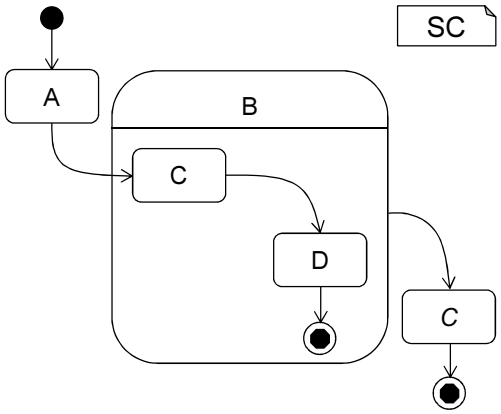


Abbildung 3.19: Beispiel für hierarchische Zustände (graphisch)

```
1  package example;
2
3  statechart HierarchyExample {
4
5    initial state A;
6    state B {
7      state C;
8      final state D;
9
10     C -> D;
11   }
12   final state C;
13
14   A -> B.C;
15   B -> C;
16
17 }
```

Quellcode 3.20: Beispiel für hierarchische Zustände (textuell)

(Die Zustände werden in der Reihenfolge A, B, B.C, D, C besucht.)

Hierarchische Zustände dienen in Automaten oft zur Komposition von Teilautomaten. Dies ist in [Rum04a] allerdings nicht vorgesehen. In der textuellen Notation können jedoch theoretisch die qualifizierten Referenzen von Transitionen dazu verwendet werden, auf andere Statecharts zu verweisen, um auf diese Weise mehrere Diagramme miteinander zu verbinden (siehe Quellcode 3.21 und 3.22). Die Syntax verbietet dies nicht. Auch wenn im Folgenden nur die bereits vorgestellte Komposition von Objekt- und Methodenstatecharts anhand der Stimuli berücksichtigt wird, stellt die Komposition über Statechart-übergreifende Referenzen dennoch eine mögliche zukünftige Erweiterung der Statechart-Semantik dar.

Neben der Darstellung von Hierarchie können Zustände interne Transitionen, Zustandsinvarianten, entry- und exit-Aktionen sowie do-Aktivitäten enthalten. Zustandsinvarianten stellen Bedingungen an den Datenzustand des modellierten Objekts, die innerhalb des Zustands erfüllt sein müssen. Sie präzisieren somit die Beziehung zwischen Diagramm- und Objektzuständen und werden wie Vor- und Nachbedingungen von Transitionen als in eckigen Klammern eingefasste Java/P- oder OCL/P-Ausdrücke im Kontext des Objekts formuliert.

Do-Aktivitäten, entry-/exit-Aktionen und interne Transitionen sind ähnlich aufgebaut wie gewöhnliche Transitionen, können also Vor- und Nachbedingungen sowie auszuführende Blöcke von Java-Statements enthalten, referenzieren aber keinen Quell- oder Zielzustand, da sie nicht zu einem Zustandsübergang führen. Sie werden deshalb nur durch die Schlüsselwörter `do`, `entry` und `exit`, bzw. interne Transitionen durch einen einfachen Pfeil, eingeleitet. Der wesentliche

```
1  package example;
2
3  statechart Outer {
4
5    initial state A;
6    final state C;
7
8    A -> example.Inner;
9    example.Inner -> C;
10
11 }
```

Quellcode 3.21: Eine mögliche Form der Statechartkomposition (vgl. Quellcode 3.20)

```
1  package example;
2
3  statechart Inner {
4
5    initial state C;
6    final state D;
7
8    C -> D;
9
10 }
```

Quellcode 3.22: Referenziertes Statechart der Komposition

Unterschied zwischen diesen Konstrukten liegt im Zeitpunkt und Häufigkeit der Ausführung. Während entry- und exit-Aktionen beim Erreichen bzw. Verlassen eines Zustands einmalig ausgeführt werden, erfolgt der Aufruf von do-Aktivitäten iteriert, solange sich das Statechart im entsprechenden Zustand befindet. Die Häufigkeit der Aufrufe von do-Aktivitäten ist nicht näher spezifiziert und kann im Zielsystem etwa in festen Zeitintervallen oder parallelen Threads erfolgen. Interne Transitionen werden als Transitionen ohne Zustandsänderung verstanden, so dass deren Ausführung von der Vorbedingung und einem Stimulus abhängt. Dabei werden insbesondere keine entry- oder exit-Aktionen ausgeführt, da der Zustand nicht verlassen wird. Insgesamt stellen diese Konstrukte keine semantische Erweiterung dar, sondern dienen einer kompakten Darstellung. Eine entry-Aktion kann alternativ auf alle eingehenden Transitionen des zugehörigen Zustands bzw. eine exit-Aktion auf alle ausgehenden Transitionen verschoben werden, ohne die Bedeutung zu verändern. Ähnliche Semantik-erhaltenden Transformationen (Refactorings) existieren auch für do-Aktivitäten und interne Transitionen (siehe [Rum04a]).

Durch die aufgeführten Möglichkeiten der Unterspezifikation als auch der detaillierten Modellierung von Verhalten können Statecharts der UML/P von der Spezifikation abstrakter Anforderungen bis hin zu ausführbaren Implementierungsbeschreibungen eingesetzt werden. Auch Testfälle lassen sich daraus ableiten. Allerdings beschreibt ein Statechart nur das Verhalten genau eines Objekts als Reaktion auf erhaltene Stimuli. Zur Darstellung von Interaktionen zwischen Objekten, die insbesondere für die Spezifikation von Testfällen von Interesse sind, eignen sich hingegen die im Abschnitt 3.4 vorgestellten Sequenzdiagramme.

3.4 Sequenzdiagramme

In der UML/P werden Sequenzdiagramme im Wesentlichen zur Modellierung von exemplarischen Abläufen in einem Softwaresystem eingesetzt. Im Gegensatz zu Statecharts, die das Verhalten eines Objekts oder einer Methode spezifizieren, stellen Sequenzdiagramme die Interaktionen zwischen Objekten dar. Da in der UML/P im Unterschied zum UML-Standard [OMG10d] in

Sequenzdiagrammen keine Bedingungen, Schleifen oder Nebenläufigkeiten dargestellt werden können, ist deren Mächtigkeit stark eingeschränkt, so dass sie sich kaum zur Modellierung von ausführbaren Systemverhalten einsetzen lassen. Auf diese Weise werden die Sequenzdiagramme der UML/P vereinfacht und eine zu den Statecharts redundante Möglichkeit zur Beschreibung von Abläufen wie im UML-Standard vermieden. Stattdessen werden Sequenzdiagramme in der UML/P nur für die Modellierung von Testfällen und für die Anforderungsdefinition eingesetzt.

Abbildung 3.23 und Quellcode 3.24 zeigen am Beispiel einer Kommunikation zwischen zwei Instanzen der Klassen `Turtle` und `Controller` die beiden Notationsformen von Sequenzdiagrammen. Danach bestehen diese im Wesentlichen aus Objekten und deren Interaktionen. Die Notation der Objekte erfolgt äquivalent zu den Objektdiagrammen in Abschnitt 3.2, verzichtet allerdings auf die Angabe von Attributen. Bei Interaktionen handelt es sich ähnlich wie bei den Stimuli in Statecharts um konkrete Methodenaufrufe, Fehlerausgaben (Exceptions) oder Rückgabewerte, wobei auch hier für Parameter- und Rückgabewerte von Methoden komplexe Java-Ausdrücke oder - als Abstraktion von den exakten Werten - drei Punkte angegeben werden können. Darüber hinaus kann die Angabe von Werten auch ganz ausbleiben. Da die Menge der Interaktionen und insbesondere die der Objekte unvollständig sein kann, muss das sendende Objekt eines Methodenaufrufs bzw. das empfangende Objekt eines entsprechenden Resultats nicht immer angegeben sein (siehe Zeile 13 und 14 in Quellcode 3.24). Ansonsten werden die sendenden und empfangenden Objekte in der textuellen Notation anhand ihres Namens oder alternativ über den Typnamen referenziert, falls dieser eindeutig ist.

Um zwischen Methodenaufrufen und den darauf folgenden Reaktionen in Form von Rückgabewerten und Exceptions leichter unterscheiden zu können, werden in der UML/P verschiedene Pfeile für Interaktionen verwendet. So werden Aufrufe in der graphischen Notation als durchgehende und deren Ergebnisse als gestrichelte Pfeile abgebildet. Da es in der Textform nicht darauf ankommt, auf welcher Seite ein Objekt steht, wurde die kompaktere und intuitivere Darstellung über die Pfeilrichtung gewählt (`->` für Methodenaufrufe bzw. `<-` für Reaktionen). Darüber hinaus werden Rückgabewerte von erfolgreichen Methodenaufrufen durch das Schlüsselwort `return` von fehlerhaften Abbrüchen durch Exceptions abgegrenzt. Insgesamt ergeben sich damit die in Quellcode 3.25 zusammengefassten Möglichkeiten für Interaktionen, wobei es sich bei den Zeilen 8-9, 12-13 und 16-18 um äquivalent Angaben handelt.

Sequenzdiagramme sind die einzige Sprache der UML/P, bei der die (zweidimensionale) Anordnung der Elemente eine Rolle spielt. Dies stellte eine besondere Herausforderung bei der Konzeption einer textuellen Syntax dar, dessen Ergebnis im Folgenden diskutiert wird. So werden Interaktionen graphisch auf einer Zeitlinie aufgetragen, wobei dies nicht maßstabsgetreu zu verstehen ist, sondern nur die zeitliche Reihenfolge abbildet. Diese Reihenfolge der Interaktionen ergibt sich in der textuellen Syntax auf natürliche Weise durch die Leserichtung von oben nach unten, so dass auf eine explizite Darstellung der Zeitlinie verzichtet werden kann.

Die optionalen Aktivitätsbalken entlang der Zeitlinien zeigen die qualitative Dauer der Reaktion auf einen erhaltenen Stimulus an. Dies kann z.B. die Bearbeitung eines Methodenaufrufs sein, so dass sich durch weitere Aufrufe innerhalb dieser Bearbeitung Verschachtelungen ergeben

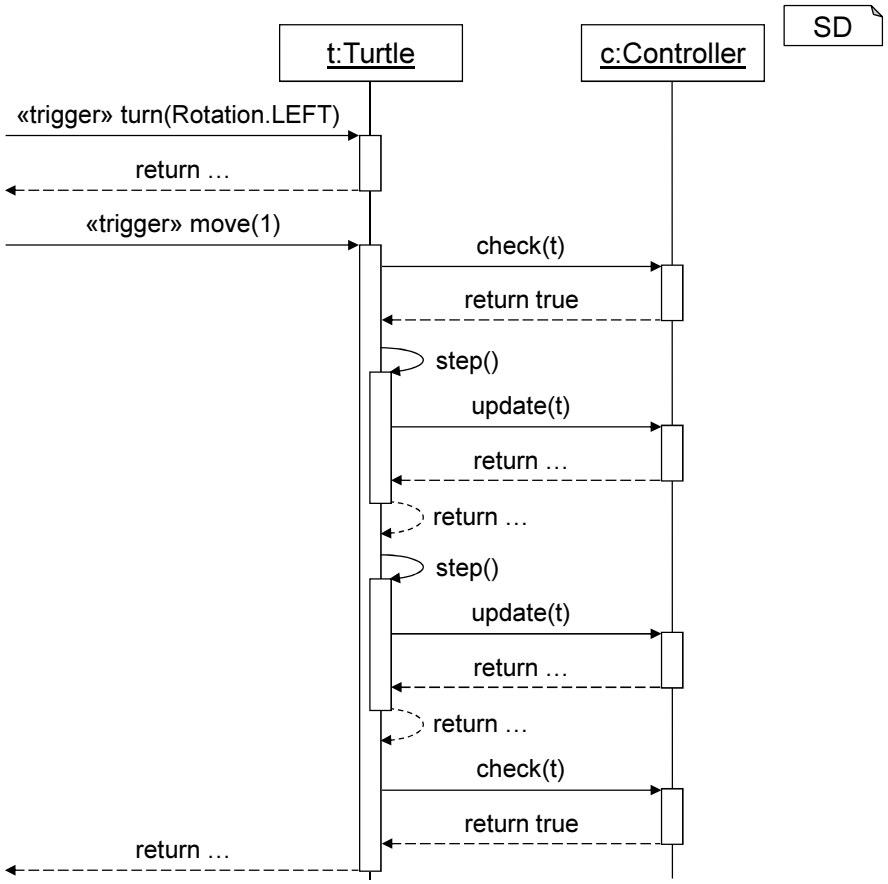


Abbildung 3.23: Sequenzdiagramm einer Kommunikation zwischen den Klassen `Turtle` und `Controller` (graphisch)

können. Auch bei den Aktivitätsbalken handelt es sich nicht um eine zeitlich maßstabsgetreue Abbildung, sondern um eine Gruppierung von Interaktionen, die auf einen gemeinsamen Auslöser basieren. Diese Gruppierung wird in der textuellen Syntax wie in Java üblich durch Einfassen in geschweifte Klammern abgebildet, wobei sich das Zielobjekt der Aktivität aus der ersten Interaktion der Gruppierung ergibt. Da diese Form bei vielen ineinandergeschachtelten Aktivitäten schnell unübersichtlich werden kann, bietet die textuelle Syntax noch eine alternative Schreibweise, indem ein Name für eine Aktivität vergeben und darüber explizit dessen Start und Ende markiert wird. Dieser Name wird entweder auf der rechten Seite der ersten bzw. letzten Interaktion einer Gruppierung direkt nach dem Pfeil angegeben, oder in Form von separaten Statements, die mit den Schlüsselwörtern `start` bzw. `end` eingeleitet werden (siehe Sequenzdiagramm-Grammatik im Anhang C.7, Zeile 91-150 oder dessen komprimierte Fassung in Quellcode 3.28, Zeile 24-36).

Nicht alle Objekte müssen bereits zu Beginn des Sequenzdiagramms definiert worden sein. So kann es vorkommen, dass eine Instanziierung im Rahmen der Interaktionen modelliert werden soll. Wie Abbildung 3.26 zeigt, wird dies in der graphischen Notation durch eine vertikal versetzte Angabe des Objekts dargestellt, so dass die Zeitlinie später beginnt. Die textuelle Syntax

```
1  package mc.tl.test;
2
3  import mc.tl.TripleLogo.Controller;
4  import mc.tl.TripleLogo.Turtle;
5  import mc.tl.TripleLogo.Rotation;
6
7  sequencediagram MoveTurtleTest {
8
9    c:Controller;
10   t:Turtle;
11
12   {
13     -> t : <<trigger>> turn(Rotation.LEFT);
14     <- t : return ...;
15   }
16
17   {
18     -> t : <<trigger>> move(1);
19
20     {
21       t -> c : check(t);
22       t <- c : return true;
23     }
24     {
25       t -> t : step();
26       {
27         t -> c : update(t);
28         t <- c : return ...;
29       }
30       t <- t : return ...;
31     }
32     {
33       t -> t : step();
34       {
35         t -> c : update(t);
36         t <- c : return ...;
37       }
38       t <- t : return ...;
39     }
40     {
41       t -> c : check(t);
42       t <- c : return true;
43     }
44
45     <- t : return ...;
46   }
47
48 }
```

Quellcode 3.24: Sequenzdiagramm einer Kommunikation zwischen den Klassen `Turtle` und `Controller` (textuell)

unterscheidet hierfür bei den aufrufenden Interaktionen zwischen Methoden- und den durch das Schlüsselwort `new` eingeleiteten Instanziierungsaufrufen. Bei letzteren wird der Name des neuen Objekts gleichzeitig über die Zielreferenz auf der rechten Seite der Interaktion eingeführt (siehe Quellcode 3.27, Zeile 13). Auf diese Weise wird eine kompakte Darstellung erreicht, die durch die zeitliche Interpretation entlang der Reihenfolge der Interaktionen mit dem graphischen Modell konsistent ist. Neben der Instanziierung sieht der UML-Standard ebenfalls eine Darstellung für

```
1  sequencediagram Interactions {
2
3    c:Controller;
4    t:Turtle;
5
6    -> c   : run();
7    t -> c : check(t);
8    t -> c : check(...);
9    t -> c : check;
10
11   t -> c : MoveException(t.species);
12   t -> c : MoveException(...);
13   t -> c : MoveException;
14
15   t <- c : return true;
16   t <- c : return ...;
17   t <- c : return;
18   t <- c;
19
20 }
```

Quellcode 3.25: Interaktionen in Sequenzdiagrammen

die Objektzerstörung in Sequenzdiagrammen vor. Durch die Orientierung an den Konzepten von Java wird allerdings in der UML/P davon ausgegangen, dass nicht mehr benötigte Objekte automatisch verworfen werden, so dass eine explizite Darstellung dieses Vorgangs nicht notwendig ist.

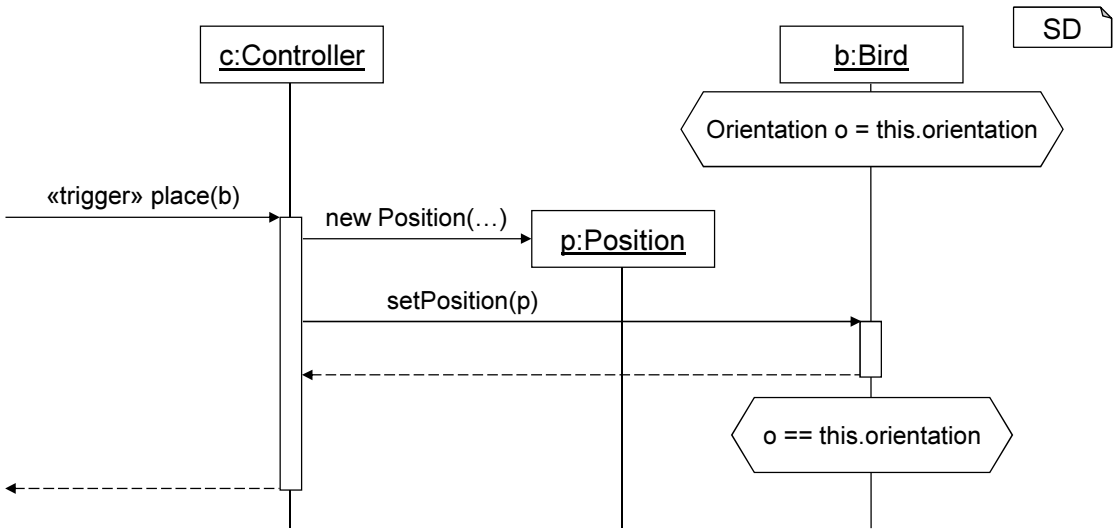


Abbildung 3.26: Sequenzdiagramm mit Objekterzeugung und Bedingungen (graphisch)

Bei der Modellierung von Abläufen werden oft Situationen dargestellt, in denen ein Ereignis zu einer Reihe von Reaktionen führt. Dieses auslösende Ereignis wird in Sequenzdiagrammen mit dem Stereotyp `<<trigger>>` gekennzeichnet (siehe z.B. Quellcode 3.27, Zeile 12). Da Sequenz-

```
1  package mc.tl.test;
2
3  import mc.tl.TripleLogo.*;
4
5  sequencediagram PlaceBirdTest {
6
7    c:Controller;
8    b:Bird;
9
10   {
11     <b: [Orientation o = this.orientation]>;
12     -> c : <<trigger>> place(b);
13     c -> p: new Position(...);
14     {
15       c -> b: setPosition(p);
16       c <- b;
17     }
18     <b: [o == this.orientation]>;
19     <- c;
20   }
21
22 }
```

Quellcode 3.27: Sequenzdiagramm mit Objekterzeugung und Bedingungen (textuell)

diagramme in der UML/P hauptsächlich für die Modellierung von Testfällen eingesetzt werden, handelt es sich bei einem Trigger häufig um die vom Testtreiber aufgerufene Methode, deren resultierende Interaktionen im Rahmen des Testfalls überprüft werden. Der Testtreiber selbst wird in den Modellen im Allgemeinen nicht mit angegeben. Darüber hinaus bieten Sequenzdiagramme die Möglichkeit, neben den erwarteten Ereignissen weitere Bedingungen während des Ablaufs zu überprüfen. Diese in spitzen Klammern eingefassten Java/P- oder OCL/P-Ausdrücke werden entlang der Zeitlinien notiert und haben entsprechend nur eine zeitlich begrenzte Gültigkeit. Der Kontext, in dem die Bedingungen formuliert werden können, bezieht sich immer auf alle zu dem Zeitpunkt bekannten Objekte und Variablen. Dies gilt auch, falls die Bedingung nur eine Zeitlinie überdeckt. Allerdings kann in solch einem Fall auf das Objekt ohne Qualifizierung, oder alternativ über `this` zugegriffen werden. In der textuellen Notation wird diese Überdeckung der Zeitlinien über eine Referenzierung der entsprechenden Objekte direkt zu Beginn einer Bedingung angegeben. Im Beispiel in Quellcode 3.27 wird über die Bedingungen in den Zeilen 11 und 18 sichergestellt, dass sich bei der zufälligen Neuplatzierung eines Tieres über die `place()`-Methode dessen Orientierung nicht verändert, wobei sich in beiden Fällen der `this`-Operator auf das Objekt `b` bezieht.

Durch die Fokussierung auf die Testfallmodellierung und der sich daraus ergebenden Reduktion des Sprachumfangs im Vergleich zum UML-Standard haben Sequenzdiagramme in der UML/P genauso wie Objektdiagramme einen exemplarischen Charakter (siehe Abschnitt 3.2). Die modellierten prototypischen Objekte und deren Interaktionen müssen sich demnach nicht notwendigerweise im System wieder finden. Andererseits können diese auch mehrmals oder verschachtelt auf reale Objekte und Systemabläufe abgebildet werden. Insbesondere ist diese

```
grammar SD extends mc.umlp.common.Common {

  abstract SDElement;
  abstract SDInteraction extends SDElement;
  abstract SDCallMessage;
  abstract SDReturnMessage;

  interface SDActivity;

  external Expression;

  SDDefinition =
      Completeness? Stereotype?
      "sequencediagram" Name
      "{" ( SDActivity | SDObject )* "}";

  SDObject astextends SDElement =
      Completeness? Modifier?
      ( Name (":" type:ReferenceType)? | ":" type:ReferenceType ) ";";

  SDComplexActivity implements SDActivity =
      "{" SDActivity+ "}";

  SDActivityStart implements SDActivity =
      "start" Name referenceName:Name ";";

  SDActivityEnd implements SDActivity =
      "end" Name ("," Name)* ";";

  SDCallInteraction extends SDInteraction implements SDActivity =
      sourceReferenceName:Name? "->" activityStart:Name? targetReferenceName:Name
      ":" Stereotype? SDCallMessage ";";

  SDReturnInteraction extends SDInteraction implements SDActivity =
      targetReferenceName:Name? "<-" activityEnd:Name? sourceReferenceName:Name
      ( ":" Stereotype? SDReturnMessage )? ";";

  SDMethodCall extends SDCallMessage =
      methodName:QualifiedName SDArguments?;

  SDNewStatement extends SDCallMessage =
      "new" type:ReferenceType SDArguments?;

  SDException extends SDReturnMessage =
      type:ReferenceType SDArguments?;

  SDReturnStatement extends SDReturnMessage =
      "return" (Expression | incomplete:["..."])?;

  SDArguments =
      incomplete:["(...)"] | "(" ")" | "(" Expression ("," Expression)* ")";

  SDCondition implements SDActivity =
      "<" range:Name ("," range:Name)* ":" Invariant ">" ";";
}
```

Quellcode 3.28: Grammatik für Sequenzdiagramme (komprimierte Darstellung)

Abbildung auch deshalb nicht immer eindeutig, da bisher nicht festgelegt wurde, wie zusätzliche Interaktionen im realen System zu interpretieren sind, die keiner prototypischen Interaktion im Sequenzdiagramm entsprechen. Sollen diese Mehrdeutigkeiten aufgelöst werden, stehen dem Modellierer in der UML/P spezielle Stereotypen zur Verfügung, die für das gesamte Diagramm oder einzelne Zeitlinien an den Objekten angegeben werden können (siehe auch Abschnitt 3.8). So können die Interaktionen der aufgeführten Objekte vollständig sein (`<<match_complete>>`[5]), beliebig unvollständig (`<<match_free>>`) oder unvollständig nur im Bezug auf nicht dargestellte Objekte (`<<match_visible>>`). Auch `<<match_initial>>` erlaubt weitere Interaktionen, bildet aber immer die erste passende Interaktion im System auf einen entsprechenden Prototyp ab. Eine genauere Diskussion dieser einzelnen Varianten findet sich in [Rum04a].

3.5 Testspezifikationssprache

Für die Qualitätssicherung eines Softwaresystems sind Testfälle unabdingbar [Lig09]. Vor allem die agilen Methoden, deren oberstes Ziel es ist, schon in der frühen Entwicklungsphase ein lauffähiges System bereit zu stellen und trotzdem jederzeit flexibel auf Anforderungsänderungen reagieren zu können, erfordern automatisierte Tests. Auf diese Weise werden Seiteneffekte bei Änderungen schneller erkannt, die Konformität des Systems zu den Anforderungen sicher gestellt sowie das Vertrauen der Entwickler in den Code erhöht. All dies wirkt sich positiv auf die Qualität, Weiterentwickelbarkeit und Flexibilität der Software aus. Hierfür ist es jedoch essentiell, dass Testfälle leicht zu erstellen und verständlich sind, um einerseits eine gute Testfallüberdeckung zu erreichen und andererseits die Kosten für die Entwicklung möglichst gering zu halten. In den letzten Jahren haben sich diesbezüglich insbesondere Testfälle auf Modulebene bewährt, die sogenannten Unit-Tests. Dies geschah nicht zuletzt deshalb, weil Frameworks wie JUnit für Java [Lin05] die Entwicklung von Test- und Produktivcode in derselben Sprache ermöglichen.

Spätestens wenn Modelle wie in der UML/P das zentrale Artefakt der Softwareentwicklung darstellen, bietet es sich an, auch für die Qualitätssicherung diese höhere Abstraktionsebene zu nutzen [Rum03]. Im Allgemeinen besteht ein Testfall aus dem Aufbau einer definierten Testumgebung, der Ausführung der zu prüfenden Systemteile in dieser Umgebung ggf. unter Nutzung weiterer Testdaten, und schließlich der Kontrolle, ob das erwartete Verhalten bzw. Ergebnis eingetreten ist. Diese allgemeine Struktur findet sich in der im Folgenden vorgestellten Testspezifikationssprache wieder, wobei sich für die Modellierung der Testinhalte Objektdiagramme als Vor- und Nachbedingung sowie Sequenzdiagramme zur Darstellung des Testablaufs bestens eignen (siehe Abschnitte 3.2 und 3.4 sowie Abbildung 3.1). Die textuelle Syntax der Sprache kann der MontiCore-Grammatik im Anhang C.8 und dessen komprimierte Fassung in Quellcode 3.29 entnommen werden.

[5] Ursprünglich sieht die UML/P die Schreibweise ≪match:complete≫ vor. Für einen Java-Identifier, aus dem sich entsprechend [Rum04a] ein Stereotyp zusammensetzt, verbietet sich jedoch der Doppelpunkt, so dass dieser durch einen Unterstrich ersetzt wurde.

```
1  grammar TC extends mc.umlp.common.Common {
2
3    external Code;
4
5    TCDefinition =
6        "testcase" Name
7        "{" TCUnitTest* "}";
8
9    TCUnitTest =
10       Name "{"
11         ( "setup" setupReferenceName:QualifiedName ";" )?
12         ( Code | "events" eventsReferenceName:QualifiedName ";" )
13         ( "result" resultReferenceName:QualifiedName ";" )?
14       "}";
15 }
```

MC

Quellcode 3.29: Grammatik für die Testspezifikationssprache (komprimierte Darstellung)

Ein Testfall in der Testspezifikationssprache gruppiert mehrere Unit-Tests und wird durch das Schlüsselwort `testcase` sowie einen Namen eingeleitet. Jeder Unit-Test beginnt ebenfalls mit einem innerhalb des Testfalls eindeutigen Namen, gefolgt von dem eigentlichen Inhalt in geschweiften Klammern. Dieser ist in drei Teile gegliedert:

1. Vorbedingung: Wird durch das Schlüsselwort `setup` eingeleitet und beschreibt die initiale Testumgebung als Objektdiagramm.
2. Testausführung: Diese kann durch eingebettetes Java oder ein Sequenzdiagramm beschrieben werden. Letzteres wird durch das Schlüsselwort `events` gekennzeichnet.
3. Nachbedingung: Nach dem Schlüsselwort `result` beschreibt ein weiteres Objektdiagramm die erwartete Objektstruktur nach Ausführung des Tests.

Die Namen für Testfälle und Unit-Tests dienen im Wesentlichen der Referenzierung während der Testausführung und sollten möglichst so gewählt werden, dass sie einen ersten Eindruck von dem Zweck des Tests vermitteln. Auf diese Weise kann ein fehlgeschlagener Test vom Entwickler identifiziert und der Grund leichter ermittelt werden.

Die Vor- und Nachbedingung sind jeweils optional, so dass Testfälle auch nur durch Sequenzdiagramme bzw. Java beschrieben werden können. Quellcode 3.30 zeigt die beiden sich daraus ergebenden Formen für den Aufbau eines Unit-Tests anhand der in Abschnitt 3.1 vorgestellten TripleLogo-Applikation. Das Objektdiagramm `Turtle1` aus Quellcode 3.12 wird hier wiederverwendet, um den Testaufbau zu beschreiben, indem es über dessen Namen referenziert wird. Da sich das Objektdiagramm im selben Paket wie der Testfall befindet, ist eine Referenzierung über den einfachen Namen des Diagramms ausreichend. Ansonsten könnte alternativ ein vollqualifizierter Name oder ein entsprechender Import angegeben werden. Dasselbe gilt für das erwartete Ergebnis der beiden Tests, dass in Quellcode 3.31 beschrieben ist.

Im ersten Unit-Test `StandardMove` wird nach dem Setup eingebettetes Java verwendet, um die Testausführung zu beschreiben (Quellcode 3.30, Zeile 8-11). Die Java-Statements können

```
1  package mc.tl.test;
2
3  import mc.tl.TripleLogo.Rotation;
4
5  testcase TurtleBehavior {
6    StandardMove {
7      setup Turtle1;
8      {
9        t.turn(Rotation.LEFT);
10       t.move(1);
11     }
12     result Turtle2;
13   }
14
15   StandardMoveWithSD {
16     setup  Turtle1;
17     events MoveTurtleTest;
18     result Turtle2;
19   }
20 }
```

Quellcode 3.30: Testfallspezifikation durch Integration von Objektdiagrammen und Java, sowie als Variante mittels eines Sequenzdiagramms

```
1  package mc.tl.test;
2
3  import mc.tl.TripleLogo.*;
4
5  objectdiagram Turtle2 {
6
7    c:Controller;
8
9    t:Turtle {
10     orientation = Orientation.EAST;
11   }
12
13   p:Position {
14     x = 3;
15     y = 7;
16   }
17
18   link c <-> t;
19   link t -> p;
20
21 }
```

Quellcode 3.31: Modifikation von Objektdiagramm `Turtle1` aus Quellcode 3.12 zur Verwendung als Nachbedingung

dabei über die Namen auf die im Objektdiagramm `Turtle1` beschriebenen Instanzen zugreifen. Da der Java-Code keine weiteren Überprüfungen enthält, schlägt dieser Unit-Test nur fehl, wenn die Objektstruktur nach Abschluss des Tests nicht der in Quellcode 3.31 beschriebenen entspricht. Dazu werden die Objekte aus der Setup-Struktur unter Berücksichtigung von ggf. neu erzeugten sowie geänderten Objekten im Ausführungsteil mit den Objekten der Ergebnisstruktur verglichen.

Dies geschieht wie auch schon im Java-Code unter der Annahme, dass gleich benannte Objekte in den Modellen dasselbe Objekt im System modellieren. Somit wird in diesem Beispiel das prototypische Objekt `t` im Objektdiagramm `Turtle1` erzeugt, durch den Ausführungsteil verändert und die Änderungen anhand der Beschreibung von `t` im Objektdiagramm `Turtle2` überprüft. Entsprechende Vergleiche finden für die anderen Objekte `c` und `p` statt. Für nicht benannte Objekte können hingegen mehrere Möglichkeiten einer Zuordnung bestehen. Insbesondere darf auch ein benanntes Objekt einem unbenannten zugeordnet werden (und umgekehrt), sofern über die Namen keine Zuordnung stattgefunden hat. Lässt sich auf diese Weise ein gültiges Mapping unter Berücksichtigung der angegebenen Attributwerte und Assoziationen der prototypischen Objektstruktur auf das System finden, ist die Nachbedingung gültig. Alternativ kann auch ein Vergleich ohne Beachtung der vergebenen Objektnamen erfolgen. Der Wechsel zwischen diesen Varianten ließe sich über die Angabe eines zusätzlichen Stereotyps wie `<<match_free>>` vor dem Unit-Test oder dem Testfall[6] steuern (siehe Abschnitt 3.8), wird aber in dieser Arbeit nicht weiter berücksichtigt.

Im zweiten Unit-Test `StandardMoveWithSD` wird in Zeile 17 anstelle des Java-Codes das Sequenzdiagramm aus Quellcode 3.24 verwendet. Da die Trigger genau den Aufrufen im Ausführungsteil des ersten Unit-Tests entsprechen, sind die beiden Tests vom Ablauf her identisch. Durch die Verwendung des Sequenzdiagramms wird aber zusätzlich noch die erwartete Kommunikation zwischen den Objekten beschrieben, so dass der Prüfumfang weit über den des ersten Unit-Tests hinausgeht und z.B. auch Fehler in den Aktualisierungs- und Prüfaufrufen des Controllers aufzeigt. Dabei entspricht das Mapping der Objekte dem bereits beschriebenen Verfahren.

Die im Rahmen dieser Arbeit entwickelte Testspezifikationssprache konzentriert sich auf die wesentlichen Aspekte, um Testfälle bestehend aus Objektdiagrammen und einer Ablaufbeschreibung als Sequenzdiagramm oder Java-Code zu spezifizieren. Dadurch ist eine sehr kompakte und einfach verständliche Sprache entstanden, die trotz eines hohen Abstraktionslevels im Prinzip keinen Einarbeitungsaufwand auf Seiten der Modellierer erfordert. [Rum04b] diskutiert eine deutlich komplexere Sprache, deren zusätzlichen Elemente sich bei Bedarf in die hier vorgestellte Testsprache integrieren lassen, oder als alternative Notation angeboten werden kann.

3.6 OCL/P

Die Object Constraint Language (OCL) ist eine reine Spezifikationssprache und wird in der UML eingesetzt, um Bedingungen auf den in Diagrammen modellierten Strukturen zu beschreiben. Im Gegensatz zum OCL-Standard der OMG [OMG10b] handelt es sich bei der OCL/P um eine an die Java-Notation angepasste Variante, die darüber hinaus einige Vereinfachungen, aber auch Erweiterungen gegenüber dem Standard enthält. Eine wesentliche Erweiterung stellt der Umgang mit Mengen und Listen dar, der aus funktionalen Sprachen wie Haskell [Bir98] entlehnt wurde und die OCL insbesondere um eine mathematiknahe Beschreibungsform für Mengen

[6] Je nach Ort der Angabe sind nur einzelne oder alle im Testfall enthaltenen Unit-Tests betroffen.

(Mengenkomprehension) ergänzt. Andererseits lagert die OCL/P einige der im OCL-Standard enthaltenen Operatoren in eine eigene Bibliothek auf Ebene des zugrunde liegenden Modells aus, um die Sprache selbst sowohl schlank und leichter erlernbar als auch erweiterbar zu halten. Eine detaillierte Betrachtung der Unterschiede findet sich in [Rum04a].

Da es sich bei der OCL/P bereits um eine textuelle Spezifikationssprache handelt, wurde die Syntax weitestgehend aus [Rum04a] übernommen und in Zusammenarbeit mit [PS07] und [Pin07] in die Grammatikbeschreibungssprache von MontiCore übertragen (siehe Anhang C.9). Im Folgenden soll deshalb die OCL/P nur kurz eingeführt werden. Für eine ausführliche Beschreibung sei auf [Rum04a] verwiesen.

Wie bereits in den vorherigen Abschnitten 3.1-3.4 angedeutet, können OCL/P-Ausdrücke an verschiedenen Stellen in den UML/P-Diagrammen eingebettet werden. Darüber hinaus ist ebenfalls die Nutzung innerhalb eigener Dateien möglich. Trotzdem werden OCL/P-Ausdrücke immer im Kontext eines Diagramms verwendet, um zusätzliche Aussagen darüber zu spezifizieren. In der UML/P gibt es daher folgende Verwendungsformen:

- *(Typ-) Invarianten* beschreiben Eigenschaften von Typen in Klassendiagrammen, die für die entsprechenden Instanzen während ihrer gesamten Lebenszeit gültig sein müssen. Sie können u.a. dazu eingesetzt werden, um abgeleitete Assoziationen oder Attribute zu spezifizieren.
- *Methodenspezifikationen* definieren in Klassendiagrammen Vor- und Nachbedingungen von Methoden, die diese zum Zeitpunkt der Ausführung erfüllen sollen.
- *Zustandsinvarianten* können für Zustände eines Statecharts angegeben werden. Sie stellen Bedingungen an das beschriebene Objekt, die es innerhalb dieses Zustands erfüllen muss. Zustandsinvarianten spezifizieren somit die Menge von Objektzuständen, die dem Diagrammzustand zugeordnet sind.
- *Vor- und Nachbedingungen von Aktionen* sind neben den Zustandsinvarianten die zweite Anwendungsmöglichkeit von OCL/P-Ausdrücken in Statecharts. Aktionen können dabei als Teil von Transitionen oder als entry- und exit-Aktionen sowie do-Aktiviäten in Zuständen auftreten.
- *Bedingungen innerhalb von Abläufen* in Sequenzdiagrammen, die einmalig zum angegebenen Zeitpunkt gelten müssen.
- *Beschreibungen von komplexen Objektstrukturen* sind durch die Kombination von OCL/P und Objektdiagrammen möglich. So können in Objektdiagrammen abstrakte Werte für Attribute oder Qualifikatoren von Links angegeben und durch OCL/P-Bedingungen genauer spezifiziert werden. Auf diese Weise sind nicht nur konkrete Werte, sondern auch die Angabe von Wertebereichen möglich, die zu einer musterartigen kompakten Beschreibung von Objektstrukturen führen. Zusätzlich können OCL/P-Bedingungen die Anwendbarkeit eines solchen Musters einschränken.

Quellcode 3.32 zeigt eine Invariante und eine Methodenspezifikation, die sich auf die Architektur der TripleLogo-Applikation aus Quellcode 3.4 beziehen. Diese hätten auch innerhalb

des Klassendiagramms selbst angegeben werden können, wurden hier aber in eine eigenständige Datei ausgelagert. Dies ist im Allgemeinen zu empfehlen, da die OCL/P-Ausdrücke sonst leicht zu einer Überladung der Diagramme führen. Die Verbindung zur Architekturbeschreibung erfolgt mit Hilfe von Namensreferenzen, die über die Import-Anweisung in Zeile 3 qualifiziert werden können.

```
1  package mc.tl;
2
3  import mc.tl.TripleLogo.*;
4
5  ocl TripleLogoConstraints {
6
7    context Controller c inv AnimalKinds:
8      forall a1 in c.animal, a2 in c.animal:
9        (a1.species == a2.species) implies (a1 == a2);
10
11   context Animal.reverse()
12     pre: true;
13     post: orientation@pre != orientation
14           || position.x@pre != position.x
15           || position.y@pre != position.y;
16
17 }
```

Quellcode 3.32: OCL/P-Bedingungen der TripleLogo-Architektur in Quellcode 3.4

Eine Invariante spezifiziert zu Beginn die Menge der Typen, den sogenannten Kontext, für deren Instanzen die darauf folgenden Ausdrücke zu jedem Zeitpunkt im laufenden System gelten sollen. Im Falle der Invariante in den Zeilen 7-9 von Quellcode 3.32 ist dies nur die Klasse `Controller`, für die darüber hinaus der optionale Name `c` definiert wurde, die im Folgenden als eine Referenz auf eine beliebige Instanz verstanden werden kann. Da der Kontext in diesem Fall nur eine Klasse umfasst, könnte alternativ auch der aus Java bekannte `this`-Operator verwendet oder auf Attribute, Assoziationen und Methoden der Klasse direkt zugegriffen werden. Der optionale Name `AnimalKinds` kann dazu genutzt werden, um die Invariante zu referenzieren. So wurde in [Rum04a] eine Erweiterung von Java vorgeschlagen, die den expliziten Aufruf solcher Invarianten erlaubt (siehe auch Abschnitt 3.7). Der darauf folgende Inhalt der Invariante demonstriert die mathematiknahe Beschreibungsweise der Spezifikationssprache OCL/P. So wird über den `forall`-Ausdruck spezifiziert, dass sich das Attribut `species` aller assoziierten Tiere des Kontrollers unterscheidet. Dabei ist die OCL/P nicht durch Sichtbarkeiten oder Assoziationsrichtungen eingeschränkt, kann also auch entgegen unidirektionaler Assoziationen navigieren, oder wie im Beispiel, auf mit `private` oder `protected` ausgezeichnete Attribute zugreifen. Zusammen mit den Spezifikationen, die sich aus der Architekturbeschreibung in Quellcode 3.4 ergeben[7], stellt diese Invariante insgesamt sicher, dass mit einem Kontroller maximal nur eine Instanz von jedem Tier assoziiert sein kann.

[7] insbesondere die Kardinalität der Assoziation zwischen `Animal` und `Controller`

Bei der zweiten Bedingung in Zeile 11-15 handelt es sich um eine Methodenspezifikation. Diese trifft im Gegensatz zur Invariante keine Aussagen über den gesamten Lebenszyklus eines Objekts, sondern nur zum Zeitpunkt eines Methodenaufrufs durch Vor- und Nachbedingungen, die jeweils durch die Schlüsselwörter `pre` und `post` eingeleitet werden. Im vorliegenden Fall wird sichergestellt, dass sich nach dem Aufruf der Methode `reverse()` entweder die Orientierung des Tieres oder mindestens eine der Koordinaten dessen Position verändert hat. Dabei kann in der Nachbedingung durch Anfügen des `@pre`-Operators auf Werte von Attributen und Assoziationen zum Zeitpunkt vor dem Methodenaufruf zugegriffen werden. Eine Vorbedingung für den Aufruf der Methode wurde nicht gestellt. Die Zeile 12 dient hier nur der Vollständigkeit und hätte auch weggelassen werden können.

Die Einbettung von OCL/P-Invarianten in Modellen zeigt Quellcode 3.33 am Beispiel eines Objektdiagramms, wobei der Kontext durch das Objektdiagramm vorgegeben und innerhalb der Invarianten durch die `import`-Anweisung präzisiert wird (siehe Zeilen 16 und 22). Die erste Invariante `I1` in den Zeilen 15-19 iteriert über die Menge der möglichen Werte für das Attribut `species`. Dabei führt die Bedingung in Zeile 18 dazu, dass das prototypische Objekt `a` im Modell bei jedem Durchlauf auf ein anderes Objekt im System gemappt wird und somit drei Instanzen vom Typ `Animal` vorhanden sein müssen. Gleichzeitig wird durch die Invariante `I2` in den Zeilen 21-24 gefordert, dass keines der Tiere nach Norden ausgerichtet ist. Die Variable `x` in Zeile 10 wird somit erst durch die Invariante spezifiziert. Auf diese Weise können komplexe Objektstrukturen deutlich kompakter beschrieben werden, als dies allein mit Objektdiagrammen möglich wäre (siehe dazu auch [Rum04a]).

Die Beispiele zeigen nur einen kleinen Ausschnitt der Möglichkeiten der OCL/P. Selbst eine Profilbildung über die Verwendung von Stereotypen ist vorgesehen. Letztendlich handelt es sich jedoch immer um die Spezifikation von Bedingungen im Kontext eines Modells, bei deren Überprüfung eine Änderung des entsprechenden Systems ausgeschlossen werden soll. Mit Ausnahme von lokalen Variablen, die nur im Kontext der Bedingung gültig sind, bietet die OCL/P deshalb keine eigene Definition von Datentypen, Variablen oder Methoden, sondern nutzt die des zugrunde liegenden Modells. Dabei ist allerdings zu beachten, dass nur auf solche Methoden zugegriffen wird, die wiederum keine Zustandsänderung des Systems hervorrufen. Da eine solche Seiteneffektfreiheit einer Methode automatisiert nicht immer nachweisbar ist, sieht [Rum04a] die Kennzeichnung über spezifische Stereotypen vor. So kennzeichnet der Stereotyp `<<query>>` seiteneffektfreie Methoden, deren Verwendung auch gleichzeitig im Produktivsystem möglich ist. Mit `<<OCL>>` werden hingegen reine Spezifikationsqueries markiert, die ausschließlich von der OCL/P genutzt werden können. Über diesen Mechanismus ließe sich u.a. auch eine eigene Bibliothek von allgemein nützlichen und garantiert seiteneffektfreien Methoden für die OCL/P entwickeln.

Trotz oder gerade wegen der oben genannten Einschränkungen, ist die OCL/P als Ergänzung der anderen Notationsformen der UML/P bestens geeignet, um von der Beschreibung von zu realisierenden Eigenschaften bis hin zur Generierung von Tests oder Invarianten im Code eingesetzt zu werden.

```
1  package mc.tl.test;
2
3  import mc.tl.TripleLogo.*;
4
5  objectdiagram NorthFree {
6
7    c:Controller;
8
9    a:Animal {
10     orientation = x;
11   }
12
13   link c <-> a;
14
15   ocl:[
16     import Animal a inv I1:
17       forall String s in {"Turtle", "Bird", "Snake"}:
18         NorthFree && I2 && a.species == s;
19   ]
20
21   ocl:[
22     import Orientation x inv I2:
23       x != Orientation.NORTH;
24   ]
25
26 }
```

Quellcode 3.33: Objektdiagramm mit eingebetteten OCL/P-Invarianten

3.7 Java/P

Nach dem ursprünglichen Konzept von Java/P handelt es sich dabei im Wesentlichen um eine Erweiterung der Programmiersprache Java [GJS05] um die Angabe von OCL/P-Bedingungen. Hierzu wurde in [Rum04a] das zusätzliche Schlüsselwort `ocl` eingeführt, das die Angabe und Referenzierung von OCL/P-Ausdrücken erlaubt und diese vom regulären Java-Code abgrenzt. Dadurch ist es möglich, die Überprüfung von OCL/P-Bedingungen im Systemablauf genauer festzulegen, was vor allem bei Invarianten von Interesse ist. Diese gelten von Seiten der Spezifikation während der gesamten Laufzeit des Systems. Eine entsprechende Validierung wäre jedoch nicht zuletzt aus Effizienzgründen unpraktikabel, so dass eine explizite Definition der Überprüfungszeitpunkte im Code sinnvoll ist. Doch auch ohne eine solche Erweiterung ist die Nutzung und effiziente Umsetzung von OCL/P-Bedingungen im Rahmen der anderen Teilsprachen möglich (siehe Abschnitt 3.6). Insbesondere für die Generierung von Testfällen lassen sich selbst Invarianten durch einen intelligenten Codegenerator sinnvoll umsetzen, indem z.B. Änderungen von Attributen und assoziierten Objekten beobachtet werden und zur Überprüfung von Invarianten mit entsprechenden Objektkontext führen (vgl. Abschnitt 6.4).

Auch wenn sich eine entsprechende OCL/P-Erweiterung der Java-Syntax leicht in das im Rahmen dieser Arbeit entwickelte UML/P-Framework integrieren ließe, wird dies im Folgenden nicht weiter berücksichtigt. Stattdessen steht die Integration von Java und der UML/P im Vordergrund. In der in [Rum04a, Rum04b] beschriebenen ursprünglichen Fassung der UML/P wurde

nicht festgelegt, ob sich die in den Diagrammen eingebetteten Java-Anteile auf den generierten Systemcode oder die Modellelemente beziehen. Aus verschieden Gründen, die im Folgenden diskutiert werden, wurde dies nun konkretisiert und der Java-Code auf die Modellierungsebene angehoben (siehe dazu auch Abschnitt 3.9). Die Integration findet dabei nicht auf syntaktischer Ebene sondern in der Infrastruktur und Umsetzung des unterstützenden Frameworks statt, die in den Kapiteln 6 und 7 näher behandelt werden. Da es sich im Grunde bei der Anhebung von Java um eine konsequente Weiterführung der Kombination von Modellen und Code im Sinne der UML/P handelt, wird diese Java-Form im Folgenden auch als Java/P bezeichnet.

Quellcode 3.34 zeigt ein Beispiel im Rahmen der TripleLogo-Applikation. Syntaktisch unterscheidet sich diese Klasse nicht von einer normalen Java-Klasse. Dennoch handelt es sich um Java/P-Code, da neben Klassen der Java-Bibliothek wie `javax.swing.JFrame` (Zeile 3) auch Typen aus dem Klassendiagramm in Quellcode 3.4 referenziert werden (siehe z.B. Import-Anweisung in Zeile 5). Dabei bezieht sich der Code nur auf die Informationen, die dem Klassendiagramm zu entnehmen sind. Inwieweit dieser Zugriff allerdings mit dem des generierten Produktionscodes übereinstimmt, hängt vom verwendeten Generator ab. Bildet dieser Klassen, Attribute und Methoden des Klassendiagramms auf die entsprechenden Elemente in Java ab, ohne dabei Sichtbarkeiten zu verändern, und werden diese im Paket `mc.tl.TripleLogo` abgelegt, kann der Code in Quellcode 3.34 direkt für das Produktivsystem übernommen werden. Oft werden aber Änderungen bei der Transformation von Diagramm- in den Produktivcode vorgenommen, die einen geänderten Zugriff bewirken. Beispiele sind die Steuerung des Attributzugriffs über get- und set-Methoden oder die Erzeugung von Instanzen mit Hilfe von Factories[8]. Diese Design-Pattern [GHJV95] stellen einerseits eine Verbesserung der Systemarchitektur dar, andererseits handelt es sich bei deren Umsetzung um Routineaufgaben. Damit sind sie bestens dazu geeignet, vom Generator übernommen zu werden, um die Entwickler zu entlasten. In solchen Fällen ist der Generator dafür verantwortlich, den Java/P-Code in entsprechenden nativen Java-Code zu transformieren, der mit dem restlichen generierten System zusammenarbeitet (siehe Abschnitt 5.2). Dies schließt die in den Diagrammen eingebetteten Code ein, da es sich auch hier um Java/P handelt.

Der wichtigste Vorteil dieses Verfahrens ist die klare Trennung von Modellierungs- und (generiertem) Systemcode, was eine Reihe weiterer Vorteile mit sich bring:

- *Konsistente Modellierung*: Durch den Bezug von Java/P auf die Modellelemente entsteht eine in sich konsistente Beschreibung des Systems, da keine Ausdrücke verwendet werden, die sich nur durch Kenntnis des Generators bzw. des generierten Codes, nicht aber durch die Analyse der Modelle ableiten lassen.
- *Unabhängige Weiterentwicklung*: Der Generator wird austausch- und weiterentwickelbar, ohne dass dies Änderungen in bestehenden Modellen nach sich zieht. Dasselbe gilt umgekehrt für die Modelle, so dass generierte Systeme sowohl von weiterentwickelten Modellen, als auch von besseren Generatoren profitieren können. Insbesondere können über neue Generatoren

[8] Auch im Deutschen ist in diesem Fall die Verwendung der englischen Bezeichnung für "Fabrik" üblich.

Java/P

```
1  package mc.tl.swing;
2
3  import javax.swing.JFrame;
4
5  import mc.tl.TripleLogo.*;
6  import mc.tl.view.View;
7
8  public class SwingController extends Controller {
9
10   private int rectHeight = 500; // height and width of the inner rectangle
11   private int rectWidth  = 800;
12
13   private int imgHeight = 50; // height and width of the animal images
14   private int imgWidth  = 50;
15
16   private int margin = 20; // additional space outside of the inner rectangle
17   private int scale  = 30; // animal step length
18
19   private View view;
20
21   ...
22
23   public void update(Animal a) {
24     view.repaint();
25   }
26
27   public boolean check(Animal a) {
28     int x = a.position.x*scale;
29     int y = a.position.y*scale;
30
31     if ((x <= margin+imgWidth && a.orientation == Orientation.WEST) ||
32         (x >= rectWidth+margin-imgWidth && a.orientation == Orientation.EAST)) {
33       return false;
34     }
35     if ((y <= margin+imgHeight && a.orientation == Orientation.NORTH) ||
36         (y >= rectHeight+margin-imgHeight && a.orientation == Orientation.SOUTH)) {
37       return false;
38     }
39     return true;
40   }
41
42 }
```

Quellcode 3.34: Beispiel für eine Java/P-Klasse im Modell-Kontext (Ausschnitt). Diese realisiert den plattformspezifischen Anteil der TripleLogo-Applikation.

bei Bedarf nicht nur die in der Entwicklung befindlichen Systeme, sondern auch auf Modellen basierende Legacy-Systeme verbessert und an neueste Technologien angepasst werden. Die Entkopplung führt außerdem dazu, dass Modelle und Generatoren sich unabhängig voneinander in weiteren Projekten wieder verwenden lassen.

- *Trennung der Zuständigkeiten*: Der Modellierer braucht keine Kenntnis vom Generator und kann den Fokus ausschließlich auf den Entwurf der Modelle legen. Auch die Entwicklung des Generators ist von konkreten Modellen entkoppelt. Damit findet eine Rollentrennung von Modellierer und Generator-Entwickler statt, so dass diese Aufgaben in größeren Projekten

auf unterschiedliche Teams verteilt werden können. Auch eine Auslagerung einer der beiden Bereiche ist so möglich.

- *Zielplattform-unabhängige Modelle*: Da sich die Modelle nicht auf den generierten Code beziehen, besteht prinzipiell die Möglichkeit, diese unabhängig von der Zielplattform zu entwickeln, und die plattformspezifischen Aspekte des Produktivcodes allein im Generator zu kapseln. Dadurch ist es möglich, über verschiedene Generatoren ein System gleichzeitig für unterschiedliche Plattformen aus denselben Modellen zu erzeugen. Dies setzt allerdings voraus, dass die im Kontext der Modelle genutzten Java-Anteile, die eingebetteten Code, Java/P- und insbesondere auch Legacy-Klassen der Java-Bibliothek umfassen, ebenfalls von der Zielplattform unabhängig sind. Insbesondere wenn das Produktivsystem in einer anderen Sprache als Java generiert werden soll, müssten ansonsten diese Anteile ebenfalls übersetzt werden, was einen komplexen Generator erfordert. Je nach Komplexität der Modelle und der Wiederverwendbarkeit eines entsprechenden Generators kann es durchaus effizienter sein, statt des Generators entsprechend angepasste Modelle zu entwickeln. Dazu bietet die vorliegende Fassung der UML/P die Möglichkeit, auch andere Sprachen als Java in Modellen einzusetzen (siehe Abschnitt 3.9).

Neben der im letzten Punkt diskutierten Kapselung der Plattform-abhängigen Anteile im Generator, kann diese auch auf Architekturebene oder durch entsprechende Strukturierung der Modelle stattfinden. Auch auf diese Weise lässt sich der Aufwand einer eventuellen Portierung auf verschiedene Plattformen reduzieren, da im Idealfall nur wenige Diagramme bzw. Dateien betroffen sind. In der TripleLogo-Applikation wurde dies am Beispiel der Anzeigetechnologie umgesetzt, über die in den bisherigen Modellen des Verhaltens der Tiere in Abschnitt 3.3 oder der Architektur in Abschnitt 3.1 noch keine Aussage getroffen wurde. Die Klasse `SwingController` in Quellcode 3.34 kapselt genau diesen technologischen Aspekt, indem es auf das Swing-Framework von Java aufsetzt. Soll die Applikation über andere Mechanismen angezeigt werden, muss nur diese Klasse ausgetauscht werden. Auf diese Weise können die Vorteile von Plattform- und Technologie-unabhängigen Modellen einerseits und einer gesteigerten Effizienz bei der Entwicklung durch Nutzung gewachsener Legacy-Systeme wie die Java-Klassenbibliothek andererseits kombiniert werden.

3.8 Gemeinsame Betrachtung der Teilsprachen

Wurden bisher die Sprachen jeweils getrennt voneinander behandelt, sollen in diesem Abschnitt die gemeinsamen Anteile beschrieben werden. Diese gliedern sich in drei eigenen Grammatiken, die mittels der Sprachvererbung von MontiCore in die einzelnen Sprachen integriert wurden (siehe auch Abschnitt 7.1). So sind UML/P-spezifische Elemente wie Stereotypen in der `Common`-Grammatik in Quellcode 3.35, Literale wie Zeichenketten (Strings) oder Zahlenwerte in der `Literals`-Grammatik in Quellcode 3.36 und Basistypen sowie der Aufbau komplexer Typen in der `Types`-Grammatik in Quellcode 3.38 zusammengefasst. Dabei handelt es sich jeweils

um eine komprimierte Darstellung. Die vollständigen MontiCore-Grammatiken finden sich im Anhang C.1-C.3.

MC

```
package mc.umlp.common;

grammar Common extends mc.types.Types {

  external InvariantContent;

  Stereotype =
      "<<" values:StereoValue ("," values:StereoValue)* ">>";

  StereoValue =
      Name ("=" source:String)?;

  Cardinality =
      "[" (
        many:["*"] //upperBound=lowerBound=0
        |
        lowerBoundLit:IntLiteral //lowerBound=upperBound=lowerBoundLit.value
        (
          ".." ( upperBoundLit:IntLiteral //upperBound=upperBoundLit.value
                 | noUpperLimit:["*"] )   //upperBound=0
        )?
      ) "]";

  Completeness =
         complete:["(c)"]           | complete:["(c,c)"]
      | incomplete:["(...)"]        | incomplete:["(...,...)"]
      | rightcomplete:["(...,c)"] | leftcomplete:["(c,...)"];

  Modifier =
      Stereotype?
      (    ["public"|"+"]  | ["private"|"-"]  | ["protected"|"#"] | ["local"]
        | ["final"]        | ["abstract"]     | ["static"]
        | ["derived"|"/"] | ["readonly"|"?"]
      )*;

  Invariant =
      (kind:Name ":")?
      "[" content:InvariantContent(parameter kind) "]";
}
```

Quellcode 3.35: Grammatik für allgemeine Elemente der UML/P (komprimierte Darstellung)

Abgesehen davon, dass OCL/P- und Java/P-Code auch eingebettet in den anderen Sprachen Verwendung finden, bieten alle Sprachen die Möglichkeit, als eigenständige Dateien abgelegt zu werden. Es handelt sich dabei um gewöhnliche Textdateien, die anhand der Dateiendung zu unterscheiden sind (siehe Tabelle 3.37). Die Ablage erfolgt in einer Pakethierarchie ähnlich zu Java, die eine eindeutige Referenzierung und Strukturierung der Modelle ermöglicht, ohne die insbesondere in größeren Projekten mit vielen Modellen schnell die Übersichtlichkeit verloren gehen würde. Es handelt sich dabei aber explizit um eine reine Strukturierung auf Modellebene, die sich nicht im Produktivsystem wiederfinden muss. Ob und in welcher Form die Paketstruktur auf das modellierte System abgebildet wird, ist Entscheidung des Generators bzw. Entwicklers und wird von der UML/P nicht festgelegt.

MC

```
package mc.literals;

grammar Literals {

  interface Literal;
  interface NumericLiteral extends Literal;

  interface SignedLiteral;
  interface SignedNumericLiteral extends SignedLiteral;

  NullLiteral    implements Literal, SignedLiteral = "null";
  BooleanLiteral implements Literal, SignedLiteral = source:["true"|"false"];
  CharLiteral    implements Literal, SignedLiteral = source:Char;
  StringLiteral  implements Literal, SignedLiteral = source:String;

  IntLiteral     implements NumericLiteral = source:Num_Int;
  LongLiteral    implements NumericLiteral = source:Num_Long;
  FloatLiteral   implements NumericLiteral = source:Num_Float;
  DoubleLiteral  implements NumericLiteral = source:Num_Double;

  SignedIntLiteral    implements SignedNumericLiteral = ["-"]? source:Num_Int;
  SignedLongLiteral   implements SignedNumericLiteral = ["-"]? source:Num_Long;
  SignedFloatLiteral  implements SignedNumericLiteral = ["-"]? source:Num_Float;
  SignedDoubleLiteral implements SignedNumericLiteral = ["-"]? source:Num_Double;

  token Char   = '\'' ( ESC | ~('\''|'\n'|'\r'|'\\') ) '\'';
  token String = '"' ( ESC | ~('"'|'\\'|'\n'|'\r') )* '"';
  token Name   = ('a'..'z' | 'A'..'Z' | '_' | '$')
                 ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$')*;

  protected token Hex_Digit         = ('0'..'9' | 'A'..'F' | 'a'..'f');
  protected token Float_Suffix      = 'f'| 'F' | 'd' | 'D';
  protected token Decimal_Exponent = ('e'|'E') ('+'|'-')? ('0'..'9')+;
  protected token Hex_Exponent      = ('p'|'P') ('+'|'-')? ('0'..'9')+;

  token Numbers = //represents Num_Int, Num_Long, Num_Float, and Num_Double
      '.' ( '.' | '.' '.' | (('0'..'9')+ Decimal_Exponent? Float_Suffix?)? )
      |
      (
        '0' ( ('x'|'X') Hex_Digit* | ('0'..'9')+ | ('0'..'7')+ )?
        | ('1'..'9') ('0'..'9')*
      )
      (
        ('l'|'L')
        | (
            '.'? Hex_Digit* Hex_Exponent Float_Suffix?
            | '.' ('0'..'9')* Decimal_Exponent? Float_Suffix?
            | Decimal_Exponent Float_Suffix?
            | Float_Suffix
          )
      )?;

  protected token ESC =
      '\\' ( 'n' | 'r' | 't' | 'b' | 'f' | '"' | '\'' | '\\'
             | ('u')+ Hex_Digit Hex_Digit Hex_Digit Hex_Digit
             | '0'..'3' ( '0'..'7' ('0'..'7')? )?
             | '4'..'7' ('0'..'7')? );

  token WS = ( ' ' | '\t' | '\f' | ("\r\n"|'\r'|'\n') )+;
}
```

Quellcode 3.36: Grammatik für Literale (komprimierte Darstellung)

Sprache/Modell	*Kurzbezeichner*	*Dateiendung*
Klassendiagramm	CD	.cd
Objektdiagramm	OD	.od
Statechart	SC	.sc
Sequenzdiagramm	SD	.sd
Testspezifikationsprache	TC	.tc
OCL/P	OCLP	.ocl
Java/P	JavaP	.java

Tabelle 3.37: Kurzbezeichner und Dateiformat der UML/P-Sprachen

Die Dateien befolgen unabhängig von der Sprache einen gemeinsamen Grundaufbau, der sich wie die gesamte UML/P an der Programmiersprache Java orientiert. So wird zu Beginn jeder Datei das Paket aufgeführt, in dem diese abgelegt ist, gefolgt von einer Liste benötigter Importe. Letztere umfasst Angaben der voll-qualifizierten Namen einzelner Elementen anderer Modelle, die innerhalb der Datei verwendet werden. Alternativ ist auch der Import aller Elemente eines Modells oder Pakets möglich, indem statt des Element- bzw. Modellnamens ein "$*$" angegeben wird (siehe z.B. Quellcode 3.32). Dabei werden jeweils nur die Hauptelemente des Modells bzw. Pakets importiert, nicht aber deren Subelemente. Ein $*$-Import eines Pakets importiert demnach die darin enthaltenen Modelle. Um wiederum deren Elemente zu importieren ist die Angabe weiterer Importe nötig.

Importe sind genauso wie in Java nicht transitiv sondern gelten jeweils nur für das aktuelle Modell. Ebenfalls muss das eigene Paket eines Modells nicht explizit importiert werden. Importierte Elemente können im weiteren Verlauf der Datei über deren einfachen Namen referenziert werden. Darüber hinaus hat ein Import keine weitere Bedeutung, und kann ausbleiben, wenn stattdessen alle Referenzen auf externe Elemente über deren voll-qualifizierten Namen erfolgen. Welche Elemente jeweils als Import erlaubt sind, ist Kapitel 4 zu entnehmen.

Nach den Importen folgt ein für die jeweilige Sprache eindeutiges Schlüsselwort, das insbesondere eine schnelle Identifikation der jeweils vorliegenden Sprache erlaubt. Somit erfüllen diese Schlüsselwörter den gleichen Zweck, wie die Diagramm-Marken in der ursprünglichen Fassung der UML/P (siehe [Rum04a]). Der frei wählbare Name für die vorliegende Sprachinstanz, der nach dem Schlüsselwort angegeben wird, entspricht dem Dateinamen und muss innerhalb eines Pakets eindeutig sein. Insgesamt setzt sich damit der voll-qualifizierte Name eines Elements innerhalb einer Sprachinstanz aus den jeweils durch Punkte getrennten Namen des Pakets, der Sprachinstanz sowie des entsprechenden Elements selbst zusammen.

An diesen allgemeinen Aufbau schließt sich der eigentliche Inhalt des Modells an, der in geschweiften Klammern eingefasst wird. Dieser wurde im Einzelnen in den vorherigen Abschnitten beschrieben. Bisher nur am Rande oder unerwähnt blieben dabei Stereotypen und die in der UML/P eingeführten Repräsentationsindikatoren, die in fast allen Teilsprachen der UML/P

```
package mc.types;                                                    MC

grammar Types extends mc.literals.Literals {

  interface Type extends TypeArgument, ReturnType;
  interface ReferenceType;
  interface TypeArgument;
  interface ReturnType;

  QualifiedName = parts:Name ("." parts:Name)*;

  ComplexArrayType:ArrayType implements Type returns Type =
      ComplexType ("[" "]")*;

  PrimitiveArrayType:ArrayType implements Type returns Type =
      PrimitiveType ("[" "]")*;

  VoidType implements ReturnType = "void";

  PrimitiveType implements Type = FloatingPointType | IntegralType | BooleanType;

  NumericType:PrimitiveType = FloatingPointType | IntegralType;

  BooleanType:PrimitiveType =
      ["boolean"];

  IntegralType:PrimitiveType =
      ["byte"] | ["short"] | ["int"] | ["long"] | ["char"];

  FloatingPointType:PrimitiveType =
      ["float"] | ["double"];

  ComplexType returns Type =
      ret=SimpleReferenceType ("." ret=QualifiedType->[ret])*;

  SimpleReferenceType implements ReferenceType =
      Name ("." Name)* TypeArguments?;

  QualifiedType [qualification:Type] =
      Name TypeArguments?;

  TypeArguments =
      "<" TypeArgument ("," TypeArgument)* ">";

  WildcardType implements TypeArgument =
      "?" ( "extends" upperBound:Type | "super" lowerBound:Type )?;

  TypeParameters =
      "<" TypeVariableDeclaration ("," TypeVariableDeclaration)* ">";

  TypeVariableDeclaration =
      Name ( "extends" upperBounds:ComplexType
             ("&" upperBounds:ComplexType)* )?;
}
```

Quellcode 3.38: Grammatik für Typen (komprimierte Darstellung)

jeweils Anwendung finden. Eine Ausnahme bildet zurzeit noch Java/P, da vorerst die vollständige Abdeckung des Java-Standards bei der Entwicklung im Vordergrund stand. Dennoch können

diese allgemeinen Elemente auch hier ganz analog angewendet werden, so dass eine entsprechende Integration in einer zukünftigen Version des Frameworks durchaus sinnvoll ist.

Stereotypen können für viele Elemente der UML/P-Sprachen angegeben werden (siehe Anhang C) und verkörpern ähnlich den Modifikatoren eine Klassifizierung des damit ausgezeichneten Elements oder eine Zuweisung zusätzlicher Eigenschaften. Sie werden in doppelten spitzen Klammern eingefasst (`<< >>`) und bestehen aus frei wählbaren Bezeichnern oder - in der textuellen Fassung - aus Name-Wert-Paaren, wobei letztere die aus der UML ebenfalls bekannten Merkmale ersetzen (siehe auch Abschnitt 3.9). Es können auch mehrere Stereotypen durch Kommata getrennt in einem Klammerpaar angegeben werden. Ein Stereotyp bezieht sich dabei immer auf das folgende Element, was bei hierarchischen Strukturen die inneren Elemente mit einschließt. Insbesondere gilt dieser für das gesamte Modell, wenn er vor dem einleitenden Sprachbezeichner angegeben wird. Neben den in den vorigen Abschnitten bereits erwähnten vordefinierten Stereotypen, können Werkzeugentwickler beliebige weitere Stereotypen kreieren. Für die technische Umsetzung bietet das UML/P-Framework entsprechende Schnittstellen an (siehe Abschnitt 6.2 sowie Kapitel 7). Auf diese Weise kann die Sprache syntaktisch und semantisch erweitert werden, etwa um diese an projekt- oder domänenspezifische Besonderheiten anzupassen. Dies wird als Profilbildung bezeichnet. So könnten z.B. Stereotypen eingesetzt werden, um Entwurfsmuster innerhalb von Klassendiagrammen zu kennzeichnen, die ein Generator erkennen und in entsprechenden Code umsetzen kann. Ein anderes Beispiel ist eine Angabe wie `<<persistent>>` zur Markierung von Klassen die auf der Festplatte oder in einer Datenbank gespeichert werden sollen. Um deren Bedeutung zu definieren, wird in [Rum04a] eine tabellarische Form vorgeschlagen. Je nach Anwenderkreis kann aber auch eine informelle Beschreibung ausreichend sein.

Bei einem *Repräsentationsindikator* handelt es sich ebenfalls um eine Auszeichnung des folgenden Elements ähnlich den Stereotypen, allerdings mit einer festgelegten Aussage über die Vollständigkeit der Darstellung. So kann z.B. eine Klasse in der textuellen Syntax mit dem Vollständigkeitsindikator `(c)` markiert werden, um damit anzuzeigen, dass sämtliche Attribute und Methoden der Klasse dargestellt wurden. Fehlt eine solche Markierung oder wird stattdessen der Unvollständigkeitsindikator `(...)` verwendet, kann es sich um eine unterspezifizierte Darstellung handeln. Wichtig ist, dass hier nur auf die Möglichkeit einer Unterspezifikation hingewiesen wird. Insbesondere stellt es keinen Widerspruch dar, falls es sich doch um eine vollständige Darstellung handeln sollte. Die textuelle Syntax weicht hier von den in [Rum04a] vorgeschlagenen Zeichen © und ... ab, um für den Vollständigkeitsindikator eine einfache Eingabe zu ermöglichen. Die Klammern des Unvollständigkeitsindikators wurden für eine einheitliche Darstellung hinzugefügt.

In der graphischen Notation können Repräsentationsindikatoren nicht nur für einzelne Elemente oder Diagramme angegeben werden, sondern insbesondere bei Klassendiagrammen auch für die Felder, die Attribute und Methoden unterteilen. Da diese Felder in der textuellen Syntax nicht dargestellt werden, wird zusätzliche in einen durch Komma getrennten rechten und linken Repräsentationsindikator unterschieden. So bezeichnet z.B. `(...,c)` einen linken Unvollständigkeits- und einen rechten Vollständigkeitsindikator. Die Darstellung von `(...,...)` und `(c,c)` ist dabei

gleichbedeutend mit `(...)` bzw. `(c)`. Die genaue Bedeutung für die jeweiligen Diagramme ist Tabelle 3.39 zu entnehmen.

Sprache	*Element*	*Bedeutung*	
		links	*rechts*
CD	Diagramm	Typen	Assoziationen
	Typen	Attribute	Methoden
OD	Diagramm	Objekte	Links
	Objekte	Attribute	Attribute
SC	Diagramm	Zustände	Transitionen
	Zustände	Subzustände	Transitionen (der Subzustände)
SD	Diagramm	Interaktionen	Interaktionen
	Objekte	Interaktionen (des Objekts)	Interaktionen (des Objekts)

Tabelle 3.39: Bedeutung der Repräsentationsindikatoren

Bei Sequenzdiagrammen haben die Repräsentationsindikatoren `(c)` und `(...)` jeweils die gleiche Bedeutung wie die Stereotypen `<<match_complete>>` und `<<match_visible>>` (siehe [Rum04a] und Abschnitt 3.4). Eine Anwendung auf die in Tabelle 3.39 nicht erwähnten Sprachen OCL/P, Java/P sowie die Testspezifikationssprache ist in der UML/P bisher nicht vorgesehen.

Eine letzte, aber dadurch nicht weniger wichtige Gemeinsamkeit aller Sprachen ist die Möglichkeit, an beliebigen Stellen innerhalb einer Datei *Kommentare* anzugeben. Deren Format wurde ebenfalls dem Java-Standard entnommen, so dass einzeilige Kommentare durch `//` angeführt sowie mehrzeilige Kommentare in `/* */` eingefasst werden und ansonsten aus beliebigem Freitext bestehen. Dieser enthält im Allgemeinen keine semantisch relevanten Aspekte und dient allein dem Zweck, zusätzliche Erläuterungen und Anmerkungen einem Modell hinzuzufügen. Dennoch gibt es auch immer wieder Ansätze, die Kommentare für die Erweiterung einer Sprache nutzen, ohne deren Syntax zu verändern. So wurde z.B. eine leichtgewichtige Templateengine mit Hilfe von Kommentaren in Java-Klassen entwickelt [KR06]. Diese Form der Spracherweiterung wird im Rahmen dieser Arbeit nicht weiter betrachtet, ist aber prinzipiell nicht ausgeschlossen. Allerdings wird empfohlen, möglichst die für solche Zwecke vorgesehenen Stereotypen zu verwenden, die ebenfalls eine einfache Erweiterung der Semantik von Elementen der UML/P ohne Syntaxänderung erlauben.

3.9 Unterschiede zur ersten Fassung der UML/P

In den vorherigen Abschnitten wurden schon einige Unterschiede der textuellen zu der ursprünglich in [Rum04a] eingeführten graphischen Fassung der UML/P angesprochen. Im Folgenden sollen

nun diese und weitere Unterschiede abschließend zusammengefasst werden. Abweichungen zum UML-Standard der OMG wurden hingegen bereits in [Rum04a] ausführlich behandelt und deshalb hier nicht mehr explizit betrachtet.

3.9.1 Allgemeine Notation und neue Konzepte

Ein guter Teil der Abweichungen ist durch die unterschiedlichen Anforderungen textueller und graphischer Sprachen [GKR+07, KKP+09] und Erweiterungen des Java-Standards entstanden. Orientierte sich die ursprüngliche Fassung der UML/P noch an Java 1.4 [Rum04a], wurden in Java 1.5 mit Enumerationen und generischen Typen grundlegend neue Konzepte eingeführt, die sich als sinnvolle Erweiterungen herausgestellt haben und deshalb in die UML/P integriert wurden. Diese bietet nun neben Klassen und Interfaces mit den *Enumerationen* eine dritte Art von Typen, die eine einfachere und kompaktere Darstellung von Datentypen mit einem endlichen Wertebereich erlaubt, als dies mit Klassen und konstanten Attributen möglich ist. Darüber hinaus werden Typen im Allgemeinen mit Hilfe von anderen, bereits bestehenden Typen definiert, indem diese für Attribute sowie Rückgabewerte und Parametertypen von Methoden verwendet werden. Von diesen kann über sogenannte *generische Typen* (Generics) abstrahiert werden, indem anstelle eines konkreten Typs ein Name als Platzhalter angegeben wird. Dieser wird als Typparameter bezeichnet und kann für ganze Klassen und Interfaces oder auch nur für einzelne Methoden und Konstruktoren angegeben werden. Der Gültigkeitsbereich eines Typparameters beschränkt sich jeweils auf das Element, für das er definiert wurde. Der dafür einzusetzende konkrete Typ - das Typargument - wird in Subklassen oder erst bei Instanziierung bzw. im Falle von Typparametern an Methoden beim jeweiligen Aufruf festgelegt. Das Konzept der Generics erlaubt gegenüber den bisherigen Notationsformen eine abstraktere und damit kompaktere sowie flexiblere Definition von Typen, so dass generische Typen auch in den UML-Standard Einzug gehalten haben [OMG10d]. Die Notation in der textuellen Fassung der UML/P erfolgt dabei analog zum Java-Standard, indem die Platzhalter bei der Typdefinition und die konkreten Typzuweisungen bei der Instanziierung jeweils hinter dem Typnamen in einfachen spitzen Klammern angegeben werden. Bei Methoden bzw. Konstruktoren werden die Typparameter hingegen bei der Definition nach dem Modifikator und beim Aufruf vor dem Methoden- bzw. Konstruktornamen notiert (siehe C.3 im Anhang C.2 und [GJS05]). Auf diese Weise können generische Typen in allen Teilsprachen der UML/P einheitlich verwendet werden. Darüber hinaus lässt sich diese Syntax leicht auf die graphische Notation übertragen. Auch im UML-Standard ist eine ähnliche Syntax vorgesehen und alternativ eine Annotation in Form eines abgesetzten gestrichelten Rechtecks an der äußeren Umrandung von Klassen. In beiden Fällen ist jedoch keine Beschränkung auf Methoden möglich. Abbildung 3.40 und Quellcode 3.41 stellt in einem Beispiel beide Notationsformen gegenüber.

Bei der Entwicklung der textuellen Syntax wurde darauf geachtet, nur Zeichen aus dem ASCII-Standard[9] zu verwenden, um eine möglichst flexible Nutzung der Sprache zu erlauben [GKR+07, KKP+09], wie z.B. die leichte Eingabe mit beliebigen Texteditoren. Dies hat insbesondere zu

[9] American Standard Code for Information Interchange

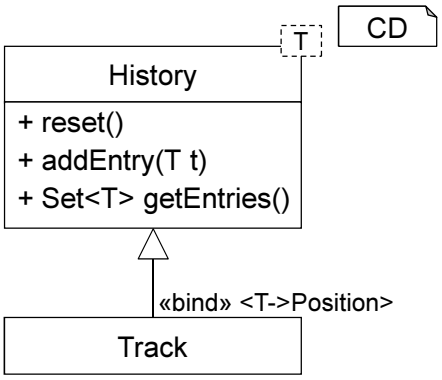


Abbildung 3.40: Generische Typen (graphisch)

```
1  package mc.tl;
2
3  import mc.tl.TripleLogo.Position;
4  import java.util.Set;
5
6  classdiagram TripleLogoTracking {
7
8    class History<T> {
9      + reset();
10     + addEntry(T t);
11     + Set<T> getEntries();
12   }
13
14   class Track extends History<Position>;
15
16 }
```

Quellcode 3.41: Generische Typen (textuell)

Änderungen bei den Stereotypen, Diagramm-Marken und Repräsentationsindikatoren geführt. So wurden für *Stereotypen* anstatt der französischen Anführungszeichen und dem Copyright-Symbol des *Vollständigkeitsindikators*, die jeweils kein Teil des ASCII-Standards sind, mit `<< >>` bzw. `(c)` möglichst ähnlich aussehende Notationen gewählt. Die *Diagramm-Marken*, die in der graphischen Darstellung rechts oben an den Modellen angegeben werden (siehe z.B. Abbildung 3.5 und Anhang B), wurden hingegen durch den in Abschnitt 3.8 beschriebenen einheitlichen Aufbau der textuellen Modelle abgelöst. Zusätzlich wurden *Importe*, eine *Paketstruktur* und ein pro Paket eindeutiger Name für Modelle eingeführt, der eine Referenzierung von Modellen und Modellelementen über *voll-qualifizierte Namen* ermöglicht. In der graphischen Notation wurde noch von global eindeutigen Namen aller Elemente ausgegangen, was vor allem den Umgang mit Modellen in größeren Projekten oder den Aufbau von Modellbibliotheken erschwert, da die Gefahr von Namenskollisionen durch die kurzen, nicht qualifizierten Namen naturgemäß ansteigt. Insbesondere die Nutzung mehrerer fremder Modellbibliotheken kann dadurch sogar unmöglich gemacht werden, da Namenskollisionen in diesen Bibliotheken nicht einfach behoben werden können. Dieses Problem zeigt sich schon allein durch die bereits in der ursprünglichen UML/P vorgesehene Verwendung von Java-Typen innerhalb der Modelle. So reicht z.B. für die Referenzierung der Klasse `Date` der einfache Name nicht aus, da eine solche Klasse in den Paketen `java.util` und `java.sql` der Java-Bibliothek definiert wurde.

Zusammen mit der Paketstruktur wurden die aus Java und der UML bekannten Sichtbarkeitsangaben `public`, `protected` und `private` mit `local` um einen zusätzlichen Modifikator ergänzt, der die Sichtbarkeit des damit ausgezeichneten Elements auf das Modell beschränkt.

Neben diesen Erweiterungen wurde die im Rahmen dieser Arbeit entwickelte Fassung der UML/P auch in einem Punkt reduziert. So unterscheidet die textuelle Notation nicht mehr zwischen Merkmalen und Stereotypen, wie dies sowohl im UML-Standard als auch in der ursprünglichen UML/P vorgesehen ist. Stattdessen wurde eine um Name-Wert-Paare erweiterte

Syntax für Stereotypen eingeführt, die die Merkmale vollständig ersetzt. Diese Änderung basiert auf zwei Gründen:

1. Der Unterschied zwischen Stereotypen und Merkmalen ist oft marginal und dadurch schwer zu vermitteln. So wird mit Merkmalen eine Eigenschaft des damit gekennzeichneten Modellelements beschrieben, während Stereotypen letzteres klassifizieren. Diese Klassifizierung steht aber im Allgemeinen wiederum für bestimmte Eigenschaften, die dem Element dadurch zugeordnet werden. Letztendlich haben beide eine definierte Implikation auf das ausgezeichnete Element und können damit äquivalent eingesetzt werden. Allerdings kann bei Merkmalen im Gegensatz zu Stereotypen neben einem Namen zusätzlich ein Wert angegeben werden, so dass hier die Syntax von Stereotypen entsprechend erweitert wurde.
2. Merkmale werden in der UML innerhalb von geschweiften Klammern notiert. Da sich die textuelle Syntax der UML/P jedoch an Java orientiert, werden diese Klammern an vielen anderen Stellen bereits zur Gruppierung von Elementen eingesetzt. Aus diesem Grund würden sich Merkmale im Gegensatz zu der Schreibweise von Stereotypen nur wenig von anderen Modellelementen abheben und so die Lesbarkeit des Modells erschweren.

Für eine knappere Schreibweise und bessere Lesbarkeit werden in der textuellen Notation mehrere Stereotypen an einem Element innerhalb eines Klammerpaares eingefasst, indem die einzelnen Werte durch Kommata voneinander getrennt werden, anstatt diese jeweils in einzelnen Klammern anzugeben. Darüber hinaus kann im Sinne der Wiederverwendung ein Stereotyp mit Hilfe mehrerer bereits definierter Stereotypen beschrieben werden.

3.9.2 Semantik und Spracheinbettung

Die größte Änderung in der Semantik hat Java/P erfahren, deren Ausdrücke sich nicht mehr auf den generierten Code sondern ausschließlich auf die Modelle direkt beziehen (siehe Abschnitt 3.7). Dies ist unabhängig davon, ob Java/P als eingebetteter Code in den Modellen oder in Form eigenständiger Dateien verwendet wird. Die Unterscheidung zwischen Modell- und Systembezug ist in sofern relevant, da in dieser Arbeit davon ausgegangen wird, dass sich die modellierten Strukturen nicht notwendigerweise identisch im System wieder finden müssen. Dadurch können Codegeneratoren die modellierten Strukturen an die jeweilige Systemplattform anpassen und auf Eigenschaften wie Effizienz oder Testbarkeit hin optimieren. Letzteres wird etwa durch get- und set-Methoden für Attributzugriffe und den Einsatz von Factories erreicht, wodurch der Zugriff auf Attribute und Klassen im System im Vergleich zum Modell unterschiedlich ausfällt. Insbesondere bei der Verwendung von nicht objektorientierten Sprachen als Zielplattform sind umfangreiche Anpassungen notwendig, um die Konzepte der Modellebene auf Konzepte der Systemebene abzubilden. Aber auch bei objektorientierten Zielsprachen wie Java muss eine Klasse im Modell nicht zwingend auf eine Klasse im System abgebildet werden, wie die Codegenerierung in Abschnitt 6.3 zeigt.

Abbildung 3.42 illustriert den Zusammenhang zwischen Modell- und Systemebene wie er in [Rum04a] dargestellt wird. Danach besteht auf der Modellebene zwischen den prototypischen

Objekten und Klassen eine ähnliche Instanzbeziehung wie zwischen Objekten und Klassen der Systemebene. Auch zwischen den Ebenen lässt sich eine Beziehung beobachten. So lassen sich die Klassen der Modelle auf Klassencode und entsprechend die prototypischen auf reale Objekte abbilden. Diese Ansicht wird hier in sofern relativiert, als dass der Klassencode nur konzeptuell für eine Klasse im System steht. Ob der Klassencode bezogen auf den konkreten Systemcode ebenfalls genau eine Klasse im Sinne des objektorientierten Paradigmas darstellt, ist damit nicht festgelegt. Entsprechend können die Objekte der Systemebene als Abstraktion der Strukturen zur Laufzeit des Systems angesehen werden[10]. Unter dieser Annahme hat die Darstellung aus [Rum04a] auch in dieser Arbeit weiterhin ihre Gültigkeit.

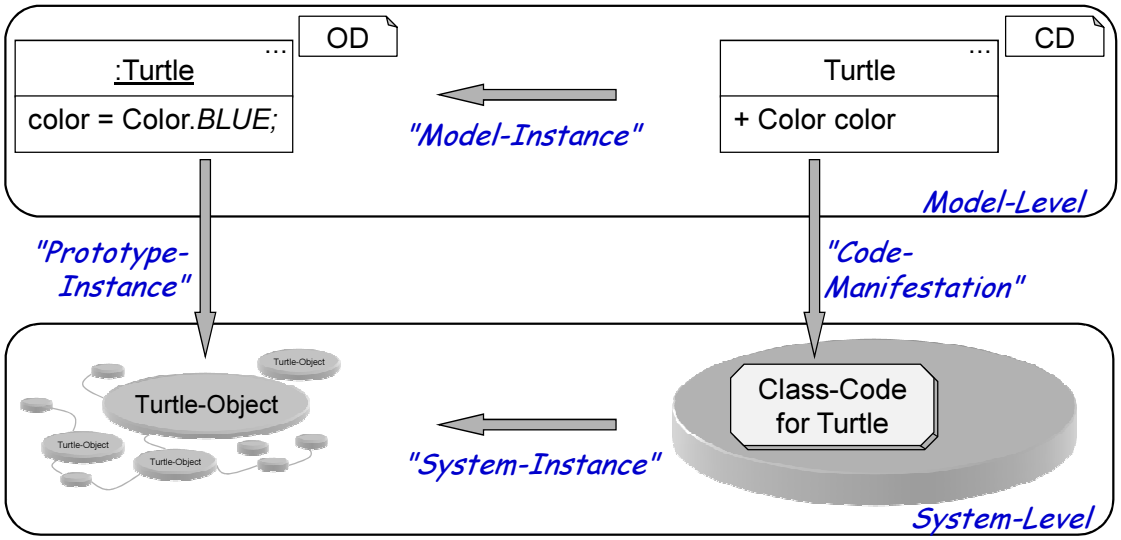


Abbildung 3.42: Beziehungen zwischen Modell- und Systemebene (adaptiert aus [Rum04a])

Durch die Anhebung von Java/P auf die Modellierungsebene wird die UML/P unabhängig von der Sprache des Zielsystems. Darüber hinaus bietet das im Rahmen dieser Arbeit entwickelte Framework die Möglichkeit, die in Modellen integrierten Sprachanteile wie Java/P mit sehr geringem Aufwand zu ersetzen, so dass in Zukunft Varianten der UML/P entstehen können, die die Einbettung anderer Sprachen wie C++ oder Haskell erlauben (siehe Abschnitt 7.1). Bei Sprachen wie Haskell, die anders als die umgebenden Diagramme nicht dem objektorientierten Paradigma entsprechen, muss jedoch die Zusammenarbeit mit den anderen Elementen der UML/P geklärt werden. Es ist sogar vorstellbar, das Framework so zu erweitern, dass mehrere Sprachen parallel in einem Modell eingebettet werden können. Für Bedingungen ist dies bereits jetzt vorgesehen. Konnten diese ursprünglich nur in OCL/P formuliert werden, erlaubt die textuelle Notation alternativ auch die Verwendung von Java/P. Um zwischen diesen beiden Möglichkeiten

[10]Diese Abstraktion findet sich z.B. auch bei der Umsetzung von Java-Code in ausführbaren Maschinencode durch den Compiler wieder. Sowohl der Maschinencode als auch die Laufzeitstrukturen in Form von Speicherbereichen und Programmzähler repräsentieren nur noch indirekt Klassen bzw. Objekte.

im Modell zu unterscheiden, kann vor der Bedingung mit `java:` bzw. `ocl:` die nachfolgend verwendete Sprache festgelegt werden (vgl. Quellcode 3.33).

3.9.3 Lokale Unterschiede

Bezogen sich die bisher beschriebenen Unterschiede mehr oder weniger auf alle Sprachen der UML/P gleichermaßen, wird im Folgenden auf Abweichungen in einzelnen Teilsprachen eingegangen.

In Klassendiagrammen wurde mit der Berücksichtigung von Mehrfachvererbung, Aggregationen und weiter gefassten Kardinalitäten einige in der ursprünglichen UML/P nicht vorhandene bzw. eingeschränkte Notationen aus dem UML-Standard übernommen. So sind in [Rum04a] nur die *Kardinalitäten* `1`, `0..1` ("optional") und `*` ("beliebig viele") vorgesehen. Andere Kardinalitäten können nur über zusätzliche OCL/P-Invarianten ausgedrückt werden. Insbesondere relativ häufig verwendete Kardinalitäten wie `1..*` ("mindestens 1") sind dadurch nur umständlich zu modellieren. Aus diesem Grund können in der textuellen Schreibweise beliebige ganze Zahlen und der Kleene Stern als einzelner Wert oder als über `..` getrennte obere und unter Grenze angegeben werden. Darüber hinaus können neben der Implementierung mehrerer Interfaces Klassen nun auch von mehr als einer anderen Klasse erben. Diese *Mehrfachvererbung* wurde insbesondere aus dem Grund integriert, da die vorliegende Fassung der UML/P und das entwickelte Framework nicht mehr ausschließlich auf die Einbettung und Generierung von Java-Code festgelegt sind. Denn im Gegensatz zu Java ist in einigen Programmiersprachen wie C++ oder Python Mehrfachvererbung vorgesehen, so dass nun die UML/P wie auch der UML-Standard dem Modellierer mehr Freiheiten vor allem beim Entwurf von Systemen in diesen Zielsprachen erlaubt. Allerdings können bei Mehrfachvererbung Mehrdeutigkeiten auftreten, wenn Methoden gleicher Signatur in verschiedenen Superklassen definiert wurden. Je nach Einsatzzweck der Modelle muss dies aber nicht notwendigerweise ein Problem darstellen, etwa wenn es sich um ein in der Anfordungsanalyse entworfenes, unterspezifiziertes Modell handelt, das im Laufe der Entwicklung noch präzisiert wird. Für die Generierung von Produktivcode könnten dann Mehrfachvererbungen nur erlaubt werden, wenn keine Mehrdeutigkeiten existieren, oder der Generator in der Lage ist, diese geeignet aufzulösen. Es ist demnach an dieser Stelle nicht Aufgabe der Modellierungssprache, diesbezüglich eine Festlegung zu treffen. Trotzdem sind in Zukunft Erweiterungen der UML/P denkbar, um solche Mehrdeutigkeiten aufzulösen. So sieht C++ etwa spezifische Qualifikatoren vor, über die bei einer Mehrdeutigkeit die zu erbende Methode explizit bestimmt werden kann. Ein ähnlicher Mechanismus ließe sich über entsprechende Stereotypen in der UML/P nachbilden.

Ebenfalls aus dem UML-Standard übernommen wurde die *Aggregation* als schwächere Form der Komposition. Über beide wird modelliert, dass sich ein Objekt aus mehreren anderen Objekten zusammensetzt. Im Gegensatz zur Komposition sind die Teilobjekte bei der Aggregation jedoch austauschbar und können auch einzeln existieren.

Ähnlich wie in Java sieht die ursprüngliche Fassung der UML/P nur Konstanten als Attribute für *Interfaces* vor. Diese Einschränkung wurde in der vorliegenden Arbeit aufgehoben, so dass Attribute genauso als Schnittstellenbeschreibung verwendet und an Subtypen vererbt werden

können, wie dies bei Methoden der Fall ist. Auf diese Weise lassen sich Attribute z.B. für die Generierung von Getter- und Setter-Schnittstellen einsetzen. Konstanten lassen sich wie üblich über den Modifikator `final` deklarieren und sind die einzigen Attribute in Interfaces, die die Angabe von Werten erlauben (siehe Abschnitt 4.2.2).

Objektdiagramme nutzen ebenfalls die beschriebenen Erweiterungen der Kardinalitäten und bieten entsprechende Link-Instanzen für Assoziationen und Kompositionen. Darüber hinaus können hier mehrere Links eines Objekts sehr kompakt dargestellt werden, indem auf einer Seite eine Komma-separierte Liste von Objektreferenzen angegeben wird. In solch einem Fall ist jedes Objekt der einen Seite mit jedem Objekt der jeweils anderen Seite verlinkt. So modelliert z.B. der Ausdruck "`link o1 -- o2, o3, o4;`" drei Links des Objekts `o1` zu den jeweiligen Objekten auf der rechten Seite. Ein weiteres Objekt auf der linken Seite würde hingegen insgesamt sechs Links repräsentieren. Eine vergleichbare kompakte Darstellung ist in der graphischen Notation nicht möglich.

Die OCL/P erlaubt den Einsatz von Objektdiagrammen als Aussagen über das System, die mit weiteren Bedingungen ergänzt oder logischen Operatoren wie `&&`, `||` oder `!` kombiniert werden können (siehe [Rum04a]). Auf diese Weise können mit Hilfe von Objektdiagrammen in OCL/P-Bedingungen unerwünschte oder geforderte Situationen im System beschrieben sowie mehrere Objektdiagramme zu einem kombiniert werden. Sah die ursprüngliche OCL/P für den Verweis auf ein Objektdiagramm noch dessen Namen mit einem vorangestellten "`OD.`" vor, wurde dies in der vorliegenden Fassung im Sinne einer einheitlichen und konsistenten Referenzierung durch die Angabe des voll-qualifizierten Namens des Objektdiagramms ersetzt. Darüber hinaus bietet die OCL/P nun wie alle Teilsprachen die Möglichkeit, Importe anzugeben und somit innerhalb der Ausdrücke nur den einfachen Namen zu verwenden.

In der ursprünglichen Java/P ist ebenfalls die Einbettung von OCL/P vorgesehen, indem die Java-Syntax durch spezielle Konstrukte erweitert wurde. Dies ist in der vorliegenden Fassung noch nicht berücksichtigt, da der Fokus auf einer vollständigen Abdeckung der Syntax von Java 1.5 lag, kann aber in zukünftige Versionen des Frameworks nachträglich integriert werden.

Wie bereits in Abschnitt 3.4 beschrieben, stellte die Schaffung einer textuellen Notation für Sequenzdiagramme eine besondere Herausforderung dar, da es sich um die einzige Diagrammart handelt, bei der die zweidimensionale Anordnung der Elemente von Bedeutung ist. Trotzdem ist eine vollständige Übertragung auf eine sequenzielle Textform gelungen. Abgesehen von den allgemeinen Abweichungen wurden bei Sequenzdiagrammen ansonsten keine weiteren Änderungen vorgenommen.

Anhand der in diesem Kapitel aufgeführten Erläuterungen und Beispiele wurde ein Überblick über die Bedeutung und Konzepte der Teilsprachen der UML/P gegeben. Dabei wurde sowohl auf die Einsatzmöglichkeiten der Sprachen im Einzelnen als auch deren gemeinsame Verwendung eingegangen, wobei der Fokus auf der im Rahmen dieser Arbeit entwickelten textuellen Fassung der UML/P lag. Eine detaillierte Darstellung der graphischen Notation findet sich in [Rum04a].

Im Gegensatz dazu bietet die textuelle Fassung jedoch einige Änderungen und inhaltliche sowie konzeptuelle Erweiterungen, auf die in diesem Kapitel ebenfalls eingegangen wurde.

Viele Bedingungen, die ein fehlerfreies Modell ausmachen, lassen sich bereits aus den bisherigen Beschreibungen ableiten. Eine genauere Betrachtung und Diskussion über die korrekte Verwendung der Sprachen findet sich in Kapitel 4. Darüber hinaus wurde auf Basis der textuellen Syntax in [Grö10] eine formale Semantik der UML/P erarbeitet.

Kapitel 4

Kontextbedingungen

In diesem Kapitel werden Bedingungen der Sprachen spezifiziert, die sich nicht allein aus den in Anhang C aufgeführten Grammatiken ergeben, aber syntaktisch überprüfbar sind. Die Grundlagen für diese sogenannten Kontextbedingungen werden in Abschnitt 4.1 beschrieben. Darüber hinaus wird eine Kategorisierung eingeführt, anhand derer die Bedingungen in den folgenden Abschnitten 4.2 bis 4.5 geliedert werden.

4.1 Grundlagen

Die Syntax textueller Sprachen wird häufig mit Hilfe kontextfreier Grammatiken definiert. Dies trifft auch auf die in dieser Arbeit entwickelte textuelle Version der UML/P zu, die in der EBNF-ähnlichen Beschreibungssprache von MontiCore entworfen wurde (siehe Anhang C). Aufgrund des fehlenden Kontextbezugs ist es mit diesen Grammatiken im Allgemeinen nicht möglich, die Korrektheit eines Ausdrucks vollständig daraus abzuleiten. So wird z.B. nur der Aufbau eines Methodenaufrufs beschrieben, nicht aber, in welchen Fällen ein Aufruf zulässig ist. Dies hängt u.a. davon ab, ob eine entsprechende Methode definiert und welche Sichtbarkeit vergeben wurde. Da für eine entsprechende Analyse die Modelle nicht ausgeführt werden müssen, sondern die Überprüfung allein auf Basis der (abstrakten) Syntax möglich ist, wird dies auch als *statische Analyse* bezeichnet.

Kontextbedingungen sind somit syntaktisch prüfbare Regeln, die die zulässigen Ausdrücke einer Sprache einschränken. Gemeinsam mit der kontextfreie Syntax definieren sie die Menge der *wohlgeformten Modelle* [HR00]. Da die Bedingungen zu einem gewissen Grad die Bedeutung (Semantik) einer Sprache berücksichtigen, wird im Compilerbau die Menge der Kontextbedingungen auch als *statische Semantik* bezeichnet [ASU86]. Der Begriff “Semantik” ist an dieser Stelle jedoch irreführend, da hierunter im Allgemeinen eine Abbildung der syntaktischen Ausdrücke einer Sprache auf deren Bedeutung sowie die Beschreibung der möglichen Bedeutungen selbst (die *semantische Domäne*) verstanden wird [HR04b]. Kontextbedingungen schränken die Menge der gültigen Ausdrücke zwar ein, so dass diesen eine Semantik gegeben werden kann, beinhalten

aber weder eine semantische Abbildung noch eine Beschreibung der semantischen Domäne. Aus diesem Grund wird im Folgenden der Ausdruck “statische Semantik” nicht weiter verwendet.

Die Semantik einer Sprache kann auf verschiedene Arten definiert werden. In dieser Arbeit und auch in [Rum04a] wurde für die UML/P eine informelle Beschreibung gewählt. Diese findet sich im Wesentlichen im Kapitel 3 und wird durch die Erläuterungen in diesem Kapitel ergänzt. Darüber hinaus wurde in [Grö10] eine formale Semantik der UML/P erarbeitet. Die Formalisierung basiert mit dem Systemmodell auf einer Sammlung mathematischer Theorien, die objektbasierte Systeme abstrakt beschreiben [BCGR09a, BCGR09b, BCGR08, Rum96].

Welche Aspekte einer Sprache syntaktisch oder als Kontextbedingung beschrieben werden, ist nicht immer eindeutig, sondern liegt oft in der Entscheidung des Sprachdesigners. Er muss dabei zwischen einer kompakten Formulierung der Grammatik und einer präzisen Abdeckung der angestrebten Syntax abwägen. So sieht die UML/P zum Beispiel die Angabe verschiedener Modifikatoren für bestimmte Sprachelemente vor. In der Grammatik wurde dies so umgesetzt, dass beliebig viele Modifikatoren an diesen Elementen stehen können. Dadurch sind jedoch auch nicht sinnvolle Kombinationen möglich, die über entsprechende Kontextbedingungen ausgeschlossen werden müssen. Auch wenn es die kontextfreie Grammatik prinzipiell erlaubt, nur die gültigen Kombinationen aufzuführen, würde dies zu einer sehr unübersichtlichen und schwer verständlichen Grammatik führen. Die Beschreibungssprache von MontiCore wurde aber so entworfen, dass die Grammatik einer Sprache auch zu deren Dokumentation geeignet ist. Dies würde durch die Aufzählung aller Kombinationsmöglichkeiten erschwert. Hinzu kommt, dass es durch die Behandlung über Kontextbedingungen möglich ist, dem Modellierer in einem Werkzeug hilfreichere Fehlermeldungen an die Hand zu liefern, als die, die vom Parser erzeugt werden.

Ein weiterer Aspekt in diesem Zusammenhang ist die Möglichkeit, Kontextbedingungen zu gewichten und projektspezifisch zusammenzustellen. Dies ist vor allem bei der UML/P von Interesse, die den Einsatz von Modellen in verschiedenen Phasen der Softwareentwicklung vorsieht. So stellt ein in der Anforderungsanalyse entworfenes Modell ganz andere Anforderungen an die Korrektheit und den Detailgrad als ein Modell, das als Eingabe für einen Codegenerator eingesetzt werden soll. Erstere sind im Allgemeinen hochgradig unterspezifiziert und abstrakt und werden erst im Laufe der Entwicklung detailliert. Zum Beispiel könnte in der Anforderungsanalyse bereits Situationen im zu entwerfenden System mit Hilfe von Objektdiagrammen modelliert werden, die sich auf Typen und deren Eigenschaften beziehen, die noch gar nicht anderweitig spezifiziert wurden. In solch einem Fall könnte ein Werkzeug den Modellierer zwar auf diesen Umstand in Form einer Warnung hinweisen, so dass fehlende Typen leicht identifizierbar sind, insgesamt das Modell aber als korrekt bewerten. Auf diese Weise können später die erstellten Objektdiagramme für den Entwurf der Systemarchitektur herangezogen werden, ohne den Modellierer durch zu frühzeitige Fehlermeldungen zu belasten. Soll schließlich aus den Objektdiagrammen Code z.B. für Testfälle generiert werden, ist hingegen eine vollständige Definition aller referenzierten Typen notwendig, da nur so lauffähige Testfälle entstehen können. In diesem Fall muss also dieselbe Kontextbedingung stärker gewertet werden und im Zweifelsfall zum Abbruch der Generierung führen.

Neben der Unterscheidung je nach Entwicklungsphase können auch projekt- oder domänenspezifische Profile zu einer geänderten Gewichtung und insbesondere zu weiteren Kontextbedingungen führen (siehe auch Abschnitt 6.2). Genauso ist der Wegfall von Bedingungen denkbar, wenn diese durch andere Mechanismen wie etwa einen intelligenten Codegenerator sichergestellt werden. Die sich daraus ergebenden Anforderungen an die Konfigurier- und Erweiterbarkeit von Kontextbedingungen werden durch das in Abschnitt 7.3 beschriebene Framework umgesetzt. In den folgenden Abschnitten wird hingegen ein Satz von allgemein sinnvollen Kontextbedingungen erarbeitet, die als Grundlage für spezifische Profile dienen können. Da die UML/P den gemeinsamen Einsatz mehrerer Modelle und Sprachen vorsieht, ergeben sich sowohl Bedingungen, die sich auf ein einzelnes Modell oder eine Sprache beziehen, als auch solche, die erst aus der Gesamtkomposition entstehen. Aus diesem Grund werden die Kontextbedingungen in folgende Kategorien gegliedert:

1. **Sprachinterne Intra-Modell-Bedingungen** beziehen sich ausschließlich auf ein Modell in einer Sprache.
2. **Sprachinterne Inter-Modell-Bedingungen** beschreiben die Bedingungen, die sich aus Referenzen zwischen Modellen derselben Sprache ergeben.
3. **Sprachübergreifende Intra-Modell-Bedingungen** betrachten die Beziehungen zwischen äußerer und innerer Sprache, die durch die Spracheinbettung entstehen, ohne dass Referenzen auf andere Modelle berücksichtigt werden.
4. **Sprachübergreifende Inter-Modell-Bedingungen** ergeben sich erst durch die Kombination von Modellen unterschiedlicher Sprachen.

Diese Kategorisierung bietet neben der besseren Übersicht den Vorteil, dass sich Kontextbedingungen leichter wiederverwenden lassen. So könnten zukünftig neue Ideen für den Einsatz der UML/P entstehen, die nur einige Teilsprachen verwenden oder diese mit neuen Sprachen kombinieren (siehe dazu auch Abschnitt 7.1). Neben der Zuordnung zu einer der Kategorien besteht jede Kontextbedingung im Folgenden aus einer Beschreibung, einer eindeutigen Identifikationsnummer, einer Einteilung in schwerwiegende Fehler (Error) und Warnmeldungen (Warning) sowie ggf. zusätzlichen Anmerkungen. Dabei dient die Identifikationsnummer der Referenzierung und lässt darüber hinaus die Sprache und Kategorie der Kontextbedingung erkennen (siehe Abbildung 4.1). Die Kurzbezeichner der Sprachen sind in Tabelle 3.37 in Abschnitt 3.8 aufgeführt.

Abbildung 4.1: Aufbau der ID für Kontextbedingungen

Durch die zentrale Rolle von Klassendiagrammen in der UML/P beziehen sich viele Kontextbedingungen auf die Kompatibilität zu den dort definierten Typen. Darüber hinaus können

ebenfalls Java/P- oder native Java-Typen verwendet werden. Die Frage, wann Typen kompatibel sind, bedarf demnach einer genaueren Betrachtung. Allein der Java-Standard unterscheidet 11 Kategorien von möglichen Typkonvertierungen [GJS05], darunter auch solche wie das sogenannte Boxing und Unboxing[1], die in der UML/P nur innerhalb des eingebetteten Java-Codes eine Rolle spielen. Insbesondere da die vorliegende Fassung der UML/P auch die Nutzung anderer Sprachen zulässt, muss die Typkompatibilität unabhängig von Java betrachtet werden.

Die vorliegende Fassung der UML/P und das dazugehörige Framework wurden so entworfen, dass es für die Verwendung eines Typs nicht relevant ist, wie und in welcher Sprache dieser definiert wurde (siehe Abschnitt 7.2). Stattdessen wurde der Zugriff auf Typen standardisiert, indem davon ausgegangen wird, dass jeder Typ wie ein Java-Typ verwendet werden kann. Dies trifft automatisch auf die Nutzung von Java- und Java/P-Typen zu. Bis auf das Konzept der Assoziationen gilt dies auch für Typen aus Klassendiagrammen. Da Assoziationen jedoch in Java nicht existieren, wurde für diese ein Zugriff ähnlich zu Attributen gewählt (siehe Abschnitt 3.1). Auf diese Weise können theoretisch beliebige Sprachen zusammen mit der UML/P verwendet werden. Je nachdem, wie stark die Konzepte dieser Sprachen von dem objektorientierten Paradigma abweichen, kann es jedoch recht aufwändig sein, eine entsprechende Zuordnung zu definieren.

Dieser Ansatz ermöglicht es nun, von einem allgemeinen Typbegriff auszugehen, der von der verwendeten Sprache abstrahiert. Dadurch lässt sich folgende Definition für die Typkompatibilität aufstellen:

Definition 1 (Typkompatibilität) *Für zwei nicht primitive Typen A und B gilt: A ist genau dann kompatibel zu B, wenn A anstelle von B verwendet werden kann. Dies ist abhängig vom Kontext der Verwendung, so dass folgende Formen von Typkompatibilitäten zu unterscheiden sind:*

1. **Identität***: Typ A und B sind identisch.*
2. **Zuweisungskompatibilität***: Typ A ist zuweisungskompatibel zu B, wenn A ein Subtyp von B oder mit B identisch ist.*
3. **Castkompatibilität***: Typ A ist castkompatibel zu B, wenn A zuweisungskompatibel zu B ist oder ein zu A und B zuweisungskompatibler Typ C existiert.*

Die Castkompatibilität ergibt sich aus dem Umstand, dass z.B. der Typ einer Variablen in typisierten Sprachen im Allgemeinen nicht mit dem Typ des zugehörigen Wertes zur Laufzeit übereinstimmen muss. Entsprechend der Zuweisungskompatibilität kann hier ebenfalls die Instanz eines Subtyps verwendet werden. Der Subtyp ist dadurch allerdings nicht mehr direkt sichtbar. Wird er dennoch benötigt, ist oft wie in Java die explizite Angabe des Cast-Operators gefordert. Dieser ist als Zusicherung für die statische Analyse zu verstehen, dass zur Laufzeit nur (zuweisungskompatible) Instanzen des Subtyps vorkommen, was andernfalls zu Fehlern während

[1] Java stellt für jeden primitiven Typ eine zusätzlichen Klasse zur Verfügung, die alternativ verwendet werden kann [GJS05].

der Systemausführung führt. In typisierten Sprachen, zu denen auch die UML/P gehört, können demnach Zuweisungs- und Castkompatibilität durch die statische Analyse sicher gestellt werden, wobei bei der Castkompatibilität Laufzeitfehler nicht ausgeschlossen sind. Darüber hinaus sind diese Relationen per Definition reflexiv und transitiv.

Neben den bisher beschriebenen Typen, die auf der Definition von Interfaces, Enumerationen oder Klassen beruhen, bietet die UML/P zusätzlich sogenannte primitive Typen. Im Gegensatz zu den nicht-primitiven Typen erlauben diese keine Erzeugung von Instanzen und keine Vererbungsbeziehungen, sondern werden als reine Datentypen verwendet. Aus diesem Grund ist hierfür eine eigene Definition von Typkompatibilität erforderlich. Abbildung 4.2 stellt die Zuweisungskompatibilität der primitiven Typen in der UML/P dar, wobei der Typ `boolean` für die beiden logischen Werte `true` und `false` zu keinem der anderen genannten Typen kompatibel ist.

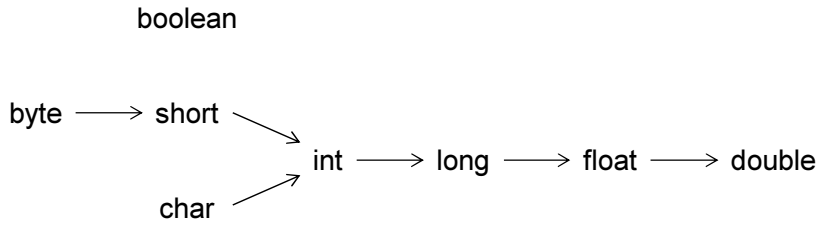


Abbildung 4.2: Zuweisungskompatibilität bei primitiven Typen in der UML/P

Die Castkompatiblität ist bei den in Abbildung 4.2 aufgeführten Typen prinzipiell in allen Kombinationen gegeben, allerdings nicht immer verlustfrei. Details wie auch mögliche Wertebereiche legt die UML/P selbst dabei nicht fest, können sich aber aus der eingebetteten Sprache oder je nach gewähltem Generator auch aus der Zielsprache der Codegenerierung ergeben.

Wird die UML/P mit anderen Sprachen kombiniert, wie dies z.B. bei eingebetteten Java-Ausdrücken der Fall ist, können diese Sprachen innerhalb der Ausdrücke wiederum eigene primitive Typen definieren. Diese müssen ähnlich den unterschiedlichen Sprachkonzepten einander zugeordnet werden, um eine konsistente Sprachkombination zu erreichen. Da Java dieselben primitiven Typen wie die UML/P verwendet, können diese einfach als gleichbedeutend angesehen werden, unabhängig, ob der Typ in einem Java-Ausdruck oder etwa für den Typ eines Attributs im Klassendiagramm verwendet wird. Dadurch lässt sich insbesondere auch das Boxing und Unboxing innerhalb der eingebetteten Java-Ausdrücke entsprechend verwenden.

Auf dieser Grundlage werden nun in den folgenden Abschnitten die Kontextbedingungen der UML/P definiert. Dabei wird auf die beiden Sprachen OCL/P und Java/P nicht im Detail eingegangen. Java/P unterscheidet sich in dieser Arbeit von gewöhnlichem Java nur durch die zusätzliche Möglichkeit, Typen aus Klassendiagrammen zu referenzieren (siehe Abschnitt 3.7). Da aber gemäß der obigen Vorgehensweise die Herkunft der Typen nicht relevant ist, können damit die Kontextbedingungen entsprechend dem Java-Standard [GJS05] vollständig übernommen werden. Ähnlich verhält es sich bei der OCL/P, bei der bis auf die Anpassung der Referenzierung keine

internen Änderungen gegenüber [Rum04a, Rum04b] vorgenommen wurden (siehe Abschnitt 3.9). Auf Besonderheiten, die sich aus der gemeinsamen Verwendung der OCL/P mit den anderen Sprachen der UML/P ergeben, wird in jeweils diesen Sprachen genauer eingegangen. Darüber hinaus gelten die allgemeinen Bedingungen in den Abschnitten 4.2.1 und 4.4.1 auch für die OCL/P.

4.2 Sprachinterne Intra-Modell-Bedingungen

4.2.1 Allgemeine Bedingungen

Die folgenden Kontextbedingungen ergeben sich aus der in Abschnitt 3.8 beschriebenen gemeinsamen Grundstruktur sowie einer einheitlichen Verwendung insbesondere von Stereotypen, Namen- und Typpreferenzen in allen Teilsprachen der UML/P. Die konkrete Syntax dieser gemeinsamen Elemente ist in den Grammatiken `Literals`, `Types` und `Common` definiert, die in allen anderen Sprachdefinitionen eingebunden sind (siehe Anhang C). Dadurch sind die Kontextbedingungen dieses Abschnitts unabhängig von der jeweiligen Teilsprache gültig.

Modifikatoren:

All-1-01: Die Sichtbarkeiten `public`, `private` und `protected` können nicht gleichzeitig auftreten.
Schweregrad: Error
Bemerkung: Laut Grammatik (siehe Quellcode C.4 in Anhang C.3) können beliebig viele Modifikatoren pro Element angegeben werden, um so eine beliebige Reihenfolge und Kombination in einer kurzen Regel zu erlauben. Obige Sichtbarkeiten schließen sich aber gegenseitig aus. Wird hingegen mehrmals derselbe Modifikator aufgeführt, stellt dies zwar einen schlechten Stil, im Grunde aber keinen Fehler dar. Da sich eine solche Mehrfachnennung nicht in der abstrakten Syntax widerspiegelt, wird dies in den Kontextbedingungen nicht weiter beachtet.

All-1-02: Der Modifikator `abstract` kann nicht zusammen mit `private`, `final` oder `static` auftreten.
Schweregrad: Error
Bemerkung: Ein abstraktes Element gibt keine Implementierungsdetails an. Diese werden stattdessen erst in konkreten Subelementen spezifiziert, was durch die zusätzlichen Modifikatoren `private` und `final` verhindert würde. Auf ein statisches Element kann hingegen direkt zugegriffen werden, so dass eine Implementierung vorhanden sein muss.

Stereotypen:

All-1-03: Ein Stereotyp kann für jedes Element nur einmal angegeben werden.

Schweregrad: Error
Bemerkung: Da Stereotypen in der hier vorgestellten Fassung der UML/P auch Name-Wert-Paare enthalten können, würden sich insbesondere unterschiedliche Wertangaben bei gleichem Namen widersprechen.

Kardinalitäten:

All-1-04: Die Untergrenze einer Kardinalität ist kleiner oder gleich der Obergrenze.
Schweregrad: Error
Bemerkung: Dabei ist zu beachten, dass der Kleene-Stern (`*`) für unbegrenzt steht und damit die größt-mögliche Zahl darstellt.

Referenzen:

All-1-05: Namen- (`Name` und `QualifiedName`) und Typreferenzen (`Type`) müssen innerhalb ihres Kontextes eindeutig auflösbar sein.
Schweregrad: Warning/Error
Bemerkung: Im Sinne der Unterspezifikation muss hier je nach Entwicklungsstand und Einsatzzweck des Modells unterschieden werden. So kann z.B. ein in der Anforderungsanalyse entworfenes Objektdiagramm Typen verwenden, für die noch keine Spezifikation existiert. Trotzdem sollte in solch einem Fall eine Warnung ausgegeben werden, um den Modellierer auf diesen Umstand hinzuweisen. Der Einsatz im Rahmen der Codegenerierung setzt hingegen eine vollständige Typdefinition voraus. Bei Typreferenzen müssen darüber hinaus Generics entsprechend den Kontextbedingungen All-1-06 bis All-1-10 beachtet werden.

Generics:

All-1-06: Der Name einer Typvariablen ist innerhalb eines Typparameters eindeutig.
Schweregrad: Error
Bemerkung: Typvariablen dürfen nicht mehrmals innerhalb desselben Kontextes definiert werden, wie z.B. `T` in `class MyClass<T,T>`. Ähnlich wie normale Variablen ist jedoch eine erneute Definition für einen Subkontext erlaubt. Ein Zugriff auf die äußere Typvariable ist dann nicht mehr möglich. So wird z.B. die Typvariable einer Klasse durch die gleichnamige Typvariable einer darin enthaltenen Methode überdeckt. Für solche Situationen kann zusätzlich eine Warnung ausgegeben werden.

All-1-07: Der referenzierte Typ einer Typvariablen-Obergrenze ist innerhalb des zugehörigen Typparameters sichtbar.
Schweregrad: Warning/Error

Bemerkung: Im Allgemeinen werden Generics wie in Java behandelt. An dieser Stelle fordert Java jedoch, dass maximal die erste Referenz auf eine Klasse und alle darauffolgenden nur noch auf Interfaces verweisen. Diese Einschränkung besteht in der UML/P durch die in Klassendiagrammen mögliche Mehrfachvererbung nicht. Eine Typvariable kann demnach auch mehrere Klassen als Obergrenze angeben, z.B. `class MyAnimal<T extends Turtle, Bird>`. Bzgl. der Gewichtung der Kontextbedingung siehe All-1-05.

All-1-08: Die referenzierten Typen eines Typarguments müssen innerhalb des Kontexts des Typarguments sichtbar sein. Dies gilt auch für die Ober- und Untergrenze eines Wildcard-Typarguments.
Schweregrad: Warning/Error
Bemerkung: z.B. `new MyAnimal<FlyingTurtle>`; vgl. All-1-07

All-1-09: Ein referenzierter Typ eines Typarguments ist zuweisungskompatibel zu den als Obergrenze angegebenen Typen der zugehörigen Typvariablen.
Schweregrad: Error
Bemerkung: Insbesondere kann für den referenzierten Typ des Typarguments Mehrfachvererbung erforderlich sein, wenn als Obergrenze mehrere Klassen angegeben sind, zwischen denen keine Vererbungsbeziehung besteht (siehe All-1-07 und Beispiel in All-1-08).

All-1-10: Der als Untergrenze eines Wildcard-Typarguments referenzierte Typ ist zuweisungskompatibel zu dem als Obergrenze angegebenen Typen der zugehörigen Typvariablen. Für die Obergrenze genügt hingegen die Castkompatibilität.
Schweregrad: Error

All-1-11: Ein generischer Typ wird mit der Angabe eines Typarguments referenziert.
Schweregrad: Warning
Bemerkung: Ähnlich wie in Java wird eine Warnung ausgegeben, sollte nur der sogenannte Raw-Type (Typname ohne Typargument) referenziert werden.

Diagramme:

All-1-12: Der Name eines Diagramms ist identisch mit dem Dateinamen ohne Dateiendung.
Schweregrad: Warning
Bemerkung: Dies wird bereits durch die in den Sprachdefinitionen angegebene Option `compilationunit` geprüft, so dass eine von MontiCore bereitgestellte Infrastruktur eine entsprechende Warnung erzeugt (siehe [Kra10]).

All-1-13: Das Paket entspricht der durch Punkte getrennten Ordnerstruktur, in der das Diagramm liegt, ausgehend vom Modellpfad.

Schweregrad: Warning
Bemerkung: siehe All-1-12.

All-1-14: Der Name eines Diagramms ist innerhalb eines Pakets eindeutig.
Schweregrad: Warning
Bemerkung: Da die Dateiendung bei der Referenzierung keine Rolle spielt, sollten auch für Diagramme unterschiedlicher Sprachen innerhalb eines Pakets nicht der gleiche Name vergeben werden.

Namen:

All-1-15: Per Konvention werden die Namen von Typvariablen und Diagrammen groß geschrieben.
Schweregrad: Warning
Bemerkung: Auch wenn es sich hier um eine Konvention handelt, wurde diese im Rahmen der Kontextbedingungen mit aufgenommen, da Abweichungen zu Missverständnissen beim Lesen der Modelle führen können.

4.2.2 Klassendiagramme

Klassendiagramme:

CD-1-01: Die Menge der Namen von Klassen, Interfaces und Enumerationen innerhalb eines Klassendiagramms sind disjunkt.
Schweregrad: Error
Bemerkung: Die Referenzierung von Typen ist unabhängig von der Typart, so dass die Eindeutigkeit der Namen innerhalb einer Typart nicht ausreichend ist (siehe auch All-1-05).

CD-1-02: Ein vollständiges Klassendiagramm enthält keine unvollständigen Typen.
Schweregrad: Warning
Bemerkung: Da ein Unvollständigkeitsindikator nur die Möglichkeit einer Unterspezifikation eines Elements ausdrückt, stellt eine solche Markierung in einem vollständigen Klassendiagramm prinzipiell keinen Widerspruch dar. Trotzdem sollte im Falle dieser expliziten Markierung eine Warnung ausgegeben werden, um Fehlinterpretationen zu vermeiden. Insbesondere ist keine zusätzliche Vollständigkeitsmarkierung für die im Klassendiagramm enthaltenen Typen gefordert.

Klassen:

CD-1-03: Die Modifikatoren `protected`, `private`, `static`, `derived` und `readonly` sind für Klassen nicht anwendbar.
Schweregrad: Error

CD-1-04: Eine Klasse kann nur Klassen erweitern (`extends`-Beziehung). Diese dürfen nicht `final` sein.
Schweregrad: Error

CD-1-05: Die angegebenen Superklassen einer Klasse sind innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: Sichtbar sind verwendete Typen dann, wenn sie im Diagramm selbst definiert oder importiert sind. Bei der Angabe von `*`-Importe kann ohne eine Betrachtung der jeweils referenzierten Pakete oder Diagramme keine Aussage getroffen werden, ob der Typ existiert (siehe auch CD-2-02). Ansonsten gilt für die Bewertung der Auflösbarkeit All-1-05.

CD-1-06: Die Vererbungsbeziehung von Klassen darf keinen Zyklus bilden.
Schweregrad: Error

CD-1-07: Eine Klasse kann nur Interfaces implementieren (`implements`-Beziehung).
Schweregrad: Error

CD-1-08: Die angegebenen Superinterfaces einer Klasse sind innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-1-05.

Interfaces:

CD-1-09: Die Modifikatoren `protected`, `private`, `final`, `static`, `derived` und `readonly` sind für Interfaces nicht anwendbar.
Schweregrad: Error

CD-1-10: Ein Interface kann nur Interfaces erweitern (`extends`-Beziehung).
Schweregrad: Error

CD-1-11: Die angegebenen Superinterfaces eines Interfaces sind innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-1-05.

CD-1-12: Die Erweiterung von Interfaces darf keinen Zyklus bilden.
Schweregrad: Error

Enumerationen:

CD-1-13: Die Modifikatoren `protected`, `private`, `final`, `abstract`, `static`, `derived` und `readonly` sind für Enumerationen nicht anwendbar.
Schweregrad: Error

CD-1-14: Eine Enumeration kann nur Interfaces implementieren (`implements`-Beziehung).
Schweregrad: Error
Bemerkung: Da die Instanzbildung von Enumerationen nicht zulässig ist und stattdessen der Zugriff auf deren Daten statisch erfolgt, können die Eigenschaften von Klassen nicht allgemeingültig übernommen werden.

CD-1-15: Die angegebenen Superinterfaces einer Enumeration sind innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-1-05.

Attribute:

CD-1-16: Der Name eines Attributs ist innerhalb einer Klasse, Enumeration oder eines Interfaces eindeutig.
Schweregrad: Error
Bemerkung: Da die Referenzierung von Attributen ähnlich wie Typen über den Namen erfolgt, ist auch hier die Eindeutigkeit gefordert.

CD-1-17: Der Typ eines Attributs ist innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-1-05.

CD-1-18: Der Modifikator `abstract` ist für Attribute nicht anwendbar.
Schweregrad: Error

CD-1-19: Die Menge der sichtbaren Attribute von Supertypen eines gemeinsamen Subtyps darf keine Attribute gleichen Namens aber castinkompatibler Typen enthalten.
Schweregrad: Error
Bemerkung: Als sichtbar werden in diesem Fall alle Attribute bezeichnet, die nicht mit dem Modifikator `private` gekennzeichnet sind. Attribute importierter Supertypen dürfen darüber hinaus nicht lokal sein. Geerbte gleichnamige Attribute können entsprechend CD-1-16 nicht in einem Subtyp vereint und insbesondere nicht initialisiert werden, wenn deren Typen castinkompatibel sind. Allerdings kann es bei vorhandenen Initialisierungen in den Superklassen zu Laufzeitfehlern kommen, so dass eine strengere Form dieser Kontextbedingung auch die Gleichheit der Attributtypen fordern könnte.

CD-1-20: Beim Überschreiben eines Attributs sind nur Typen erlaubt, die zuweisungskompatibel zum entsprechenden sichtbaren Attribut des Supertyps sind.
Schweregrad: Warning
Bemerkung: In Java ist die Benennung von Attributen unabhängig von Attributen der Superklassen möglich. Allerdings führt eine nicht zuweisungskompatible Typangabe zu seltsamen und schwer verständlichen Effekten, etwa dass eine entsprechende get-Methode für das Attribut im Subtyp nicht definiert oder dem Attribut des Supertyps kein Wert mehr zugewiesen werden kann. Durch letzteres kann es wiederum zu Laufzeitfehlern in geerbten Methoden kommen. Aus diesem Grund wird in der UML/P in solchen Fällen eine Warnung ausgegeben.

CD-1-21: Beim Überschreiben eines Attributs kann die Sichtbarkeit im Vergleich zum entsprechenden Attribut des Supertyps nicht stärker eingeschränkt sein. Ist das Attribut in mehr als einem Supertyp definiert, darf die Sichtbarkeit nicht geringer sein, als die größte Sichtbarkeit dieser Attribute.
Schweregrad: Warning
Bemerkung: Bei den Sichtbarkeiten verhält es sich ähnlich wie bei CD-1-20. Ansonsten orientiert sich die Bewertung der Sichtbarkeiten an Java.

CD-1-22: Abgesehen von finalen Attributen dürfen für Attribute in Interfaces keine Werte angegeben werden.
Schweregrad: Error
Bemerkung: Attribute können in der vorliegenden Fassung der UML/P ebenfalls als Schnittstellen verwendet werden (siehe Abschnitt 3.9). Die Implementierung findet jedoch wie bei Methoden nur in Klassen statt. Da der Inhalt der (eingebetteten) Wertzuweisung für diese Kontextbedingung keine Rolle spielt, ist eine Überprüfung unabhängig von der eingebetteten Sprache möglich.

Enumerationskonstanten:

CD-1-23: Die Namen von Enumerationskonstanten sind innerhalb einer Enumeration eindeutig.
Schweregrad: Error
Bemerkung: vgl. CD-1-16.

Konstruktoren:

CD-1-24: Die Signatur eines Konstruktors ist innerhalb der Typdefinition eindeutig, wobei der Name mit dem des Typen identisch ist.
Schweregrad: Error

CD-1-25: Die Namen der Parameter eines Konstruktors sind eindeutig.

Schweregrad: Error
Bemerkung: vgl. CD-1-16.

CD-1-26: Die Typen der Parameter eines Konstruktors sind innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-1-05.

CD-1-27: Die Exceptions eines Konstruktors sind als Typen innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: In Java wird darüber hinaus noch gefordert, dass eine Exception die vordefinierte Klasse `java.lang.Throwable` erweitert. Im Sinne der Unterspezifikation und einer flexiblen Modellierung ist etwas Vergleichbares für die UML/P nicht fest vorgegeben. Stattdessen wird eine Exception wie ein Typ behandelt, so dass etwa ein Codegenerator selbstständig entsprechende Vorgaben der Zielplattform wie bei Java hinzufügen kann. Denkbar ist auch eine Erweiterung über einen Stereotyp `<<Exception>>`, über den entsprechend verwendete Typen gekennzeichnet werden. Darüber hinaus könnten in Zukunft Modellbibliotheken ähnlich wie Java vordefinierte Exceptions zur Verfügung stellen. Bezüglich der Sichtbarkeit siehe CD-1-05.

CD-1-28: Die Modifikatoren `abstract`, `final`, `static`, `derived` und `readonly` sind für Konstruktoren nicht anwendbar.
Schweregrad: Error

Methoden:

CD-1-29: Die Signatur einer Methode ist innerhalb der Typdefinition eindeutig.
Schweregrad: Error
Bemerkung: Die Signatur orientiert sich an [GJS05] und umfasst den Methodennamen und die Parametertypen. Sind diese inklusive der Reihenfolge der Parametertypen identisch, stellt dies eine Verletzung dieser Kontextbedingung dar.

CD-1-30: Die Namen der Parameter einer Methode sind eindeutig.
Schweregrad: Error
Bemerkung: vgl. CD-1-16.

CD-1-31: Der Rückgabetyp und die Typen der Parameter einer Methode sind innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-1-05.

CD-1-32: Die Exceptions einer Methode sind als Typen innerhalb des Klassendiagramms sichtbar.
Schweregrad: Error
Bemerkung: vgl. CD-1-27.

CD-1-33: Die Modifikatoren `derived` und `readonly` sind für Methoden nicht anwendbar.
Schweregrad: Error

CD-1-34: Abstrakte Methoden können nur in einem Interface oder einer abstrakten Klasse definiert sein.
Schweregrad: Error

CD-1-35: Abstrakte Methoden und Methoden in Interfaces haben keinen Methodenrumpf.
Schweregrad: Error

CD-1-36: Die Modifikatoren `private`, `protected`, `final` und `static` sind für Methoden in Interfaces nicht anwendbar.
Schweregrad: Error

CD-1-37: Finale Methoden können nicht überschrieben werden.
Schweregrad: Error

CD-1-38: Die Menge der sichtbaren Methoden von Supertypen eines gemeinsamen Subtyps darf keine Methoden gleicher Signatur enthalten, deren Rückgabetypen untereinander nicht castkompatibel sind.
Schweregrad: Error
Bemerkung: Andernfalls ist es nicht möglich, eine Methode im Subtyp zu definieren, die sämtlichen Supertypen genügt. Bei Castkompatibilität kann hingegen ein Rückgabetyp gewählt werden, der zuweisungskompatibel zu den Rückgabetypen der geerbten Methoden ist. Die Supertypen schließen sowohl Superklassen als auch -interfaces ein. Vgl. außerdem CD-1-19 bzgl. Sichtbarkeit und CD-1-29 bzgl. Signatur.

CD-1-39: Die Menge der sichtbaren Methoden von Supertypen eines gemeinsamen Subtyps darf keine Methoden gleicher Signatur enthalten, deren Exceptions untereinander nicht castkompatibel sind.
Schweregrad: Error
Bemerkung: Insbesondere darf die Angabe einer Exception jedoch fehlen. Ansonsten können Exceptions äquivalent zum Rückgabetyp in CD-1-38 behandelt werden.

CD-1-40: Beim Überschreiben oder Implementieren einer Methode kann kein Rückgabetyp angegeben werden, der nicht zuweisungskompatibel zu dem Rückgabetyp der entsprechenden Methode der Supertypen ist.

Schweregrad: Error
Bemerkung: Ist die Methodensignatur in mehr als einem Supertyp definiert und jeweils sichtbar, muss die Zuweisungskompatibilität zum kleinsten gemeinsamen Subtyp dieser Rückgabetypen sichergestellt sein (vgl. CD-1-38).

CD-1-41: Beim Überschreiben oder Implementieren einer Methode kann keine Exception hinzugefügt werden, die nicht zuweisungskompatibel zu den Exceptions der entsprechenden Methode der Supertypen ist.
Schweregrad: Error
Bemerkung: vgl. CD-1-39.

CD-1-42: Beim Überschreiben oder Implementieren einer Methode kann die Sichtbarkeit im Vergleich zur entsprechenden Methode der Supertypen nicht stärker eingeschränkt sein.
Schweregrad: Error
Bemerkung: Ist die Methodensignatur in mehr als einem Supertyp definiert, darf die Sichtbarkeit nicht geringer sein, als die größte Sichtbarkeit dieser Signaturen (vgl. CD-1-38).

CD-1-43: Treten mehrere sichtbare Methoden gleicher Signatur in den Supertypen eines gemeinsamen Subtyps auf, müssen diese im Subtyp überschrieben werden, wenn es sich dabei nicht um eine abstrakte Klasse oder ein Interface handelt.
Schweregrad: Warning
Bemerkung: Durch die Möglichkeit der Mehrfachvererbung in der UML/P ist es bei unterschiedlicher Implementierung derselben Methode in den Supertypen nicht entscheidbar, welche Umsetzung für den Subtyp zu wählen ist. Selbst wenn eine konkrete Spezifikation der Implementierung in Form von Methodenrümpfen fehlt, kann zumindestens eine erneute Aufführung der Methodensignatur im Subtyp die Notwendigkeit einer Neuimplementierung andeuten, da anderweitig im Klassendiagramm nicht erkennbar ist, ob und wo diese umgesetzt wurde. Aus diesem Grund ist ein Warnhinweis für den Modellierer sinnvoll, sollte diese Angabe fehlen. Die Warnung kann auch ausgeschaltet werden, etwa für unterspezifizierte Modelle in der Analysephase oder wenn ein Codegenerator eine korrekte Implementierung sicher stellt.

Assoziationen:

CD-1-44: Der Modifikator `abstract` ist für Assoziationen nicht anwendbar.
Schweregrad: Error
Bemerkung: Die Modifikatoren werden für jede Assoziationsseite getrennt angegeben und dienen der Spezifikation von Zugriffsmöglichkeiten auf die jeweils assoziierten

Objekte. Dies entspricht der Verwendung von Modifikatoren zur Steuerung von Attributzugriffen, weshalb hier wie in CD-1-18 nur `abstract` ausgeschlossen ist. Bei einer abgeleiteten Assoziation, die gesondert gekennzeichnet wird (siehe Quellcode C.5 im Anhang C.4), sind automatisch beide Seiten abgeleitet.

CD-1-45: Die referenzierten Typen einer Assoziation sind innerhalb des Klassendiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-1-05.

CD-1-46: Der Zugriff auf assoziierte Objekte muss sich von Attributzugriffen der zugehörigen Objekttypen unterscheiden.
Schweregrad: Error
Bemerkung: Dies schließt geerbte Attribute und Assoziationen mit ein. Wie der Zugriff auf ein assoziiertes Objekt erfolgt, hängt davon ab, ob ein Rollen- oder Assoziationsname angegeben wurde (siehe Abschnitt 3.1). Dadurch entstehen wiederum weitere Implikationen dieser Kontextbedingung, etwa dass Rollennamen über allen von einem Typ aus navigierbaren Assoziationen eindeutig sind und sich bei fehlender Angabe eines Assoziations- oder Rollennamens nicht mit dem Namen eines assoziierten Typs überschneiden. Die Überprüfung dieser Kontextbedingung ist damit sehr komplex. Im Framework können daher solche Implikationen auch separat implementiert werden, um eine feingranularere Konfiguration der Bedingungen für verschiedene Anwendungsszenarien zu ermöglichen. So gilt z.B. beim Einsatz der OCL/P die vorliegende Bedingung sogar unabhängig von etwaigen Einschränkungen der Navigationsrichtung. Darüber hinaus ist zu beachten, dass Aggregationen und Kompositionen immer in Richtung des Kompositums navigierbar sind, auch wenn dies nicht explizit angegeben ist.

CD-1-47: Für reflexive bidirektionale Assoziationen muss mindestens ein Rollenname angegeben werden, der sich vom kleingeschriebenen Typnamen unterscheidet.
Schweregrad: Warning/Error
Bemerkung: Reflexive Assoziationen sind solche, bei denen ein Typ mit sich selbst assoziiert ist. Da in solchen Fällen die Assoziationsseiten nicht unterschieden werden können, ist bei bidirektionalen Assoziationen die Vergabe von einem Rollennamen gefordert. Dies schließt Assoziationen ohne Navigationsrichtung mit ein, nicht aber solche zwischen einem Typ und dessen Subtyp. Beim Zugriff über die OCL/P kann dies auch für unidirektionale reflexive Assoziationen gefordert werden (vgl. CD-1-46). Die Bewertung dieser Bedingung hängt darüber hinaus vom Einsatzzweck des Modells ab.

CD-1-48: Die Kardinalität auf Kompositionsseite ist standardmäßig `1` und kann alternativ nur auf `0..1` angepasst werden.

Schweregrad: Error
Bemerkung: Bei einer Komposition kann ein Objekt nur einem Kompositum zugeordnet sein.

CD-1-49: Qualifikatoren sind eindeutig als sichtbare Typreferenzen oder Referenzen auf ein Attribut auflösbar.
Schweregrad: Warning/Error
Bemerkung: Ist als Qualifikator eine Referenz auf einen Typ angegeben, muss dieser innerhalb des Klassendiagramms sichtbar sein (vgl. CD-1-05). Handelt es sich hingegen um den Namen eines Attributs, muss dieses im referenzierten Typ am gegenüberliegenden Assoziationsende vorhanden sowie für den Typ am Ausgangspunkt der Assoziation sichtbar sein (vgl. CD-1-19).

Namen:

CD-1-50: Per Konvention werden die Namen von Typen und Konstruktoren groß, sowie Methoden-, Parameter-, Attribut-, Rollen- und Assoziationsnamen klein geschrieben.
Schweregrad: Warning
Bemerkung: siehe All-1-15.

4.2.3 Objektdiagramme

Objektdiagramme besitzen einige besondere Eigenschaften, die sich nur indirekt in den Kontextbedingungen widerspiegeln, aber für deren Verständnis wichtig sind. Diese werden im Folgenden kurz erläutert.

Ähnlich wie Klassendiagramme stellen Objektdiagramme eine Abstraktion des Systems dar. Während Klassendiagramme eine abstrakte Sicht auf die Architektur modellieren, beschreiben Objektdiagramme eine konkrete Situation im System. Letztere lassen damit ebenfalls Rückschlüsse auf die Architektur zu, die jedoch nur exemplarisch ist. Hinzu kommt, dass beide Beschreibungen unvollständig sein können. In [Rum04a] wird deshalb bei dem Vergleich der modellierten Strukturen zwischen der *Widerspruchsfreiheit* eines Objektdiagramms gegenüber einem unvollständigen und der *Konformität* gegenüber einem vollständigen Klassendiagramm unterschieden. Widerspruchsfrei ist demnach ein Objektdiagramm auch dann, wenn es zusätzliche Typen, Attribute und Links enthält, die im Klassendiagramm nicht dargestellt sind. In konformen Objektdiagrammen sind diese zusätzlichen Strukturen nicht erlaubt, wobei in dieser Arbeit davon ausgegangen wird, dass diese nicht in einem einzigen Klassendiagramm enthalten sein müssen, sondern sich auf verschiedene Klassendiagramme und auch andere Sprachen wie Java verteilen können. Ein Objektdiagramm ist demnach konform zu einer Menge von Spezifikationen, wobei es zu jeder einzelnen nur widerspruchsfrei sein kann.

Daraus können für die in Objektdiagrammen modellierten prototypischen Objektstrukturen folgende Beobachtungen abgeleitet werden:

- **Verwendung von Supertypen:** Im Sinne der Abstraktion stellen Typangaben von Objekten und Attributen in Objektdiagrammen eine Art Obergrenze für die möglichen Typen im System dar. Es können somit auch Supertypen angegeben sein, selbst wenn durch die Architekturspezifikation bedingt nur Subtypen im System vorkommen können.
- **Auslassung von Details:** Für die Widerspruchsfreiheit oder Konformität zu einer Architekturspezifikation müssen nicht sämtliche Details im Objektdiagramm wiederholt werden. So können insbesondere Typen, Attribute und Modifikatoren fehlen.
- **Angabe spezifischer Eigenschaften:** Spezielle Eigenschaften in Form von Stereotypen müssen sich nicht mit der Architekturspezifikation decken, sondern beziehen sich nur auf die dargestellte prototypische Objektstruktur.

Darüber hinaus sind Einschränkungen einer Architekturspezifikation für Objektdiagramme insofern nicht bindend, als dass Attribute und Assoziationen der Objekte unabhängig von deren Sichtbarkeit im Klassendiagramm angegeben werden können. Dabei ist insbesondere auch ein Zugriff auf private und lokale Attribute und Methoden des jeweiligen Objekts bei der Spezifikation von Attributwerten möglich. Nur auf diese Weise lassen sich vollständige Objektstrukturen modellieren.

Objektdiagramme:

OD-1-01: Ein vollständiges Objektdiagramm enthält keine unvollständigen Objekte.
Schweregrad: Warning
Bemerkung: vgl. CD-1-02.

Objekte:

OD-1-02: Der Name eines Objekts ist innerhalb eines Diagramms eindeutig.
Schweregrad: Error
Bemerkung: siehe auch OD-1-03.

OD-1-03: Der Name eines Objekts darf sich nicht mit den Namen verwendeter Typen innerhalb des Diagramms überschneiden.
Schweregrad: Error
Bemerkung: Bei unbenannten Objekten dient ggf. der Typname als Referenz, so dass ein gleich gewählter Name zu Fehlinterpretationen führen kann.

OD-1-04: Die Modifikatoren `protected`, `private`, `abstract`, `static`, `derived` und `read-only` sind für Objekte nicht anwendbar.
Schweregrad: Error
Bemerkung: vgl. CD-1-03.

OD-1-05: Zwei Objekte gleichen Typs haben keine Attribute gleichen Namens aber castinkompatibler Typangabe.

Schweregrad: Error
Bemerkung: Ohne die konkrete Typdefinition der Objekte zu betrachten, ist mindestens Castkompatibilität bei gleichnamigen Attributen notwendig (siehe OD-4-06).

OD-1-06: Für vollständige Objekte ist ein Typ angegeben.
Schweregrad: Error

Attribute:

OD-1-07: Der Name eines Attributs ist innerhalb eines Objektes eindeutig.
Schweregrad: Error
Bemerkung: ergibt sich aus CD-1-16.

OD-1-08: Die Menge der Attributnamen eines Objekts ist Teilmenge der Attributnamen eines anderen Objekts gleichen Typs, falls letzteres vollständig ist.
Schweregrad: Error
Bemerkung: Da für vollständige Objekte alle Attribute des zugehörigen Typs angegeben sind, können unvollständige Objekte keine zusätzlichen Attribute enthalten (vgl. OD-4-04).

OD-1-09: Bei vollständigen Objektdiagrammen ist die Menge der Attributnamen zweier Objekte gleichen Typs identisch.
Schweregrad: Error
Bemerkung: ergibt sich aus OD-1-08.

OD-1-10: Der Modifikator `abstract` ist für Attribute nicht anwendbar.
Schweregrad: Error
Bemerkung: ergibt sich aus CD-1-18.

Links:

OD-1-11: Der Modifikator `abstract` ist für Links nicht anwendbar.
Schweregrad: Error
Bemerkung: ergibt sich aus CD-1-44.

OD-1-12: Eine Objektreferenz eines Links besteht entweder aus dem Objektnamen oder dem Namen des Objekttyps. Dieser Name muss sich genau einem Objekt zuordnen lassen.
Schweregrad: Error
Bemerkung: Im Vergleich zu OD-1-03 stellt eine nicht eindeutig auflösbare Referenz einen Fehler dar. Allerdings können Objekte ebenfalls importiert werden (siehe OD-2-01), so dass sich die Konsistenz von Referenzen erst durch die Betrachtung der Importe vollständig überprüfen lässt (vgl. CD-1-05).

OD-1-13: Die referenzierten Objekte eines Links sind innerhalb des Objektdiagramms sichtbar.
Schweregrad: Error
Bemerkung: Sichtbarkeit bedeutet in diesem Fall, dass ein Objekt entweder im Diagramm selbst definiert oder importiert ist. Dabei ist auch OD-2-02 zu beachten.

OD-1-14: Der Rollenname am Ende eines navigierbaren Links darf in den referenzierten Objekten am Ausgangspunkt nicht als Name eines Attributs vorkommen, wenn der Link oder das entsprechende Linkende nicht abgeleitet ist.
Schweregrad: Error
Bemerkung: Da Links nicht alle Angaben der entsprechenden Assoziation enthalten müssen, kann bei fehlenden Rollennamen keine weitere Aussage entsprechend CD-1-46 getroffen werden. Ein Link wird auch dann als navigierbar angesehen, wenn keine konkrete Navigationsrichtung angegeben ist.

OD-1-15: Ein Objekt wird höchstens einmal auf der rechten Seite einer Komposition referenziert.
Schweregrad: Error

Namen:

OD-1-16: Per Konvention werden Objekt-, Attribut-, Rollen- und Linknamen klein geschrieben.
Schweregrad: Warning
Bemerkung: siehe All-1-15.

4.2.4 Statecharts

Statecharts:

SC-1-01: Ein vollständiges Statechart enthält keine unvollständigen Zustände.
Schweregrad: Warning
Bemerkung: vgl. CD-1-02.

SC-1-02: Ein vollständiges Statecharts enthält mindestens einen Startzustand auf der obersten Hierarchieebene.
Schweregrad: Warning
Bemerkung: Innere Startzustände definieren nur den Einstieg beim Betreten des übergeordneten Zustands. Daher sollte auch ein Startzustand für das Statechart selbst angegeben werden. Enden muss ein Statechart hingegen nicht notwendigerweise. Siehe auch CD-1-02.

SC-1-03: Ein vollständiges Methodenstatechart enthält genau einen Startzustand und mindestens einen Finalzustand auf der obersten Hierarchieebene.
Schweregrad: Warning
Bemerkung: vgl. SC-1-02.

Zustände:

SC-1-04: Der Name eines Zustands ist innerhalb eines Diagramms bzw. des übergeordneten Zustands eindeutig.
Schweregrad: Error
Bemerkung: Da Zustände ebenfalls über ihren voll-qualifizierten Namen referenziert werden können, genügt die Eindeutigkeit des Namens pro Hierarchieebene (siehe Abschnitt 3.3).

SC-1-05: Ein vollständiger Zustand enthält keine unvollständigen Subzustände.
Schweregrad: Warning
Bemerkung: Dies gilt auch, falls für das zugehörige Statechart die Vollständigkeit zugesichert ist (ergibt sich aus SC-1-01).

Transitionen:

SC-1-06: Der Quell- und Zielzustand einer Transition sind innerhalb des Statecharts definiert.
Schweregrad: Error
Bemerkung: Es sei denn, die voll-qualifizierten Namen werden ebenfalls für die Komposition von Statecharts eingesetzt (siehe Abschnitt 3.3).

SC-1-07: Eine Transition kann einen lokalen Zustand nur dann referenzieren, wenn dieser im selben Zustand wie die Transition oder in einem äußeren Zustand davon definiert ist.
Schweregrad: Error
Bemerkung: Dabei werden die inneren Zustände eines lokalen Zustands ebenfalls als lokal betrachtet.

SC-1-08: Stimuli, die für den Aufruf einer Methode oder einer Exception stehen, sind in Methodenstatecharts nicht zulässig.
Schweregrad: Error
Bemerkung: Da das Verhalten von Methoden modelliert wird, können nur solche Ereignisse erwartet werden, die innerhalb der Methode angestoßen wurden. Demnach sind nur Stimuli in Form von spontanen Transitionen oder Returns als Reaktion auf eine Anweisung einer vorhergehenden Transition zulässig, nicht aber Methoden-

oder Exceptionaufrufe [Rum04a]. Die Überprüfung dieser Kontextbedingung ist innerhalb des Statecharts möglich, da sich Transitionen mit diesen Stimuli syntaktisch von anderen Transitionen unterscheiden (siehe Abbildung C.7 im Anhang C.6).

Im Folgenden werden einige optionale Kontextbedingungen definiert, die sich nur auf deterministische Statecharts (DSC) beziehen. Da Determinismus eine häufige Anforderung von Codegeneratoren an Statecharts ist, werden die Bedingungen hier aufgenommen, obwohl diese in der UML/P nicht allgemein gefordert sind. Allerdings ist dabei zu beachten, dass die Kontextbedingungen notwendig aber nicht hinreichend sind, um den Determinismus eines Statecharts nachzuweisen. Insbesondere ist für ein deterministisches Statechart gefordert, dass in jedem Zustand immer nur eine ausgehende Transition schaltbereit ist. Dies ist jedoch im Allgemeinen nicht automatisiert überprüfbar, da Vorbedingungen und Argumente von Stimuli aus beliebig komplexen Ausdrücken bestehen können, deren Werte häufig erst zur Laufzeit des Systems feststehen (siehe auch All-3-03).

SC-1-09: optional für DSC: Es ist genau ein initialer Zustand definiert.
Schweregrad: Warning/Error
Bemerkung: Diese Kontextbedingung wird nur angewendet, wenn deterministische Statecharts gefordert sind. Die Bewertung hängt vom Entwicklungsstand bzw. Einsatzzweck des Modells ab. Stellt dies für Codegeneratoren häufig einen Fehlerfall dar, genügt in der Anforderungsanalyse im Allgemeinen eine Warnmeldung, um den Modellierer darauf hinzuweisen, dass dieser Punkt in einer späteren Iteration überarbeitet werden muss.

SC-1-10: optional für DSC: Hat ein Zustand eine spontane ausgehende Transition, so hat er keine weiteren ausgehenden Transitionen.
Schweregrad: Warning/Error
Bemerkung: Dabei ist zu beachten, dass in einer Zustandshierarchie auch ausgehende Transitionen von Superzuständen für den vorliegenden Zustand schaltbereit sein können [Rum04a]. Aus diesem Grund überträgt sich das Verbot einer ausgehenden Transition iterativ jeweils auch auf den Superzustand, sofern der betrachtete innere Zustand final ist oder auf dessen Hierarchieebene keine finalen Zustände existieren. Bezüglich Schweregrad siehe SC-1-09.

Namen:

SC-1-11: Per Konvention werden Namen von Zuständen groß und von Parametern klein geschrieben.
Schweregrad: Warning
Bemerkung: siehe All-1-15.

4.2.5 Sequenzdiagramme

Sequenzdiagramme:

SD-1-01: In einem vollständigen Sequenzdiagramm sind keine Objekte und damit deren Interaktionen als unvollständig markiert.
Schweregrad: Warning
Bemerkung: Dies bezieht sich auf die Markierung des Diagramms und der Objekte mit den Repräsentationsindikatoren (vgl. CD-1-02), wobei ebenfalls die äquivalente Bedeutung der Stereotypen `<<match_complete>>` und `<<match_visible>>` zu beachten ist (siehe Abschnitt 3.8).

Objekte:

SD-1-02: Der Name eines Objekts ist innerhalb eines Diagramms eindeutig.
Schweregrad: Error
Bemerkung: vgl. OD-1-02.

SD-1-03: Der Name eines Objekts überschneidet sich nicht mit den Namen verwendeter Typen innerhalb des Diagramms.
Schweregrad: Warning
Bemerkung: vgl. OD-1-03.

SD-1-04: Die Modifikatoren `protected`, `private`, `abstract`, `static`, `derived` und `read-only` sind für Objekte nicht anwendbar.
Schweregrad: Error
Bemerkung: vgl. OD-1-04.

Interaktionen:

SD-1-05: Eine Objektreferenz einer Interaktion besteht entweder aus dem Objektnamen oder dem Namen des Objekttyps. Dieser Name muss sich genau einem Objekt zuordnen lassen.
Schweregrad: Error
Bemerkung: siehe OD-1-12.

SD-1-06: Die referenzierten Objekte einer Interaktion sind innerhalb des Sequenzdiagramms sichtbar.
Schweregrad: Error
Bemerkung: siehe OD-1-13.

Aktivitäten:

SD-1-07: Der Name eines Aktivitätsstarts ist innerhalb der Aktivitäten eindeutig.
Schweregrad: Error
Bemerkung: Für die entsprechende Endmarkierung darf sich der Name nur einer Aktivität zuordnen lassen (siehe auch SD-1-09).

SD-1-08: Die Objektreferenz eines Aktivitätsstarts besteht entweder aus dem Objektnamen oder dem Namen des Objekttyps. Dieser Name muss sich genau einem Objekt zuordnen lassen.
Schweregrad: Error
Bemerkung: analog zu SD-1-05.

SD-1-09: Vor einem Aktivitätsende existiert für jeden dort aufgeführten Namen genau ein Aktivitätsstart mit diesem Namen im Sequenzdiagramm.
Schweregrad: Error
Bemerkung: Da die Komposition von Sequenzdiagrammen nicht vorgesehen ist, können Aktivitäten nur innerhalb eines Sequenzdiagramms gestartet werden.

SD-1-10: Jeder Name eines Aktivitätsstarts wird höchstens in einem Aktivitätsende referenziert.
Schweregrad: Error
Bemerkung: Das mehrmalige Beenden einer Aktivität stellt einen Fehlerfall dar. Das Ende kann hingegen offen gelassen werden.

SD-1-11: Ist ein Sequenzdiagramm vollständig, wird jeder Name eines Aktivitätsstarts in genau einem Aktivitätsende referenziert. Dies gilt auch für Aktivitäten von den Objekten, deren Interaktionen vollständig angegeben sind.
Schweregrad: Warning
Bemerkung: Aktivitäten sollten für vollständige Interaktionen abgeschlossen werden (siehe auch SD-1-01).

SD-1-12: Ist ein Sequenzdiagramm vollständig, entspricht das Zielobjekt eines Aktivitätsstarts dem Quellobjekt der letzten Interaktion innerhalb dieser Aktivität. Dies gilt auch für Aktivitäten von den Objekten, deren Interaktionen vollständig angegeben sind.
Schweregrad: Warning
Bemerkung: Eine Aktivität wird im Allgemeinen durch ein Return an das Objekt abgeschlossen, das die Aktivität gestartet hat.

SD-1-13: Ist ein Sequenzdiagramm vollständig, entspricht das Quellobjekt eines Aktivitätsstarts dem Zielobjekt der letzten Interaktion innerhalb dieser Aktivität. Dies gilt auch für Aktivitäten von den Objekten, deren Interaktionen vollständig angegeben sind.

Schweregrad: Warning
Bemerkung: vgl. SD-1-12.

SD-1-14: Ist ein Sequenzdiagramm vollständig, ist die erste Interaktion einer Aktivität ein Methodenaufruf oder eine Objektinstanziierung und die letzte Interaktion ein `return`-Aufruf oder eine Exception. Dies gilt auch für Aktivitäten von den Objekten, deren Interaktionen vollständig angegeben sind.
Schweregrad: Warning
Bemerkung: vgl. SD-1-12.

Bedingungen:

SD-1-15: Der Kontext einer Bedingung referenziert entweder Objektnamen oder Namen von Objekttypen. Jeder Name muss sich genau einem Objekt zuordnen lassen.
Schweregrad: Error
Bemerkung: äquivalent zu SD-1-05.

Namen:

SD-1-16: Per Konvention werden Objektnamen klein geschrieben.
Schweregrad: Warning
Bemerkung: siehe All-1-1.

4.2.6 Testspezifikationssprache

Unit-Tests:

TC-1-01: Der Name eines Unit-Tests ist innerhalb eines Testfalls eindeutig.
Schweregrad: Error

Namen:

TC-1-02: Per Konvention werden Namen von Unit-Tests groß geschrieben.
Schweregrad: Warning
Bemerkung: siehe All-1-15.

4.3 Sprachinterne Inter-Modell-Bedingungen

Die Beziehung zwischen Modellen derselben Sprache werden in [Rum04a] nur am Rande behandelt. So werden zwar mehrere Objektdiagramme in einer OCL-Bedingung oder einem Testfall kombiniert, die Spezifikation einer Klasse auf verschiedene Sichten aufgeteilt oder Methoden- und

Objektverhalten mit Statecharts durch die Zuordnung zu einer Klasse vereint. Diese Verbindungen bestehen aber oft nur implizit aufgrund gleicher Namen innerhalb der Modelle bzw. Referenzen auf eine gemeinsame Klasse oder werden in anderen Sprachen etwa zur Testfallspezifikation festgelegt. Innerhalb eines Modells ist in der graphischen Darstellung hingegen keine Möglichkeit vorgesehen, explizit auf andere Modelle derselben Sprache oder deren Elemente zuzugreifen bzw. diese zu referenzieren. Stattdessen wird in [Rum04a] und [Rum04b] von einem globalen Namensraum ausgegangen, bei dem die einzelnen Modelle Sichten auf das System darstellen, die sich in einem vollständigen Produktmodell vereinen. Das Produktmodell ergibt sich demnach durch die Vereinigung aller Modelle, wobei aufgrund des globalen Namenskonzeptes gleich benannte Elemente derselben Elementart zu einem Element im Produktmodell vereint werden. Diese Vereinigung muss dann wiederum den sprachinternen Intra-Modell-Bedingungen aus Abschnitt 4.2 genügen. Inwieweit dieses Produktmodell in einem Werkzeug der UML/P konkret vorliegt und etwa zur Codegenerierung genutzt werden kann, oder nur ein theoretisches Konstrukt darstellt, wurde hingegen nicht festgelegt. Beide Möglichkeiten haben für die werkzeuggestützte Verarbeitung der Modelle Vor- und Nachteile, die in [Rum04a] ausführlich diskutiert wurden.

Bei der in dieser Arbeit entwickelten Werkzeuginfrastruktur für die UML/P wurde der Fokus auf die Agilität des Entwicklungsprozesses gelegt. Hierfür ist die Verwaltung eines Produktmodells als Ausgangspunkt für die Überprüfung von Kontextbedingungen sowie als Eingabe für die Codegenerierung von Nachteil, da in beiden Fällen immer die Gesamtheit der Modelle betrachtet wird, auch wenn nur ein kleiner Teil von Änderungen betroffen ist. Deshalb wurde bei der Entwicklung der Infrastruktur darauf geachtet, dass sich jedes Modell einzeln verarbeiten lässt. Allerdings entstehen bei diesem Vorgehen vor allem durch das globale Namenskonzept der UML/P andere Probleme. So ist nun bei der Betrachtung eines Modells nicht mehr bekannt, welche anderen Modelle noch Details zu den darin modellierten Strukturen hinzufügen. Dies ist insbesondere ein Problem bei den Klassendiagrammen, die durch zusätzliche Struktur-, Bedingungs- und Verhaltensspezifikationen durch weitere Modelle ergänzt werden können. Bei den anderen Modellarten ist dies nicht der Fall, so dass diese in sich auf Konsistenz geprüft werden können. Dies liegt an der Art und Weise, wie die Modelle in der UML/P eingesetzt werden:

- *Statecharts* kapseln immer das vollständige Verhalten eines Objekts oder einer Methode und können auf dieser Ebene nicht durch andere Modelle ergänzt werden.
- *Objekt-* und *Sequenzdiagramme* werden einzeln auf das System abgebildet. Inwieweit Namensgleichheit zwischen den Diagrammen eine Rolle spielt, hängt vom Verwendungszweck ab und kann deshalb nicht allgemein überprüft werden.
- Modellierte Testfälle der *Testspezifikationssprache* und *OCL/P*-Bedingungen sind jeweils in sich abgeschlossen.

Methoden- und Objektverhalten bei Statecharts und verschiedene OCL/P-Bedingungen können sich zwar widersprechen. Der allgemeine Nachweis solcher Fehler zählt jedoch laut des

Gödelschen Unvollständigkeitssatzes [Göd31] zu den nicht-berechenbaren Problemen. In diesem Fall ist also der Einsatz von Kontextbedingungen nicht möglich. Stattdessen können Testfälle zur Vermeidung von Fehlern beitragen.

Ohne ein vollständiges Produktmodell ist die Überprüfung von Kontextbedingungen zwischen Klassendiagrammen unter Beibehaltung des globalen Namenskonzeptes nur durch die Verarbeitung aller in einem Projekt vorliegenden Klassendiagramme möglich. Die Bedingungen würden in diesem Fall im Wesentlichen den in Abschnitt 4.2 aufgeführten sprachinternen Intra-Modell-Bedingungen entsprechen. Allerdings kommt die Verarbeitung aller Klassendiagramme bezüglich Aufwand und Nachteilen der Bildung des Produktmodells gleich. Abgesehen von der Verarbeitung bringt das globale Namenskonzept auch für das Verständnis der modellierten Architektur einen gewissen Nachteil mit sich, da die Betrachtung des vollständigen Aufbaus einer Klasse potentiell ebenfalls alle Diagramme erfordert. Um dieses Problem zu umgehen, wurde in dieser Arbeit das Sichtenkonzept bei Klassendiagrammen wie folgt eingeschränkt:

- Importierte Typen können nur verwendet, nicht aber in ihrer Zusammensetzung verändert werden. Erlaubt ist hingegen eine Erweiterung von Typen im Rahmen des Vererbungskonzeptes.
- Das globale Namenskonzept wird nicht verwendet. Typen sind demnach in einem Klassendiagramm entweder definiert oder importiert. Gleich benannte Typ-Definitionen in verschiedenen Klassendiagrammen werden demnach nicht als zwei Sichten desselben Typs angesehen.

Auf diese Weise entsteht genau eine Definitionsstelle für jeden Typ, die in den Modellen eindeutig mit Hilfe von Importen oder voll-qualifizierten Namen referenziert werden kann. Wie die anderen Modellarten können damit auch Klassendiagramme als in sich abgeschlossen angesehen werden. Eine weniger strikte Variante dieser Einschränkungen, die dem Sichtenkonzept der ursprünglichen UML/P näher kommt, erlaubt die erneute Aufführung ähnlich einer Definition auch von importierten Typen. In diesem Fall wird jedoch von einer Sicht auf den entsprechenden Typ ausgegangen, so dass Details ausgelassen, aber im Vergleich zur eigentlichen Definitionsstelle keine hinzugefügt werden können. Die in Teilen redundante Darstellung kann jederzeit gegenüber der Definitionsstelle durch einen einfachen Vergleich verifiziert werden.

Wie die obigen Erläuterungen zeigen, ergeben sich je nach gewählter Variante unterschiedliche Bedingungen. Auch das globale Namenskonzept ist mit den genannten Problemen weiterhin denkbar. Im Folgenden werden deshalb nur die allgemein gültigen Kontextbedingungen aufgeführt und die Auswahl des Sichtenkonzepts als semantischer Variationspunkt offen gelassen (siehe dazu auch Abschnitt 6.2). Um ebenfalls Objektdefinitionen wieder verwenden zu können, wurde zusätzlich bei Objektdiagrammen der Import von Objekten erlaubt. Die semantische Varianz kann hier entsprechend dem diskutierten Typimport bei Klassendiagrammen gesehen werden.

Auch für die agile Verarbeitung der Modelle im Rahmen der Codegenerierung ist die Aufhebung des globalen Namenskonzepts von Vorteil. Die Einschränkungen der strikten oder weniger

strikten Variante reichen dabei aus, um trotz separater Verarbeitung der Modelle ein vollständiges System zu generieren. Entsprechende Konzepte werden in Kapitel 6 vorgestellt.

4.3.1 Klassendiagramme

Klassendiagramme:

CD-2-01: Die Importe in Klassendiagrammen können Typen und Assoziationen anderer Klassendiagramme referenzieren. ∗-Importe beziehen sich auf Pakete oder Klassendiagramme. Die referenzierten Elemente müssen jeweils sichtbar sein.
Schweregrad: Error
Bemerkung: Bzgl. Sichtbarkeit siehe CD-2-02.

CD-2-02: Lokale Elemente sind außerhalb des Klassendiagramms nicht sichtbar.
Schweregrad: Error
Bemerkung: Insbesondere können lokale Typen und Assoziationen nicht in anderen Diagrammen importiert werden. Die Bewertung der Auflösbarkeit entspricht dabei All-1-05. Sollte sich jedoch bei der Auflösung herausstellen, dass es sich um ein importiertes lokales Element handelt, stellt dies den hier beschriebenen Fehlerfall dar. Lokale Attribute und Methoden werden außerhalb des Klassendiagramms wie private behandelt (siehe auch Tabelle 3.10).

4.3.2 Objektdiagramme

Objektdiagramme:

OD-2-01: Die Importe in Objektdiagrammen können Objekte anderer Objektdiagramme referenzieren. ∗-Importe beziehen sich sprachintern auf Pakete oder Objektdiagramme. Die referenzierten Elemente müssen jeweils sichtbar sein.
Schweregrad: Error
Bemerkung: Links werden als Eigenschaft der Objekte importiert, sofern das verlinkte Objekt ebenfalls importiert wurde. Bzgl. Sichtbarkeit siehe OD-2-02.

OD-2-02: Lokale Elemente oder anonyme Objekte sind außerhalb des Objektdiagramms nicht sichtbar.
Schweregrad: Error
Bemerkung: vgl. CD-2-01.

4.4 Sprachübergreifende Intra-Modell-Bedingungen

Die sprachübergreifenden Intra-Modell-Bedingungen beziehen sich auf die in den Diagrammen eingebetteten Sprachen. Dies sind zurzeit Java und OCL/P. Das UML/P-Framework wurde

jedoch so entworfen, dass diese Sprachen austauschbar sind (siehe Kapitel 7). Darüber hinaus kann der Inhalt von Invarianten in verschiedenen Sprachen angegeben werden (siehe All-3-02). Da die Sprache somit nicht feststeht, beziehen sich die Kontextbedingungen im Folgenden auf allgemeine Anforderungen, die jeweils an der Stelle der Spracheinbettung erfüllt sein müssen. Gleichzeitig gelten die internen Bedingungen der jeweils eingebetteten Sprache.

4.4.1 Allgemeine Bedingungen

Invarianten:

All-3-01: In der Bedingung einer Invariante sind nur Modellelemente referenzierbar, die innerhalb des Modells bzw. des umschließenden Elements sichtbar sind.
Schweregrad: Error
Bemerkung: Der Zugriff einer Invariante unterscheidet sich demnach zunächst nicht von den Möglichkeiten des umschließenden Modells bzw. Elements. Allerdings kann je nach den Eigenschaften der für die Invariante genutzten Sprache hiervon abgewichen werden. So ist die OCL/P nicht an Sichtbarkeiten bei der Vererbung oder Navigationsrichtungen bei Assoziationen gebunden (siehe Abschnitt 3.6). Auch der Zugriff auf lokale Elemente ist erlaubt. Bei der Verwendung der OCL/P muss demnach ein referenziertes Element nur auflösbar sein.

All-3-02: Ist mehr als eine Sprache für das Ausdrücken von Bedingungen vorgesehen, wird die Sprache eindeutig über den Parameter `kind` für eine Invariante festgelegt.
Schweregrad: Error
Bemerkung: Der Parameter, der einer Invariante vorangestellt wird, ermöglicht es, Invarianten in verschiedenen Sprachen zu spezifizieren (siehe Abschnitt 3.9.2).

All-3-03: Eine Invariante lässt sich einem Wert des primitiven Typs `boolean` zuordnen.
Schweregrad: Error
Bemerkung: Da sich eine Invariante aus komplexen Ausdrücken zusammensetzen kann, ist diese Kontextbedingung nicht in allen Fällen überprüfbar. Dies hängt mit dem aus der Theoretischen Informatik bekannten Halteproblem zusammen, das besagt, dass die Terminierung von Berechnungsvorschriften nicht generell nachweisbar ist. Dennoch ist eine statische Analyse des erwarteten Typs eines Ausdrucks möglich, die allerdings keine Laufzeitfehler ausschließt.

4.4.2 Klassendiagramme

Attribute:

CD-3-01: Die Initialisierung eines Attributs ist zuweisungskompatibel zum Attributtyp.
Schweregrad: Error

CD-3-02: Die Initialisierung eines Attributs greift nur auf Methoden, Konstruktoren, Attribute und assoziierte Objekte zu, die in dem Typ sichtbar sind, in dem das Attribut definiert wurde. Typreferenzen müssen darüber hinaus im zugehörigen Klassendiagramm sichtbar sein.

Schweregrad: Warning/Error

Bemerkung: Die obige Bedingung sagt nichts über die Reihenfolge der Attribut-Initialisierungen aus, die sich bei Referenzen auf andere Attribute oder assoziierte Objekte ergeben können. Dies liegt daran, dass die Klassendiagramme in erster Hinsicht als Spezifikation anzusehen sind. So kann ein intelligenter Codegenerator unter Umständen die notwendige Reihenfolge automatisch bestimmen. Alternativ können die Kontextbedingungen entsprechend erweitert oder Konstruktoren verwendet werden, um eine Reihenfolge sicher zu stellen. Daher handelt es sich hier um einen Variationspunkt, der von der UML/P selbst nicht direkt vorgeschrieben wird. Siehe außerdem All-1-05 bzgl. Bewertung und CD-1-19 bzgl. Sichtbarkeit.

CD-3-03: Finale Attribute müssen im Konstruktor oder durch Angabe eines Wertes an der Definitionsstelle initialisiert werden.

Schweregrad: Warning/Error

Bemerkung: Entsprechend All-1-05 kann bei unterspezifizierten Modellen eine Initialisierung noch offen gelassen werden, so dass eine Warnung für den Modellierer hinreichend ist. Als Eingabe für Codegeneratoren stellt dies jedoch einen Fehlerfall dar, da finalen Attributen auf anderem Wege kein Wert mehr zugewiesen werden kann.

CD-3-04: Für abgeleitete Attribute muss eine Ableitungsregel angegeben werden. Darüber hinaus ist für das Attribut eine Wertzuweisung nicht zulässig.

Schweregrad: Warning/Error

Bemerkung: Die Ableitungsregel kann sowohl als Attributwert als auch über eine zusätzliche Invariante angegeben werden. Diese Regel wird beim Abruf des Attributs ausgeführt. Die Bewertung des Schweregrades hängt vom Einsatzzweck des Modells ab (vgl. CD-3-03).

CD-3-05: Treten mehrere sichtbare Attribute gleichen Namens in den Supertypen eines gemeinsamen Subtyps auf, müssen diese im Subtyp überschrieben werden, wenn in mindestens einem der Supertypen das entsprechende Attribut initialisiert wurde.

Schweregrad: Warning/Error

Bemerkung: Andernfalls ist es nicht entscheidbar, welche Form der Initialisierung im Subtyp gilt (vgl. CD-1-43).

Enumerationskonstanten:

CD-3-06: Die Werte von Enumerationsparametern sind zuweisungskompatibel zu den Parametern eines Konstruktors.
Schweregrad: Error
Bemerkung: Wie in Java können die Konstanten von Enumerationen eigene Werte in Form von Attributen kapseln, die über die Konstruktoren der jeweiligen Enumeration initialisiert werden können [GJS05].

Konstruktoren:

CD-3-07: Der Rumpf eines Konstruktors greift nur auf Methoden, Konstruktoren, Attribute und assoziierte Objekte zu, die in dem Typ sichtbar sind, in dem der Konstruktor definiert wurde. Darüber hinaus können Parameter des Konstruktors über deren Namen referenziert werden. Typreferenzen müssen hingegen im zugehörigen Klassendiagramm sichtbar sein.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-3-02.

Methoden:

CD-3-08: Der Rumpf einer Methode greift nur auf Methoden, Konstruktoren, Attribute und assoziierte Objekte zu, die in dem Typ sichtbar sind, in dem die Methode definiert wurde. Darüber hinaus können Parameter der Methode über deren Namen referenziert werden. Typreferenzen müssen hingegen im zugehörigen Klassendiagramm sichtbar sein.
Schweregrad: Warning/Error
Bemerkung: vgl. CD-3-02.

CD-3-09: Alle Rückgabetypen eines Methodenrumpfes müssen zuweisungskompatibel zum Rückgabetyp der Methode sein.
Schweregrad: Error
Bemerkung: Wie ein Rückgabetyp eines Methodenrumpfes definiert wird, hängt von der eingebetteten Sprache ab. Im Falle von Java wird dieser durch die Typen der `return`-Ausdrücke bestimmt.

4.4.3 Objektdiagramme

Attribute:

OD-3-01: Der Wert eines Attributs muss zuweisungskompatibel zum Attributtyp sein.

Schweregrad: Error
Bemerkung: entspricht CD-3-01.

OD-3-02: Die Wertangabe eines Attributs in Form eines komplexen Ausdrucks greift nur auf Attribute und verlinkte Objekte zu, die in dem Objekt sichtbar sind, in dem das Attribut definiert wurde. Referenzierte Typen und Objekte müssen darüber hinaus im zugehörigen Objektdiagramm sichtbar sein.
Schweregrad: Warning/Error
Bemerkung: Ob ebenfalls mögliche Methoden- und Konstruktoraufrufe innerhalb des Objekts zulässig sind, lässt sich nur anhand des Objektdiagramms nicht überprüfen. Auch Attribute müssen in den Objekten nicht vollständig aufgeführt sein. Daher ist eine umfassende Konsistenzprüfung des Ausdrucks erst durch Hinzunahme der Typdefinitionen etwa innerhalb von Klassendiagrammen möglich (siehe OD-4-09). Bzgl. Objekt- und Typreferenzen siehe auch OD-1-12 und All-1-05.

4.4.4 Statecharts

Aktionen und Zustandsinvarianten:

SC-3-01: In entry-/exit-Aktionen, do-Aktivitäten oder Zustandsinvarianten kann nur auf lokale Variablen zugegriffen werden, die in diesen Sprachelementen selbst oder einem Codeblock des Statecharts definiert wurden.
Schweregrad: Error
Bemerkung: Entry-/exit-Aktionen, do-Aktivitäten und Zustandsinvarianten bilden jeweils einen eigenen Namensraum, aus dem auf Elemente des Statechart-Kontextes zugegriffen werden kann (siehe SC-4-04).

Transitionen:

SC-3-02: In einer Transition kann nur auf lokale Variablen zugegriffen werden, die innerhalb der Transition selbst oder einem Codeblock des Statecharts definiert wurden.
Schweregrad: Error
Bemerkung: Wie bei SC-3-01 ist ebenfalls der Zugriff auf Elemente des Statechart-Kontextes möglich (siehe SC-4-04).

4.4.5 Sequenzdiagramme

Interaktionen:

SD-3-01: In den Argumenten einer Interaktion können nur Typen und Objekte referenziert werden, die im zugehörigen Sequenzdiagramm sichtbar sind.

Schweregrad: Warning/Error
Bemerkung: Die Situation ist ähnlich wie in OD-3-02. Weitere mögliche Referenzen und Aufrufe werden daher in SD-4-08 behandelt.

Bedingungen:

SD-3-02: In Bedingungen können nur Typen und Objekte referenziert werden, die im zugehörigen Sequenzdiagramm sichtbar sind.
Schweregrad: Warning/Error
Bemerkung: vgl. SD-3-01.

4.5 Sprachübergreifende Inter-Modell-Bedingungen

Das UML/P-Framework erlaubt es, die Zusammensetzung der Sprachfamilie zu verändern (siehe Kapitel 7). So können Sprachen hinzugefügt oder entfernt werden. Wie die sprachübergreifenden Intra-Modell-Bedingungen in Abschnitt 4.4 beziehen sich deshalb die folgenden Kontextbedingungen auf allgemeine Anforderungen, die für Beziehungen zu Modellen in anderen Sprachen erfüllt sein müssen. Diese ergänzen die jeweiligen sprachinternen Bedingungen.

4.5.1 Klassendiagramme

Klassendiagramme:

CD-4-01: Es dürfen nur solche Elemente anderer Sprachen importiert werden, die sich in eine der Typarten `Class`, `Interface` oder `Enum` oder in eine Assoziation übersetzen lassen. Je nach Übersetzung gelten die entsprechenden sprachinternen Kontextbedingungen.
Schweregrad: Error
Bemerkung: In Klassendiagrammen ist nur die Verwendung der oben genannten Elemente vorgesehen. Deshalb muss für jede neben den Klassendiagrammen verwendete Sprache überlegt werden, wie sich deren Konzepte und Zugriffe auf diese Elemente übertragen lassen, um einen Import in den Diagrammen zu ermöglichen (siehe Abschnitt 4.1 und 7.2).

4.5.2 Objektdiagramme

Objektdiagramme:

OD-4-01: Es dürfen nur solche Elemente anderer Sprachen importiert werden, die sich in eine der Typarten `Class`, `Interface` oder `Enum` oder in ein Objekt übersetzen lassen. Je nach Übersetzung gelten die entsprechenden sprachinternen Kontextbedingungen.

Schweregrad: Error
Bemerkung: siehe OD-2-01 und CD-4-01.

Objekte:

OD-4-02: Der Typ eines Objekts ist innerhalb des Objektdiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: siehe auch CD-2-02.

OD-4-03: Die Modifikatoren eines Objekts sind eine Teilmenge der Modifikatoren des Objekttyps.
Schweregrad: Warning/Error
Bemerkung: Die Angaben eines Objekts müssen nicht alle Details des zugehörigen Typs widerspiegeln. Sind jedoch zusätzliche Modifikatoren im Objekt angegeben, stellt dies einen Widerspruch dar, der je nach Einsatzzweck oder Entwicklungsstand des Modells zu bewerten ist (vgl. All-1-05).

OD-4-04: Bei einem vollständigen Objekt ist für jedes Attribut des Objekttyps ein entsprechendes Attribut im Objekt angegeben.
Schweregrad: Error
Bemerkung: Dies gilt auch, falls für das zugehörige Objektdiagramm die Vollständigkeit zugesichert ist. Falls die Typdefinition unvollständig ist, können jedoch zusätzliche Attribute im Objektdiagramm aufgeführt sein (vgl. All-1-05).

Attribute:

OD-4-05: Der Typ eines Attributs ist innerhalb des Objektdiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: siehe auch CD-2-02.

OD-4-06: Für jedes Attribut eines Objekts existiert ein Attribut mit zuweisungskompatibler Typangabe und gleichem Namen im zugehörigen Objekttyp.
Schweregrad: Warning/Error
Bemerkung: Im Sinne der Unterspezifikation ist explizit der Typ des Attributs an der Definitionsstelle zuweisungskompatibel zur Typangabe des Attributs im Objekt und nicht umgekehrt. Für bestimmte Anwendungsfälle kann dies auf die Forderung der Identität eingeschränkt werden, wie es ebenfalls vom UML-Standard [OMG10d] gefordert ist. Ist kein Typ aber ein Wert für das Attribut angegeben, gilt dies entsprechend für den daraus abgeleiteten Typ.
Die Bewertung dieser Kontextbedingung ist abhängig vom Entwicklungsstand der Modelle (vgl. All-1-05). Insbesondere kann die Typspezifikation in Klassendiagrammen unvollständig sein, wodurch im Objektdiagramm Attribute

angegeben sein können, die keine Entsprechung im Klassendiagramm besitzen. Trotzdem sollte auch in solchen Fällen zumindestens eine Warnung ausgegeben werden. Ein Fehler ist es hingegen in jedem Fall, wenn ein gleichnamiges Attribut im Klassendiagramm spezifiziert ist, für das die Zuweisungskompatibilität nicht gilt.

OD-4-07: Die Modifikatoren eines Attributs im Objekt sind eine Teilmenge der Modifikatoren des entsprechenden Attributs im Objekttyp.
Schweregrad: Warning/Error
Bemerkung: siehe OD-4-03.

OD-4-08: Bei einem vollständigen Objekt ist für jedes Attribut im zugehörigen Objekttyp ein entsprechend OD-4-06 und OD-4-07 kompatibles Attribut im Objekt angegeben.
Schweregrad: Warning
Bemerkung: Dies gilt auch, falls für das zugehörige Objektdiagramm die Vollständigkeit zugesichert ist. Allerdings ist eine solche Markierung nur als Zusicherung des Modellierers zu verstehen. Aufgrund des generellen exemplarischen Charakters eines Objektdiagramms stellt die Verletzung dieser Kontextbedingung deshalb kein inkonsistentes Modell und damit keinen Fehlerfall dar.

OD-4-09: Die Wertangabe eines Attributs in Form eines komplexen Ausdrucks greift nur auf Methoden und Konstruktoren zu, die in dem Objekt sichtbar sind, in dem das Attribut definiert wurde.
Schweregrad: Warning/Error
Bemerkung: Da Methoden und Konstruktoren im Objektdiagramm nicht angegeben sind, kann dies nur mit Hilfe der Definitionsstelle des Objekttyps überprüft werden (siehe auch OD-3-02).

OD-4-10: Die Werte von gleichnamigen statischen Attributen sind für Objekte gleichen Typs innerhalb eines Objektdiagramms identisch oder nicht angegeben.
Schweregrad: Error
Bemerkung: Statische Attribute werden auch als Klassenattribute bezeichnet und sind nur über den Typnamen zugreifbar.

Links:

OD-4-11: Die referenzierten Objekte jeweils einer Seite eines Links haben einen gemeinsamen Supertyp.
Schweregrad: Error
Bemerkung: Jeder Link soll nur Instanzen einer Assoziation darstellen (siehe OD-4-12).

OD-4-12: Für einen Link existiert eine kompatible Assoziation zwischen zwei gemeinsamen Supertypen beider Linkseiten.

Schweregrad: Warning/Error

Bemerkung: Eine Assoziation ist genau dann kompatibel zu einem Link, wenn die Rollennamen, Sichtbarkeiten und Navigationsrichtungen im Link identisch sind, der Assoziationsname dem Linknamen entspricht und die Anzahl der Objektreferenzen nicht größer ist als die Kardinalitätsangabe der Assoziation. Ist darüber hinaus bei qualifizierten Assoziationen auch ein Qualifikator für den Link angegeben, muss dieser entweder auf dasselbe Attribut verweisen oder, falls stattdessen ein Wert angegeben wurde, muss dessen Typ zuweisungskompatibel zum Qualifikator der Assoziation sein. Aufgrund von Unterspezifikation kann die Angabe von Qualifikator, Rollennamen, Sichtbarkeiten, des Linknamen und der Navigationsrichtung im Objektdiagramm auch fehlen. Auch kann eine Aggregation oder Komposition im Objektdiagramm als normaler Link angegeben werden. Andere Abweichungen dieser Kennzeichnung sind jedoch nicht erlaubt. Bzgl. der Bewertung fehlender Assoziationen in unvollständigen Klassendiagrammen ist die Situation ähnlich wie bei Attributen (siehe OD-4-06).

OD-4-13: Ein Qualifikator referenziert ein Objekt eindeutig.

Schweregrad: Error

Bemerkung: Dadurch wird gefordert, dass Qualifikatorwerte von Links derselben qualifizierten Assoziation innerhalb eines Objektdiagramms eindeutig sind. Referenziert der Qualifikator ein Attribut, muss dessen Wert innerhalb von Objekten gleichen Typs eindeutig sein (sofern angegeben).

4.5.3 Statecharts

Statecharts:

SC-4-01: Es dürfen nur solche Elemente anderer Sprachen importiert werden, die sich in eine der Typarten `Class`, `Interface` oder `Enum` übersetzen lassen. Je nach Übersetzung gelten die entsprechenden sprachinternen Kontextbedingungen.

Schweregrad: Error

Bemerkung: vgl. CD-4-01.

SC-4-02: Statecharts können einer Klasse oder der Methode einer Klasse zugeordnet werden. In beiden Fällen muss die Klasse sichtbar sein.

Schweregrad: Error

Bemerkung: Statecharts werden in der UML/P im Wesentlichen zu Spezifikationszwecken eingesetzt. Es ist daher auszuschließen, dass das Verhalten eines Elements vor dessen Definition modelliert wird, so dass beim Fehlen der Klassenspezifikation

von einem Fehlerfall ausgegangen wird. Dies kann jedoch im Bedarfsfall durch eine schwächere Bewertung ersetzt werden.

SC-4-03: Ist ein Statechart einer Methode zugeordnet, müssen die Parametertypen und der Name der Methode mit einer Methode der zugehörigen Klasse übereinstimmen.
Schweregrad: Error
Bemerkung: Insbesondere dürfen die Namen der Parameter im Statechart abweichen. In solch einem Fall werden die Parameter über die im Statechart angegebenen Namen referenziert (siehe auch SC-4-02).

SC-4-04: In Statecharts kann nur auf Methoden, Konstruktoren, Attribute und assoziierte Objekte zugegriffen werden, die in dem Objekt sichtbar sind, dessen Verhalten das Statechart modelliert. Bei Methodenstatecharts gilt dies für das zur Methode gehörige Objekt, wobei zusätzlich der Zugriff auf die Methodenparameter möglich ist.
Schweregrad: Error
Bemerkung: Davon ist insbesondere der eingebetteten Code von entry-/exit-Aktionen, do-Aktivitäten, Zustandsinvarianten, Transitionen und Codeblöcken betroffen. Diese Sprachanteile werden demnach im Kontext des Objekts bzw. der Methode spezifiziert, dessen Verhalten das Statechart modelliert (vgl. SC-3-01 und SC-4-01).

Transitionen:

SC-4-05: Handelt es sich bei dem Stimulus einer Transition um den Aufruf einer Methode, so lässt sich dieser anhand des Namens und der Argumenttypen eindeutig einer Methode des Typs zuordnen, dessen Verhalten das Statechart spezifiziert.
Schweregrad: Error
Bemerkung: Insbesondere sind hier auch private Methoden des Typs möglich. Da Methoden darüber hinaus wie in Java überladen werden können, erfolgt die Auswahl der passenden Methode anhand der Argumenttypen entsprechend dem Vorgehen im Java-Standard [GJS05].

SC-4-06: Handelt es sich bei dem Stimulus einer Transition um den Aufruf einer Exception, so muss diese innerhalb des Statecharts sichtbar sein. Darüber hinaus wird gefordert, dass die die Argumenttypen zuweisungskompatibel zu den Parametertypen eines sichtbaren Konstruktors der Exception sind.
Schweregrad: Error
Bemerkung: siehe SC-4-05.

4.5.4 Sequenzdiagramme

Sequenzdiagramme:

SD-4-01: Es dürfen nur solche Elemente anderer Sprachen importiert werden, die sich in eine der Typarten `Class`, `Interface` oder `Enum` oder in ein Objekt übersetzen lassen. Je nach Übersetzung gelten die entsprechenden sprachinternen Kontextbedingungen.
Schweregrad: Error
Bemerkung: Insbesondere ist der Import von Objekten aus anderen Sequenzdiagrammen nicht vorgesehen (siehe Abschnitt 4.3). Vgl. auch CD-4-01.

Objekte:

SD-4-02: Der Typ eines Objekts ist innerhalb des Sequenzdiagramms sichtbar.
Schweregrad: Warning/Error
Bemerkung: siehe auch CD-2-02.

SD-4-03: Die Modifikatoren eines Objekts sind eine Teilmenge der Modifikatoren des Objekttyps.
Schweregrad: Warning/Error
Bemerkung: siehe OD-4-03.

Interaktionen:

SD-4-04: Für einen Methodenaufruf existiert eine gleichnamige sichtbare Methode im Typ des Zielobjekts der Interaktion, wobei die Argumente entweder unvollständig oder zuweisungskompatibel zu den Typen der Methodenparameter sind.
Schweregrad: Warning/Error
Bemerkung: Ähnlich wie Objektdiagramme können auch Sequenzdiagramme bereits in einer frühen Entwicklungsphase eingesetzt werden, was zu unterschiedlichen Gewichtungen dieser Kontextbedingung führt (vgl. All-1-05). Bzgl. Auswahl der Methode siehe SC-4-05.

SD-4-05: Wird eine Exception übertragen, so ist diese als Typ innerhalb des Sequenzdiagramms sichtbar. Angegebene Argumente sind darüber hinaus zuweisungskompatibel zu den Parametern eines sichtbaren Konstruktors des Exceptiontyps.
Schweregrad: Warning/Error
Bemerkung: vgl. SD-4-04.

SD-4-06: Wird ein neues Objekt instanziiert, so ist dessen Typ innerhalb des Sequenzdiagramms sichtbar. Angegebene Argumente sind darüber hinaus zuweisungskompatibel mit den Parametern eines sichtbaren Konstruktors des Objekttyps.

Schweregrad: Warning/Error
Bemerkung: vgl. SD-4-04.

SD-4-07: In den Argumenten einer Interaktion oder in `return`-Ausdrücken kann nur auf Methoden, Konstruktoren, Attribute und assoziierte Objekte zugegriffen werden, die in dem Quellobjekt der Interaktion sichtbar sind.
Schweregrad: Warning/Error
Bemerkung: Insbesondere ist der Zugriff auf private Elemente des Quellobjekts möglich (vgl. auch SD-3-01 und SD-4-04).

SD-4-08: Für jedes Quellobjekt eines `return`-Aufrufs gibt es innerhalb derselben Aktivität eine zeitlich vorgelagerte Interaktion mit diesem als Zielobjekt. Handelt es sich dabei um einen Methodenaufruf, muss darüber hinaus der Wert des `return`-Aufrufs zuweisungskompatibel zum Rückgabewert der Methode sein.
Schweregrad: Warning/Error
Bemerkung: vgl. SD-4-04.

Aktivitäten:

SD-4-09: Bei der vollständigen Angabe der Interaktionen eines Objekts wird jede Aktivität durch einen ausgehenden `return`-Aufruf oder die Übertragung einer Exception abgeschlossen.
Schweregrad: Error
Bemerkung: Dies gilt auch, falls für das zugehörige Sequenzdiagramm die Vollständigkeit zugesichert ist (siehe SD-1-01). Insbesondere ist SD-4-08 für die erste und letzte Interaktion einer Aktivität gültig.

Bedingungen:

SD-4-10: Die in einer Bedingung referenzierten Methoden, Konstruktoren, Attribute und assoziierten Objekte müssen sich einem Objekt im Sequenzdiagramm eindeutig zuordnen lassen und sichtbar sein.
Schweregrad: Error
Bemerkung: Für den Zugriff auf die von der Bedingung überdeckten Objekte ist keine Objektreferenz notwendig, solange der jeweilige Zugriff innerhalb der überdeckten Objekte eindeutig ist. Wird die Bedingung in OCL/P spezifiziert, besteht darüber hinaus für den Zugriff auf die Objekte keine Beschränkung hinsichtlich der Sichtbarkeit.

4.5.5 Testspezifikationssprache

Testfälle:

TC-4-01: Importierte Modelle müssen sichtbar und als Objektaufbau oder Ablaufbeschreibung interpretierbar sein. Gleichzeitig werden die in den Modellen definierten Objekte importiert. Darüber hinaus können Elemente der eingebetteten Sprache importiert werden.
Schweregrad: Error
Bemerkung: In der vorliegenden Fassung erlaubt die Testspezifikationssprache den Import von Objekt- und Sequenzdiagrammen als Objektaufbau bzw. Ablaufbeschreibung. Soll die Testspezifikationssprache mit anderen Modellarten kombiniert werden, ist eine entsprechende Interpretation dieser Modelle notwendig (vgl. CD-4-01). Durch die Einbettung von Java/P ist es insbesondere möglich, auch auf Typen aus Klassendiagrammen und Java zuzugreifen und diese zu importieren (siehe Abschnitt 3.7). Darüber hinaus ist es nicht ausgeschlossen, dass die Testspezifikationssprache in zukünftigen Versionen der UML/P etwa um die Verwendung der OCL/P erweitert wird.

Unit-Tests:

TC-4-02: Lokale und unbenannte Objekte aus den importierten Modellen sind innerhalb des Testfalls nicht sichtbar.
Schweregrad: Error

TC-4-03: Die Referenzen im `setup`- und `result`-Bereich eines Unit-Tests verweisen jeweils auf einen im Testfall sichtbaren Objektaufbau.
Schweregrad: Error
Bemerkung: Ein Objektaufbau ist genau dann sichtbar, wenn dieser importiert wird oder sich im gleichen Paket wie der Testfall befindet.

TC-4-04: Die Referenz im `events`-Bereich eines Unit-Tests verweist auf eine im Testfall sichtbare Ablaufbeschreibung.
Schweregrad: Error
Bemerkung: vgl. TC-4-03.

TC-4-05: Im Codeblock können nur sichtbare Objekte des im `setup`-Bereich des zugehörigen Unit-Tests referenzierten Objektaufbaus sowie Elemente entsprechend der eingebetteten Sprache referenziert werden.
Schweregrad: Error
Bemerkung: Die Referenzierung von Objekten erfolgt über deren Namen.

Die in diesem Kapitel aufgeführten Kontextbedingungen stellen einen Basissatz dar, der projekt- oder unternehmensspezifischen Anforderungen entsprechend angepasst und erweitert werden kann. Dies wird von dem in Kapitel 7 beschriebenen Framework unterstützt. So können etwa der Einsatz spezieller Codegeneratoren oder Zielplattformen zusätzliche Bedingungen oder andere Gewichtungen notwendig machen, wie z.B. das Verbot von Mehrfachvererbung in Klassendiagrammen, wenn diese auf der Zielplattform durch den Generator nicht umgesetzt werden kann. Auch die in Kapitel 6 vorgestellten Konzepte für eine Codegenerierung aus der UML/P stellen weitere Anforderungen an die Modelle, die bereits in Abschnitt 4.3 diskutiert wurden. Darüber hinaus bietet die UML/P über die Stereotypen die Möglichkeit, die Sprache zu erweitern oder an bestimmte Domänen anzupassen (Profilbildung), die ebenfalls zusätzliche Kontextbedingungen mit sich bringen können.

Die einheitliche Strukturierung und Gruppierung der Kontextbedingungen erleichtert die Übersicht und die Auffindbarkeit. Zusätzlich kann über die ID ein eindeutiger Bezug zu einer Bedingung hergestellt werden, was bereits in den Diskussionen in diesem Kapitel genutzt wurde, aber auch in der Implementierung und für zukünftige Erweiterungen eingesetzt werden kann. Für letzteres können die IDs entsprechend fortgeführt, oder auch projekt- oder unternehmensspezifische Kürzel eingeführt werden. Darüber hinaus wurde darauf geachtet, dass sich auch die sprachübergreifenden Kontextbedingungen nicht auf eine bestimmte Sprachkombination beziehen. Dadurch können die Teilsprachen der UML/P auch getrennt voneinander oder in Kombination mit anderen Sprachen eingesetzt werden. Auch in diesem Punkt bietet insbesondere die Aufteilung in sprachinterne und sprachübergreifende Bedingungen den Vorteil einer leichteren Wiederverwendbarkeit. Auswirkungen der Bedingungen auf die in dieser Arbeit vorgestellten Kombinationen wurden in den Bemerkungen diskutiert.

In den bisherigen Veröffentlichungen zur UML/P [Rum04a, Rum04b] wurde nur am Rande auf Kontextbedingungen eingegangen. Darüber hinaus ergeben sich viele der hier aufgeführten Bedingungen erst durch die in Abschnitt 3.9 beschriebenen Unterschiede zur ursprünglichen graphischen Fassung. Auch der UML-Standard [OMG10d] gibt Bedingungen zu einzelnen Sprachelementen an, behandelt aber nicht den Aspekt der unterschiedlichen Gewichtung je nach Einsatzzweck oder Entwicklungsstand der Modelle.

Durch die Sammlung an Kontextbedingungen und deren Diskussion werden viele Detailfragen zur UML/P behandelt, die im Laufe der Entwicklung dieser Arbeit des öfteren aufgetaucht sind. Nicht zuletzt trägt dieses Kapitel dadurch zu einem vertiefenden Verständnis der UML/P bei.

Kapitel 5

Templatebasierte Codegenerierung

Die UML wird immer noch primär zur Planung und Dokumentation von Softwaresystemen eingesetzt [WWM+07]. Für die Implementierung dienen die Modelle nur als Grundlage für die weiterhin manuelle Umsetzung in eine Programmiersprache. Durch diese Vorgehensweise entstehen jedoch zwei in Teilen redundante Beschreibungen des eigentlichen Zielsystems. Werden diese während der Entwicklung nicht konsistent gehalten, verlieren die ursprünglich erstellten Modelle an Wert und können etwa für Dokumentationszwecke nicht mehr ohne Weiteres wiederverwendet werden. Der notwendige zusätzliche Aufwand für die Überarbeitung der Modelle führte sogar dazu, dass agile Vorgehensweisen wie Extreme Programming (XP, [BA04]) entwickelt wurden, die weitestgehend auf Planung und Modellierung verzichten und stattdessen den lauffähigen Code in den Vordergrund stellen.

Um den Nutzen der Modelle zu erhöhen, unterstützten erste Modellierungswerkzeuge wie Rational Rose [Qua02, BB02] oder ArgoUML [RVR+09] schon früh die Generierung von Coderümpfen aus Klassendiagrammen. Diese wurden manuell um die Verhaltensimplementierung erweitert. Danach kann jedoch die Generierung bei Änderungen an den Modellen nicht erneut durchgeführt werden, um die manuellen Implementierungsdetails nicht zu überschreiben. Auf diese Weise wird somit ebenfalls nur zu Beginn die Konsistenz zwischen Modellen und Code sichergestellt. Um dieses Problem zu umgehen, wurden geschützte Bereiche in den Coderümpfen eingeführt, die bei der Generierung nicht überschrieben werden. Dieses Verfahren funktioniert jedoch nur in einfachen Fällen, bei denen sich Änderungen am Generator oder den Modellen nicht auf den manuellen Code auswirken (siehe auch Abschnitt 8.3).

Die Generierung von Verhalten wurde hingegen lange nicht unterstützt. Eine umfassende und vollständige Generierung des Zielsystems aus Modellen bietet jedoch eine Reihe von Vorteilen:

- **Effizienter Code**: Laut [CE00] ist es oft schwierig oder sogar unmöglich, gute Performance, verständliche Codierung und Flexibilität in einer Implementierung zu vereinen. Generatoren können die Aufgabe übernehmen, für eine leicht verständliche Spezifikation effizienten Code zu erzeugen.
- **Entlastung der Entwickler**: Die manuelle Umsetzung der Modelle ist in vielen Fällen wiederkehrende und gleichbleibende Arbeit. Durch den Einsatz von Codegeneratoren kön-

nen die Entwickler von diesen Arbeiten entlastet werden. Gleichzeitig ist dieses Vorgehen weniger fehleranfällig und bietet weiter reichende Möglichkeiten wie die Einhaltung von Codierungsstandards.

- **Konsistente Umsetzung**: Oft werden bei manueller Umsetzung die Modelle bei späteren Änderungen vernachlässigt, so dass Modelle und Code nicht mehr konsistent sind. Für die Dokumentation ist dann häufig ein aufwändiges Reverse Engineering notwendig.
- **Testbarkeit des Systems**: Neben der Einhaltung von Codierungsstandards kann auch die Testbarkeit des Systems durch Generatoren sicher gestellt werden. Darüber hinaus besteht die Möglichkeit, unterschiedliche Generatoren für das System zur Entwicklungszeit und für das eingesetzte System zu entwickeln. Auf diese Weise kann während der Entwicklung zusätzliche Infrastruktur wie z.B. Mock-Klassen oder Codeinstrumentierung sowie weiterreichende Zugriffsmöglichkeiten etwa für OCL-Ausdrücke zur Verfügung stehen, um leichter und umfangreicher testen zu können. Sicherheits- oder Performancegründe können es erfordern, dass diese Testinfrastruktur hingegen nicht Bestandteil des finalen Systems sein darf.
- **Kapselung von technischen Details**: Modelle besitzen eine höhere Abstraktion als der daraus abgeleitete ausführbare Systemcode. Eine sinnvolle Möglichkeit der Abstraktion ist, die Modelle frei von technischen Details der Zielplattform zu halten. Oft erfordern diese Details viel Expertenwissen über die Plattform, um etwa Performanceanforderungen wie Ausführungsgeschwindigkeit oder Speicherbedarf zu optimieren. Dieses Wissen kann in Generatoren gekapselt werden und ist auf diese Weise wiederverwendbar.
- **Unterstützung mehrerer Plattformen**: Soll ein System mehrere Plattformen unterstützen oder auf eine andere Plattform portiert werden, erfordern die technischen Unterschiede oft eine jeweils angepasste Implementierung. Sind die Modelle wie im vorigen Punkt beschrieben frei von technischen Details, können diese Implementierungen aus den Modellen durch den Einsatz mehrerer entsprechender Generatoren erzeugt werden.

Trotz dieser Vorteile beschränkt sich selbst heute die Codegenerierung aus Verhaltensmodellen oft auf die Erzeugung von Testfällen oder fehlt weiterhin vollständig wie bei ArgoUML, so dass das Verhalten nachträglich im generierten Code ergänzt werden muss. Im Gegensatz dazu wird in diesem und im folgenden Kapitel ein modulares Verfahren der Codegenerierung vorgestellt, mit dessen Hilfe sich Architektur und Verhalten eines Softwaresystems mit unterschiedlichen Artefakten bzw. Sprachen beschreiben lässt, ohne dass eine Anpassung des generierten Codes notwendig ist. Dazu werden zunächst im folgenden Abschnitt 5.1 die Grundlagen erläutert. Daran anschließend beschreibt Abschnitt 5.2 den allgemeinen Aufbau der in dieser Arbeit vorgeschlagenen Generatoren, mit deren Implementierung und Erweiterung sich schließlich die Abschnitte 5.3, 5.4 und 5.5 befassen.

5.1 Grundlagen

5.1.1 Modelltransformationen

Bei der Codegenerierung handelt es sich im Grunde um eine spezielle Form der Modelltransformation, die ein Modell in eine Programmiersprache überführt. In [CE00] wird eine Transformation wie folgt definiert:

Definition 2 (Transformation) *Eine Transformation ist eine automatisierte, semantisch korrekte (im Gegensatz zur beliebigen) Modifikation der Repräsentation eines Programms.*

Modelltransformationen sind demnach Transformationen, die Modelle als Programmrepräsentation verwenden. Allgemein können drei wesentliche Kategorien von Modelltransformationen unterschieden werden [CH03, CH06]:

- **Modell-zu-Modell**: Diese Transformationen werden auf Basis der abstrakten Syntax bzw. der Metamodelle spezifiziert. Quell- und Zielmodelle einer Transformation können dabei sowohl derselben Sprache angehören, als auch verschiedenen Sprachen und damit Metamodellen entsprechen.
- **Modell-zu-Text**: Das Ziel dieser Transformationen setzt sich im Gegensatz zu Modell-zu-Modell-Transformationen aus Textstücken (Strings) zusammen. Diese können zwar ebenfalls konform zu einem Metamodell sein, das bei der Erzeugung jedoch nicht verwendet wird.
- **Text-zu-Modell**: Diese Form der Modelltransformation wird in [CH06] nur am Rande erwähnt, soll hier aber als dritte Kategorie mit aufgenommen werden. In diesem Fall besteht die Eingabe der Transformation aus Text, der im Allgemeinen mit Hilfe eines Parsers in ein Modell überführt wird.

[CE00] unterscheidet darüber hinaus in vertikale Transformationen, die abstraktere Darstellungen in eine weniger abstrakte überführen, sowie horizontale Transformationen, die auf demselben Abstraktionslevel stattfinden.

Die Schwierigkeit in der vorliegenden Arbeit besteht darin, dass die Modelle selbst textuell dargestellt werden, so dass die obige Kategorisierung auf den ersten Blick verwirrend erscheinen mag. Dabei ist allerdings zu beachten, dass es sich bei der textuellen Notation nur um eine mögliche Beschreibungsform des eigentlichen Modells handelt. Im Falle der UML/P existiert für die meisten Teilsprachen ebenfalls eine graphische Notation (siehe Kapitel 3 und [Rum04a, Rum04b]), so dass ein und dasselbe Modell sowohl graphisch als auch textuell beschrieben werden kann. Unabhängig von der Art der Darstellung wird aus einem Dokument jedoch im Grunde erst dann ein Modell, wenn es als solches interpretiert wird. Dies kann durch einen menschlichen Leser geschehen, der anhand der Notation das Modell erkennt, oder durch die Verarbeitung eines Werkzeugs. In der vorliegenden Arbeit geschieht letzteres durch den Parser, der aus den jeweiligen Grammatikbeschreibungen der Sprachen vom MontiCore-Framework generiert wird [Kra10, GKR+06]. Der Parser liest eine Datei bzw. einen String ein und erzeugt daraus mit

dem abstrakten Syntaxbaum (AST[1]) eine interne Datenstruktur, die die weitere Verarbeitung erlaubt. Die Erstellung des ASTs ist nur möglich, wenn das Modell vom Parser erkannt wird, so dass dieser Schritt demnach als "Text-zu-Modell"-Transformation verstanden werden kann (siehe Abbildung 5.1). "Modell-zu-Modell"-Transformation wiederum finden auf Basis des ASTs statt. Dabei kann der AST selbst modifiziert werden, so dass sich zwar das Modell, nicht aber dessen Sprache verändert (endogene Transformation [MCG05]). Soll hingegen aus dem bestehenden Modell ein Modell einer anderen Sprache abgeleitet werden (exogene Transformation), erfordert dies den Aufbau eines entsprechenden neuen ASTs. In jedem Fall ist durch den bestehenden AST das Modell als solches im Werkzeug existent. Im Gegensatz dazu ist das Ergebnis einer "Modell-zu-Text"-Transformation ein String oder ein Dokument, für das nicht bekannt ist, ob es eine Interpretation als Modell zulässt. Häufig handelt es sich dabei um den Abschluss einer Reihe von "Modell-zu-Modell"-Transformationen, um das Ergebnis als Datei zu sichern. Darüber hinaus können sich "Modell-zu-Modell"-Transformationen intern aus einer "Modell-zu-Text"- und "Text-zu-Modell"-Transformation zusammensetzen, wobei im Allgemeinen der Text der ersten Transformation nur im Speicher erzeugt und direkt als Eingabe der zweiten Transformation verwendet wird. Der Vorgang der Umwandlung eines ASTs in die zugehörige textuelle Notation wird auch als Pretty-Print bezeichnet. Auf diese Weise können ebenfalls Dateien einer Programmiersprache entstehen, weshalb bei solchen Transformationen von Codegenerierung gesprochen wird. Darauf aufbauend lässt sich ein Generator bzw. Codegenerator wie folgt definieren:

Definition 3 (Generator) *Ein Generator transformiert eine Menge von Eingabedateien in eine Menge von Ausgabedateien. Das Ergebnis eines Generators wird als Generat bezeichnet.*

Definition 4 (Codegenerator) *Ein Codegenerator transformiert eine Menge von Eingabedateien in eine Menge von ausführbaren Dateien einer Programmiersprache.*

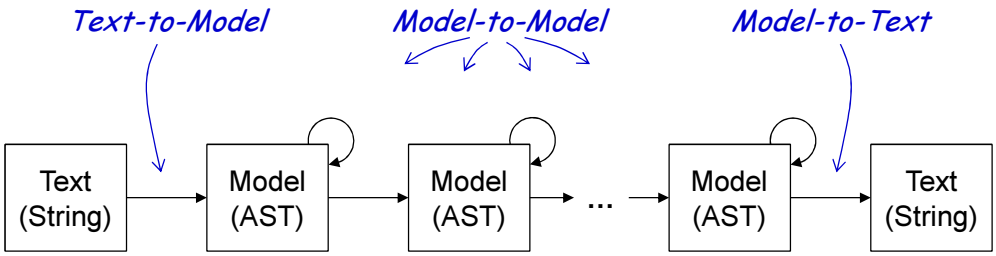


Abbildung 5.1: Modelltransformationen

Wie die obigen Erläuterungen zeigen, spielt der AST für die automatisierte Ausführung von Transformationen eine zentrale Rolle. Dessen Aufbau basiert wiederum auf der abstrakten Syntax,

[1]engl.: Abstract Syntax Tree

die häufig auch als Metamodell bezeichnet wird. Diese wird von MontiCore aus der Grammatikbeschreibung einer Sprache abgeleitet [Kra10, GKR+06]. Abbildung 5.2 zeigt aufbauend auf der klassischen vier-Schichten Architektur der OMG [OMG10c] die einzelnen Ebenen, die durch diese Form der Sprachdefinition entstehen. Auf der untersten Ebene M0 sind Objekte der realen Welt dargestellt, also im Falle der Softwareentwicklung das eigentliche System ohne Abstraktion. Diese wird durch die erste Abstraktionsebene M1 modelliert. In Abbildung 5.2 ist dazu beispielhaft ein Statechart dargestellt, sowie der dazugehörige AST als Objektdiagramm, der vom Parser aus dem Statechart erstellt wird. Dies ist demnach die Ebene, in der die einzelnen Teilsprachen der UML/P verwendet werden, um das System zu beschreiben. Dabei befolgen die Modelle den Grammatikdefinitionen im Anhang C. Diese konkrete und die daraus abgeleitete abstrakte Syntaxdefinitionen bilden die Meta-Ebene M2. Die als Klassendiagramm dargestellte abstrakte Syntax wird dabei vom MontiCore-Framework als entsprechender Java-Code umgesetzt.

Da die Grammatikdefinitionen selbst wiederum in einer Sprache verfasst sind - in diesem Fall der Grammatikbeschreibungssprache von MontiCore, die in Abbildung 5.2 mit "MG" gekennzeichnet ist - existiert für diese ebenfalls eine konkrete und abstrakte Syntax. Diese steht für die Meta-Metaebene M3. Im Prinzip könnte dieser Aufbau unendlich fortgesetzt werden. Wie in Abbildung 5.2 jedoch zu erkennen ist, befolgen die Beschreibungen der Statechart-Grammatik und der Grammatikbeschreibungssprache von MontiCore den gleichen Regeln. Die Meta-Metaebene M3 ist demnach mit sich selbst beschrieben, so dass eine weitere Ebene M4 nicht notwendig ist. Auch die Ableitung der abstrakten Syntax erfolgt genauso wie auf der Ebene M2. Technisch ist dies möglich, da MontiCore in einem evolutionären Entwicklungsprozess entstanden ist (Bootstrapping, [Kra10]), bei dem die aktuelle Version mit der vorherigen weiterentwickelt wird.

Die OMG beschreibt die Verbindung zwischen den einzelnen Ebenen als Instanzbeziehungen. Dies kann zwar im Prinzip so gesehen werden, wird jedoch den Unterschieden nicht ganz gerecht [Béz05a]. So handelt es sich bei den Beziehungen zwischen den Ebenen M1-M3 um eine Verbindung, die exakt festgelegten Regeln befolgt, die im vorliegenden Fall vom MontiCore-Framework vorgegeben werden. Aus diesem Grund wurde hier die Bezeichnung "conform to" ("konform zu") gewählt. Im Gegensatz dazu ist die Beziehung zwischen M0 und M1 weniger strikt festgelegt, was durch "represented by" ("dargestellt durch") ausgedrückt wird (in Anlehnung an [Béz04, Béz05a]). Insbesondere bieten die Modelle aufgrund der Abstraktion einen gewissen Interpretationsspielraum, der unter anderem je nach Vorgaben des Projekts, der technischen Zielplattform oder anhand von Unternehmensrichtlinien für das System unterschiedlich ausgelegt werden kann. Im Grunde handelt es sich dabei um Designentscheidungen, die bei der Umsetzung der Modelle in den Code getroffen werden müssen. Diese können in Codegeneratoren gekapselt werden.

5.1.2 MontiCore

Für das Verständnis und die Umsetzung von Generatoren ist es hilfreich, die grundlegenden Ableitungsregeln der abstrakten aus der konkreten Syntax von MontiCore zu kennen [KRV07b]. Auf diese Weise können die Grammatiken aus Anhang C als eine Art Dokumentation für

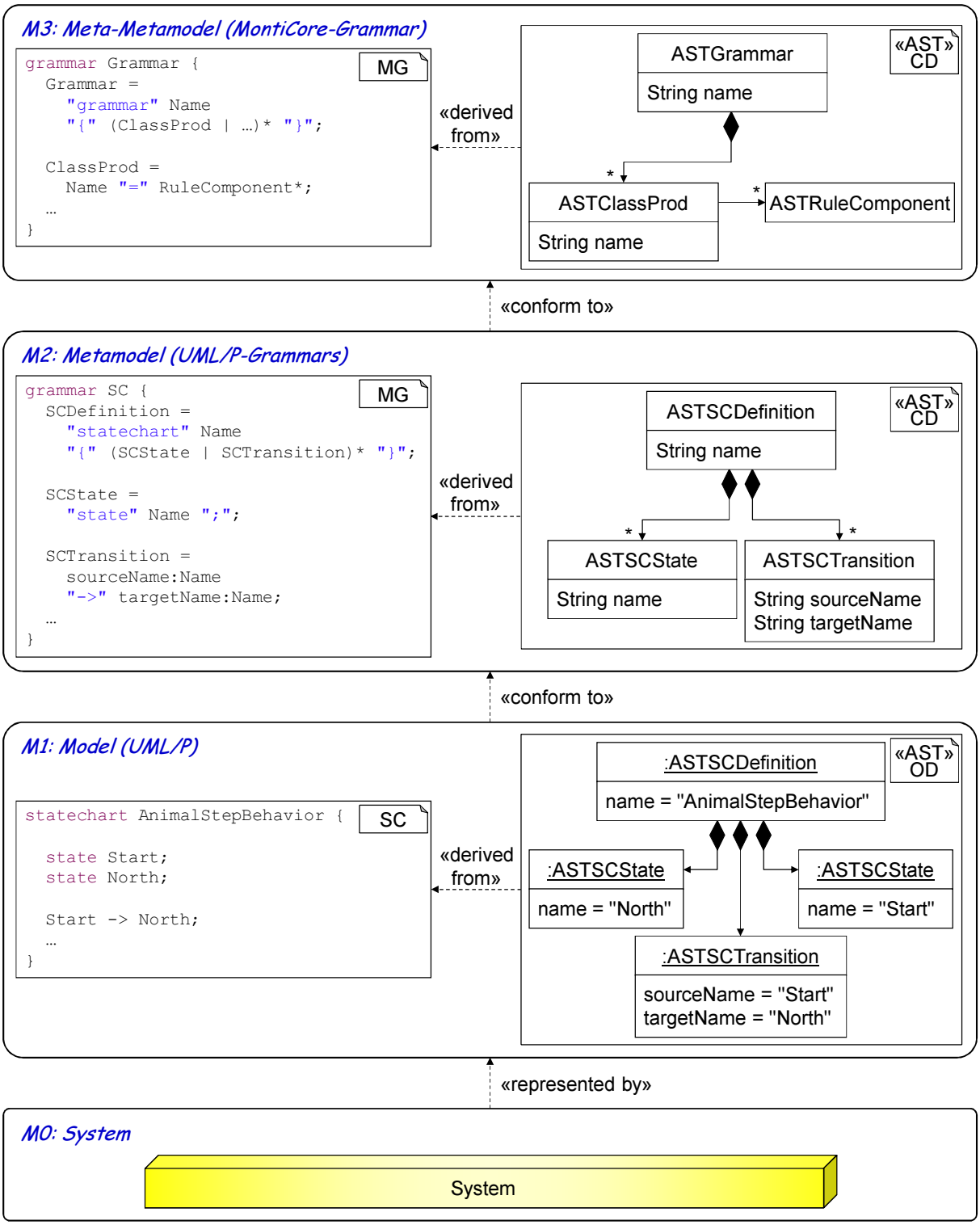


Abbildung 5.2: Hierarchie der Metamodellierung in der UML/P entsprechend der klassischen vier-Schichten Architektur der OMG [OMG10c]. Bei den dargestellten Grammatiken und der abstrakten Syntax handelt es sich nur um stark vereinfachte Auszüge.

die Generatorentwicklung genutzt werden. Dazu genügt es bereits, das Beispiel, mit dem die Metaebene M2 in Abbildung 5.2 dargestellt ist, zu betrachten.

Vereinfacht ausgedrückt setzt sich eine Grammatik aus mehreren Produktionen zusammen, die wiederum jeweils aus einer durch ein Gleichheitszeichen getrennten linken und rechten Seite bestehen. Die linke Seite definiert den Namen eines sogenannten Nichtterminals, dessen Aufbau auf der rechten Seite beschrieben wird. Dazu können auf der rechten Seite andere Nichtterminale, konstante Zeichenketten in Anführungszeichen (Terminale) sowie Strukturierungen in Form von Blöcken in runden Klammern verwendet werden. Um die Anzahl dieser Elemente genauer festzulegen, bietet MontiCore darüber hinaus die Angabe von Alternativen (`|`) und Kardinalitäten, wobei `?` ein optionales Element kennzeichnet, sowie der Kleene-Stern `*` für beliebig viele und `+` für mindestens ein Element steht. Ist keine Kardinalität angegeben, tritt das Element genau einmal auf.

In dem Auszug der Statechart-Grammatik in Abbildung 5.2 sind mit `SCDefinition`, `SCState` und `SCTransition` drei Nichtterminale definiert, die von MontiCore jeweils auf eine Klasse mit dem Prefix "AST" und dem Namen des Nichtterminals abgebildet werden. Auf der rechten Seite einer Produktion referenzierte Nichterminale werden konzeptuell als Assoziationen bzw. im Java-Code als Attribute mit entsprechenden get- und set-Methoden umgesetzt. Dies ist abhängig von der Kardinalität. So setzt sich laut Grammatik durch den Kleene-Stern eine `SCDefinition` aus mehreren Zuständen und Transitionen zusammen. Die zugehörigen get- und set-Methoden verwenden demnach eine Liste der entsprechenden Objekte, die im Java-Code als Klassen `ASTSCStateList` und `ASTSCTransitionList` repräsentiert werden. Ohne den Kleene-Stern würden die Objekte entsprechend direkt verwendet.

Eine Ausnahme von dieser Ableitungsregel bilden die sogenannten *lexikalischen Produktionen*, die die variablen Zeichenketten einer Sprache definieren. Diese werden in der Grammatikbeschreibungssprache von MontiCore mit dem Schlüsselwort `token` vor der Produktion gekennzeichnet und auf die Klasse `java.lang.String` der Java-Bibliothek abgebildet. Im Beispiel der Statechart-Grammatik ist `Name` eine solche lexikalischen Produktion, die hier nur verwendet wird. Die Definition findet sich in Quellcode C.2 im Anhang C.1. Für Listen von solchen Nichtterminalen wird mit der generischen Klasse `java.util.List<String>` eine weitere Klasse der Java-Bibliothek wiederverwendet. Die möglichen Werte der Strings werden durch die Beschreibung der rechten Produktionsseite über die angegebenen Terminale bestimmt. Ansonsten dienen Terminale wie `"state"` oder `"->"` nur der Lesbarkeit der konkreten Syntax und werden nicht in der abstrakten Syntax abgebildet.

Die obige Darstellung führt nur einen kleinen Teil der Konzepte und Möglichkeiten von MontiCore auf, sollte aber für das Verständnis des Folgenden ausreichen. Für einen tieferen Einblick sei hingegen auf [Kra10] verwiesen.

5.1.3 FreeMarker

Für die Umsetzung von Modell-zu-Text-Transformationen bzw. Codegeneratoren unterscheidet [CH03, CH06] zwei grundlegende Ansätze:

- **Visitorbasierter Ansatz**: Bei Visitoren (zu deutsch "Besucher") handelt es sich um ein spezielles Entwurfsmuster, das eine automatisierte Verarbeitung von zusammenhängenden Objektstrukturen wie dem AST erlaubt [GHJV95]. Ein Visitor durchläuft dabei nach einem festen Schema genau einmal die Objektstruktur und kapselt verschiedene Algorithmen, die abhängig vom vorgefundenen Objekttyp ausgeführt werden. Dabei können mehrere Visitoren kombiniert werden, wobei in der Grundversion jeweils nur ein Visitor pro Objekttyp aktiv ist. MontiCore bietet darüber hinaus verschiedene Varianten von Visitoren [Kra10]. Der MultiVisitor erlaubt die Reaktion von mehreren Visitoren auf einen Objekttyp. InheritanceVisitoren berücksichtigen hingegen die Vererbungsstruktur der Objekttypen, so dass auch Algorithmen ausgeführt werden, die für einen Supertyp des vorliegenden Objekttyps spezifiziert wurden. Schließlich existiert mit dem MultiInheritanceVisitor ebenfalls eine Kombination dieser beiden Varianten.

  Durch diese Spezifikation von Reaktionen auf bestimmte Objekttypen lassen sich Visitoren einsetzen, um gezielt einzelne Knoten eines ASTs in Strings zu überführen. Auf diese Weise lässt sich nach und nach der zu generierende Code aufbauen. Dieses Verfahren eignet sich jedoch nur, wenn die Struktur des generierten Codes dem AST ähnelt. Dazu muss sich jeder Knoten in einen in sich abgeschlossenen String übersetzen lassen, ohne dass dieser aufgrund anderer Knoten zu einem späteren Zeitpunkt nochmals modifiziert werden muss. Somit eignen sich Visitoren beispielsweise für die Programmierung von typischerweise strukturerhaltenden Pretty-Printern. Für komplexere Generatoren ist es hingegen oft sinnvoller, mit Hilfe der Visitoren den Ziel-AST aufzubauen und diesen abschließend mit einem weiteren Visitor in den Code zu überführen. Der Vorteil liegt darin, dass sich der AST im Allgemeinen leichter modifizieren lässt als ein String, da auch nachträglich noch Knoten hinzugefügt werden können.

- **Templatebasierter Ansatz**: Templates sind eine weitere Form, das Ergebnis einer Codegenerierung zu spezifizieren und bestehen im Wesentlichen aus festen Anteilen, Einschüben sowie Kontrollstrukturen. Erstere sind in der konkreten Syntax der Zielsprache formuliert und werden bei der Verarbeitung durch die Template-Engine unverändert übernommen. Im Gegensatz dazu werden Einschübe durch konkrete Werte etwa aus dem AST ersetzt. Schließlich bieten die Kontrollstrukturen die Möglichkeit, bestimmte Abschnitte zu wiederholen oder an Bedingungen zu knüpfen. Einschübe und Kontrollstrukturen sind dabei in einer Engine-spezifischen Sprache formuliert, die im Folgenden als Templatesprache bezeichnet wird. Auch die Verarbeitung von Daten aus dem AST ist damit im Allgemeinen möglich. Am Ende entsteht ein String oder eine Datei, die nur noch aus dem Zielcode besteht.

Beide Verfahren bieten Vor- und Nachteile beim Einsatz für die Codegenerierung. Visitoren abstrahieren von der konkreten Struktur des ASTs und werden in einer höheren Programmiersprache wie Java umgesetzt, die nicht zuletzt durch umfangreiche Bibliotheken und Konzepte wie die Objektorientierung für komplexe Berechnungen deutlich besser geeignet sind als die Templa-

tesprachen. Allerdings werden dabei oft die Strings, aus denen sich der Zielcode zusammensetzt, an vielen Stellen innerhalb der Algorithmik verteilt. Dadurch lässt sich der generierte Code kaum in den Visitoren wiedererkennen, was wiederum die Wartung und das Auffinden von Fehlern erschwert, die im generierten Code identifiziert wurden. Dieser Nachteil wird sogar noch verstärkt, wenn - wie oben beschrieben - anstelle von Strings der AST der Zielsprache aufgebaut wird. Dem kann zwar durch die Erstellung entsprechender Programmierschnittstellen (API, application programming interface) entgegengewirkt werden, die den Entwickler beim Aufbau komplexer AST-Strukturen unterstützen. Ganz beheben lässt sich das Problem dadurch jedoch nicht.

Im Gegensatz dazu sind Templates in Bezug auf Inhalt und Aufbau dem generierten Code recht ähnlich, da sie insbesondere die Anteile des Zielcodes in sich kapseln. Auf der anderen Seite stehen die Nachteile einer oft nur rudimentären Templatesprache und einer fehlenden Unterstützung für die Traversierung des ASTs. Im Abschnitt 5.2 wird deshalb ein Ansatz vorgestellt, der die Vorteile beider Ansätze kombiniert. Dieser baut auf der Template-Engine FreeMarker auf, die im Folgenden näher beschrieben wird.

Die Template-Engine von FreeMarker [FM] ist in Java umgesetzt und konnte dadurch gut in MontiCore integriert werden (siehe auch Kapitel 7). Bei den Templates handelt es sich um Textdateien mit der Endung `.ftl`, die in einem zweistufigen Prozess von der Template-Engine verarbeitet werden (siehe Abbildung 5.3). Im ersten Schritt wird das Template geladen und für die weitere Verarbeitung in ein Java-Objekt überführt. Dieses wird zusammen mit dem Datenmodell im zweiten Schritt abermals der Template-Engine übergeben, wodurch schließlich das Template ausgeführt und der eigentliche Zielcode erzeugt wird. Für das Datenmodell wird in FreeMarker mit der Klasse `SimpleHash` eine Art Hashtabelle zur Verfügung gestellt, die Stringbezeichner auf beliebige Objekte abbildet. Diese können später im Template über die Bezeichner in Einschüben und Kontrollstrukturen referenziert werden.

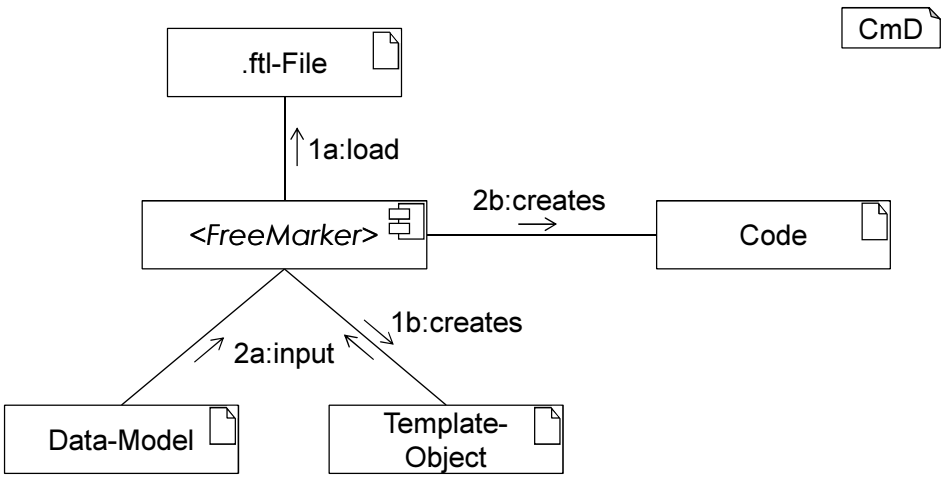


Abbildung 5.3: Kommunikationsdiagramm der Verarbeitung von FreeMarker-Templates

Um die festen Anteile eines Templates von der Templatesprache abzugrenzen, wird in FreeMarker die folgende Notation verwendet:

- **Einschübe** werden in Form von `${...}` notiert. Diese können die Bezeichner des Datenmodells enthalten und ermöglichen darüber ebenfalls den Zugriff auf Attribute und Methoden der hinterlegten Objekte mit Hilfe der Punkt-Notation wie in Java. Dabei ist zu beachten, dass das Ergebnis des Ausdrucks von der Template-Engine als String interpretiert wird, der bei der Verarbeitung an dieser Stelle eingesetzt wird.
- **Kontrollstrukturen** bestehen aus einem Start- (`<#...>`) und optional aus einem End-Tag (`</#...>`). Die verschiedenen Kontrollstrukturen, die FreeMarker bietet, werden anhand von Namen unterschieden, die direkt nach dem `#`-Zeichen angegeben werden. Start-Tags können darüber hinaus noch verschiedene, Tag-abhängige Ausdrücke als Parameter enthalten, in denen ebenfalls wie bei Einschüben auf die Objekte des Datenmodells sowie deren Attribute und Methoden zugegriffen werden kann.

Aufgrund des einheitlichen Zugriffs auf das Datenmodell in Einschüben und Kontrollstrukturen, können die Bezeichner aus Sicht des Templates als (untypisierte) Variablen verstanden werden. Der Einfachheit halber wird deshalb im Folgenden dieser Begriff verwendet.

Quellcode 5.4 zeigt ein einfaches Template für die Generierung einer Java-Klasse mit Attributen und zugehörigen get- und set-Methoden. Für dessen Verarbeitung muss der Name der zu generierenden Klasse als `c_name` zusammen mit einer Liste von Attribut-Objekten als `c_attributes` im Datenmodell hinterlegt werden (siehe Abbildung 5.5). Der Klassenname wird in Zeile 1 des Templates neben festen Anteilen der Zielsprache ausgegeben. Dies ist möglich, da es sich um einen einfachen String handelt. Die Liste der Attribute muss hingegen entsprechend aufgearbeitet werden. Dazu wird über diese mit Hilfe der FreeMarker-Kontrollstruktur `list` in Zeile 3-23 iteriert, wobei in jeder Iteration das aktuelle Attribut als Variable `attr` zur Verfügung steht (Zeile 3). Da die Sichtbarkeit eines Attributs im Datenmodell nicht als String vorliegt, wird dieser in Zeile 5-11 ermittelt und in der lokalen Variable `mod` hinterlegt. Wie das Beispiel zeigt, kann FreeMarker dabei in den Bedingungen boolesche Werte von Java auswerten. In Zeile 13-21 wird schließlich der Java-Code für das Attribut und die get- und set-Methoden erzeugt. Die hierfür notwendigen Daten liegen im Datenmodell bereits als Strings vor und können daher in den Einschüben direkt genutzt werden. Darüber hinaus bietet FreeMarker vor allem für die Ausgabe und Formatierung von Strings eine Reihe zusätzlicher Operationen an, die durch ein Fragezeichen getrennt an einen Ausdruck innerhalb von Einschüben oder Kontrollstrukturen angefügt werden. In Zeile 15 und 19 wird mit `cap_first` eine solche Operation genutzt, um den Attributnamen mit einem Großbuchstaben beginnen zu lassen.

Auch wenn FreeMarker den Zugriff auf beliebige Java-Objekte über das Datenmodell erlaubt, ist eine Erzeugung neuer Objekte innerhalb des Templates nicht direkt möglich. Diese können nur als Rückgabewerte von Methodenaufrufen über andere im Datenmodell vorhandene Objekte entstehen. Auch ein Aufruf statischer Elemente ist nur über entsprechend hinterlegte Instanzen möglich. Allerdings bietet FreeMarker einen Satz eigener Basistypen an, die für berechnete

```
public class ${c_name} {

  <#list c_attributes as attr>

    <#if attr.isPublic()>
      <#assign mod = "public">
    <#elseif attr.isProtected()>
      <#assign mod = "protected">
    <#else>
      <#assign mod = "private">
    </#if>

    private ${attr.getType()} ${attr.getName()};

    ${mod} ${attr.getType()} get${attr.getName()?cap_first}() {
      return this.${attr.getName()};
    }

    ${mod} void set${attr.getName()?cap_first}(${attr.getType()} o) {
      this.${attr.getName()} = o;
    }

  </#list>

}
```

FM

Quellcode 5.4: FreeMarker-Template

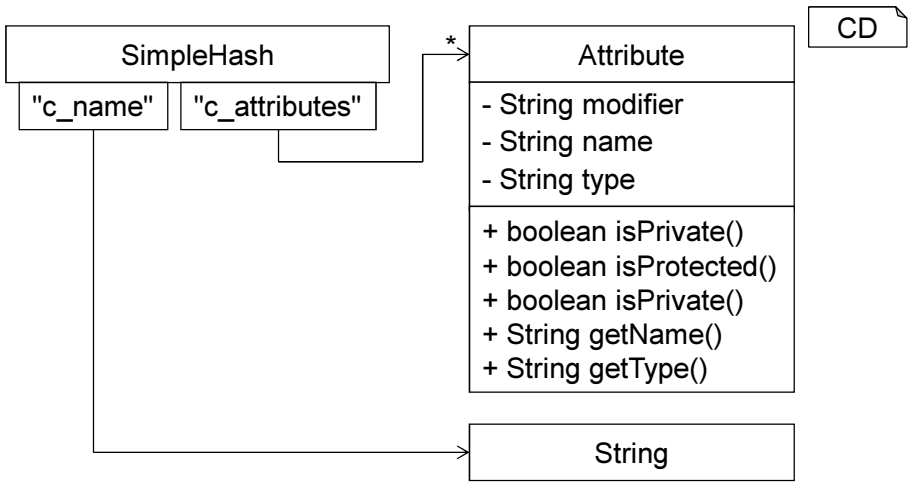


Abbildung 5.5: Datenmodell zum Template in Quellcode 5.4

Werte innerhalb eines Templates verwendet werden können (siehe Tabelle 5.6). Da Variablen in FreeMarker untypisiert sind, werden diese Typen anhand der zugehörigen Werte implizit ermittelt. Auch die Containertypen `Hash` (Hashtabellen) und `Sequence` (Listen) sind untypisiert, so dass verschiedene Typen in einem Container enthalten sein können. Für die Schlüssel von Hashtabellen sind jedoch ausschließlich Strings zugelassen.

Name	*Beispiele*
Skalare	
`String`	`"Hello World"`
`Number`	`42` oder `-3.14`
`Boolean`	Wahrheitswerte `true` und `false`
`Date`	`"06/22/1910"?date("MM/dd/yyyy")`
Container	
`Hash`	{`"name":"Alice", "size":180, "50":50`}
`Sequence`	`["foo", "bar", 3.14]`

Tabelle 5.6: Die FreeMarker Basistypen

Die Tabellen 5.7, 5.8 und 5.9 geben einen Überblick über die wichtigsten Kontrollstrukturen und Operationen von FreeMarker. Die Operationen können dabei nur auf solchen Werten ausgeführt werden, die sich als eine der Basistypen aus Tabelle 5.6 interpretieren lassen. Neben den aufgeführten Operationen für Strings und Container existieren auch einige wenige für die übrigen Basistypen. Diese beschränken sich jedoch im Wesentlichen auf die Darstellung bei der String-Konvertierung wie etwa die internationalisierte Ausgabe von Datumsangaben und sind daher hier nicht extra aufgeführt. Darüber hinaus bietet FreeMarker eine recht umfangreiche Infrastruktur für die Erstellung und Verarbeitung von XML-Dokumenten, wobei insbesondere die Generierung von HTML-Seiten unterstützt wird. Für ein vertiefendes Studium der FreeMarker Template-Engine und dessen Sprache steht mit [FM] eine ausführliche und vollständige Dokumentation zur Verfügung.

Tag	*Beschreibung*
`<#assign` *Name=Wert*`>`	Wertzuweisung an lokale Variablen, wobei auch mehrere durch Leerzeichen getrennte "Name = Wert"-Paare als Parameter angegeben werden können. Nicht existierende Variablen werden gleichzeitig angelegt.
`<#global` *Name=Wert*`>`	Definition globaler Variablen, die im Gegensatz zur `assign`-Anweisung für alle Templates gelten.
`<#if` *Bedingung_1*`>` ... `<#elseif` *Bedingung_2*`>` ... ... `<#elseif` *Bedingung_N*`>` ... `<#else>` ... `</#if>`	Führt nur den ersten zwischen den Tags stehenden Teil des Templates aus, dessen davor angegebene Bedingung als boolescher Wert `true` ausgewertet wird. Trifft dies für keine Bedingung zu, wird der Teil nach dem `else`-Tag ausgeführt. Es können mehrere `elseif` nach dem `if`-Tag angegeben werden, wobei `elseif` und `else` jeweils optional sind.

`<#switch` *Ausdruck_1*`>`
`<#case` *Ausdruck_2*`>`
  ...`<#break>`
...
`<#case` *Ausdruck_N*`>`
  ...`<#break>`
`<#default>`
  ...
`</#switch>`
	Neben dem `if`-Tag weitere Form der bedingten Ausführung. Dabei werden die Ausdrücke der `case`-Tags mit dem des `switch`-Tags verglichen. Sind die Ergebnisse der Ausdrücke gleich, wird der nachfolgende Teil des Templates bis zum nächsten `break`-Tag ausgeführt und danach keine weiteren Vergleiche mehr durchgeführt. Sind keine der Ausdrücke gleich, wird stattdessen der Teil nach dem optionalen `default`-Tag ausgeführt.

`<#list` *Liste* `as` *Element*`>`
  ...
`</#list>`
	Iteriert über eine Liste von Elementen. In jeder Iteration kann auf das aktuelle Element über den für *Element* vergebenen Namen zugegriffen werden.

`<#include` *Pfad*`>`	Fügt das Ergebnis eines anderen Templates in das vorliegende ein. Der als Parameter angegebene Ausdruck muss den vollständigen Pfad zum aufgerufenen Template inklusive Dateinamen und -endung ergeben. Im inneren Template kann auf globale Variablen und das Datenmodell zugegriffen werden.

`<#macro` *Name P_1 ... P_N*`>`
  ...
  `<#return>`
  ...
`</#macro>`
	Erstellung von benutzerdefinierten Anweisungen, die über den speziellen Tag `<@.../>` aufgerufen werden können. Dabei sind die Namen des Macros und der Parameter entsprechend anzugeben:
	`<@`*Name P_1=Wert_1 ... P_N=Wert_N* `/>`
	An der Stelle des Aufrufs wird der Template-Teil zwischen den `macro`-Tags mit den angegebenen Werten ausgeführt. Zusätzlich ist wie beim Aufruf auch im `macro`-Start-Tag die Angabe von Werten für Parameter möglich. Diese werden verwendet, wenn die Parameter beim Aufruf fehlen. Der optionale `return`-Tag beendet das Macro.

`<#function` *Name P_1 ... P_N*`>`
  ...
  `<#return` *Wert*`>`
  ...
`</#function>`
	Führt im Gegensatz zum `macro`-Tag eine Berechnung aus, dessen Ergebnis über den `return`-Tag an die Stelle des Aufrufs zurückgegeben wird. Aufgerufen wird eine Funktion über deren Namen und die Angabe von Werten für die Parameter *P_1 ... P_N*, z.B. in Einschüben:
	`${`*Name*`(`*Wert_1, ... Wert_N*`)}`

`<#noparse>...</#noparse>`	Der zwischen diesen Tags liegende Template-Teil wird unverarbeitet ausgegeben. Dies kann genutzt werden, um bei Überschneidungen zwischen Ziel- und Templatesprache eine fehlerhaften Verarbeitung zu verhindern.

`<#stop` *Grund*`>`	Beendet die Ausführung des Templates mit der als Parameter angegebenen Fehlermeldung.
`<#attempt>` ... `<#recover>` ... `</#attempt>`	Verhindert den Abbruch der Templateausführung beim Auftreten eines Fehlers im Teil nach dem `attempt`-Tag. Mögliche Ursachen sind u.a. Zugriffe auf nicht definierte Variablen, fehlerhafte Aufrufe des Datenmodells oder das Auftreten eines `stop`-Tags. Im Fehlerfall wird stattdessen der Teil nach dem `recover`-Tag ausgeführt.
`<#compress>...</#compress>`	Entfernt doppelte Leerzeichen und Zeilenumbrüche des eingefassten Template-Teils im Ergebnis.
`<#-- ... -->`	Enthalten Kommentare im Template, die nicht in der Ausgabe erscheinen. Auch wenn es sich im Grunde bei diesem speziellen Tag nicht um eine Kontrollstruktur handelt, findet er gerade in diesem Zusammenhang häufig Verwendung und wird deshalb hier mit aufgeführt.

Tabelle 5.7: FreeMarker: Übersicht der wichtigsten Kontrollstrukturen

Operation	*Beschreibung*
Listen	
`size`	Berechnet die Länge einer Liste.
`seq_contains(`*Element*`)`	Ergibt den booleschen Wert `true`, wenn das Element in einer Liste enthalten ist.
`first`, `last`	Liefert das erste bzw. letzte Element einer Liste zurück.
`seq_index_of(`*Element*`)`, `seq_last_index_of(`*Element*`)`	Gibt die Stelle des ersten bzw. letzten Vorkommens des Elements in einer Liste zurück (sonst -1).
`reverse`	Erzeugt aus einer Liste eine neue Liste mit umgekehrter Reihenfolge der Elemente.
`sort`	Verhält sich wie `reverse`, sortiert aber die Elemente in aufsteigender Reihenfolge. Dies wird nur für Elemente der Basistypen `String`, `Number` und `Date` unterstützt.
Hashtabellen	
`keys`	Liefert eine Liste aller Schlüssel einer Hashtabelle zurück.
`values`	Liefert eine Liste aller Werte einer Hashtabelle zurück.

Tabelle 5.8: FreeMarker: Übersicht der wichtigsten Container-Operationen

Operation	*Beschreibung*
`length`	Berechnet die Länge eines Strings.
`index_of(`*Teilstring*`)`, `last_index_of(`*Teilstring*`)`	Gibt die Stelle des ersten bzw. letzten Vorkommens des Teilstrings innerhalb eines Strings zurück (sonst -1).
`starts_with(`*Teilstring*`)`, `ends_with(`*Teilstring*`)`	Ergibt den booleschen Wert `true`, wenn der String mit dem Teilstring beginnt bzw. endet (sonst `false`).
`contains(`*Teilstring*`)`	Ergibt den booleschen Wert `true`, wenn der Teilstring in einem String enthalten ist (sonst `false`).
`matches(`*Ausdruck*`)`	Ergibt den booleschen Wert `true`, wenn ein String dem angegebenen regulären Ausdruck [Fri06] entspricht (sonst `false`). In diesem Fall wird die Sprache für reguläre Ausdrücke von Java verwendet [Hab04].
`upper_case`, `lower_case`	Wandelt alle Buchstaben eines Strings in die entsprechenden Groß- bzw. Kleinbuchstaben um.
`cap_first`, `uncap_first`	Ersetzt den ersten Buchstaben eines Strings durch den entsprechenden Groß- bzw. Kleinbuchstaben.
`capitalize`	Ersetzt den ersten Buchstaben jedes Wortes eines Strings durch den entsprechenden Großbuchstaben.
`replace(`*Teilstring*`, `*Ersatz*`)`	Ersetzt alle Vorkommen des Teilstrings in einem String durch den String des zweiten Parameters.
`trim`	Entfernt Leerzeichen am Anfang und Ende eines Strings.
`chop_linebreak`	Entfernt Zeilenumbrüche am Ende eines Strings.
`number`	Wandelt einen String in eine Zahl um.
`substring(`*von*`, `*bis*`)`	Extraktion eines Teilstrings. Der Parameter *bis* ist optional.
`split(`*Teilstring*`)`	Entfernt den Teilstring aus einem String, wobei der String an diesen Stellen in eine Sequenz von Teilstrings aufgeteilt wird.
`word_list`	Wie `split` bis darauf, dass ein String anhand von Leerzeichen aufgeteilt wird.

Tabelle 5.9: FreeMarker: Übersicht der wichtigsten String-Operationen

5.2 Aufbau eines Generators

Im Abschnitt 5.1.3 wurden die auf dem Visitor- und Templatekonzept basierenden Ansätze für die Umsetzung von Codegeneratoren behandelt, die jeweils ihre Vor- und Nachteile haben. Aus diesem Grund wurde im Rahmen dieser Arbeit ein integrierter Ansatz beider Konzepte entwickelt, der im Folgenden vorgestellt wird.

Einer der wichtigsten Vorteile von templatebasierten Ansätzen ist, dass Templates die zu generierenden Codeanteile kapseln. Dadurch ist darin die Struktur des Generats leichter erkennbar

als dies bei den visitorbasierten Ansätzen der Fall ist, was wiederum die Weiterentwicklung, Fehlersuche und Wartung von Codegeneratoren deutlich erleichtert. Insbesondere führt diese Kapselung dazu, dass Modifikationen oder Erweiterungen des generierten Codes in vielen Fällen allein auf Basis der Templates möglich sind. Aus diesen Gründen werden Templates im vorliegenden Ansatz als Ausgangspunkt und Hauptbestandteil eines Codegenerators betrachtet.

Die Templates basieren auf der Template-Engine FreeMarker, die bereits in Abschnitt 5.1.3 zusammen mit einem ersten Beispiel in Quellcode 5.4 vorgestellt wurde. Dieses Beispiel ist noch relativ einfach aufgebaut, da fast alle Daten in mehr oder weniger direkt verwendbaren Strings zur Verfügung stehen, aus denen sich das Generat zusammensetzt. Die notwendige Vorberechnung für die Sichtbarkeit zeigt jedoch, dass schon einfache Datenaufbereitungen die festen Anteile eines Templates in den Hintergrund rücken lassen, so dass sich das Generat nur noch schwer aus dem Template ablesen lässt. Aus diesem Grund empfiehlt es sich, diese zusätzlichen Berechnungen an anderer Stelle zu kapseln. Dazu könnte z.B. ein weiteres Template dienen, das vor dem eigentlichen Template ausgeführt wird. Allerdings ist die Templatesprache von FreeMarker im Wesentlichen auf die Verarbeitung einfacher Datentypen wie Strings ausgelegt und stößt somit bei komplexeren Berechnungen schnell an ihre Grenzen. Deshalb ist es häufig ratsamer, für solche Berechnungen auf die umfangreiche Infrastruktur höherer Programmiersprachen zurückzugreifen, die mit gut ausgebauten Bibliotheken und komfortablen Entwicklungsumgebungen deutlich mehr Möglichkeiten bieten. Auch beinhalten diese im Allgemeinen ein starkes Typsystem, das zur Vermeidung von Fehlern beiträgt. Im vorliegenden Fall bietet sich hierfür Java an, da sowohl MontiCore und als auch FreeMarker darauf basieren. Darüber hinaus erfordert die Nutzung von FreeMarker generell die Programmierung einiger Zeilen Java-Code. Diese sind nötig, um die Templates zu laden und zusammen mit dem Datenmodell der Template-Engine zu übergeben (siehe Abschnitt 5.1.3, Abbildung 5.3).

Die Ausführung von Java-Code im Rahmen der Templateverarbeitung ist in FreeMarker auf zwei Arten möglich: entweder über Methodenaufrufe auf im Datenmodell abgelegten Objekten oder indem die Daten vor der Templateausführung berechnet und direkt über das Datenmodell zur Verfügung gestellt werden. In jedem Fall beginnt die Templateverarbeitung mit der Vorbereitung des Datenmodells, obwohl eigentlich die Templates selbst bei der Codegenerierung im Vordergrund stehen sollten. Da in der vorliegenden Arbeit aus UML/P-Modellen Code generiert wird, bietet es sich darüber hinaus an, auf Basis der abstrakten Syntax der Modelle die Templates zu erstellen. In Fällen, in denen die im AST abgelegten Daten für die Programmierung der Templates ausreichend sind, wäre demnach eine Vorbereitung des Datenmodells unnötig, solange der AST dort bereits auf anderem Wege abgelegt wurde. Auf diese Weise würde sich die Programmierung eines Generators auf die Templates beschränken und es müsste nur für die erwähnten komplexen Berechnungen auf Java-Code zurück gegriffen werden.

Der im Rahmen dieser Arbeit entwickelte Ansatz greift diese Idee der Templates als zentrales Artefakt der Codegenerierung auf. Abbildung 5.10 illustriert dazu die wichtigsten Komponenten und deren Beziehungen untereinander. Wie bereits in Abschnitt 5.1.2 erläutert, erzeugt das MontiCore-Framework anhand der Grammatikbeschreibung der einzelnen Sprachen einen Parser,

der ein konkretes Modell in den AST überführt. Dieser steht den Templates automatisch zur Verfügung, wodurch ein manueller Aufbau des Datenmodells vor dem Templateaufruf entfällt. Darüber hinaus wurde die Möglichkeit geschaffen, aus den Templates heraus spezielle Java-Klassen aufzurufen, ohne dass diese bereits im Datenmodell vorliegen müssen. Diese sogenannte Kalkulatoren (engl.: Calculators) können bei Bedarf eingesetzt werden, um komplexe Berechnungen aus einem Template auszulagern und stattdessen in Java zu implementieren. Zusätzlich entsteht dadurch eine Referenz vom Template auf die dafür notwendigen Datenberechnungen und nicht umgekehrt, wie es normalerweise bei FreeMarker und auch anderen Template-Engines der Fall ist. Insgesamt bilden somit die Templates den alleinigen Ausgangspunkt für die Codegenerierung. Diese werden schließlich der FreeMarker-Engine übergeben, die daraus den Code erzeugt.

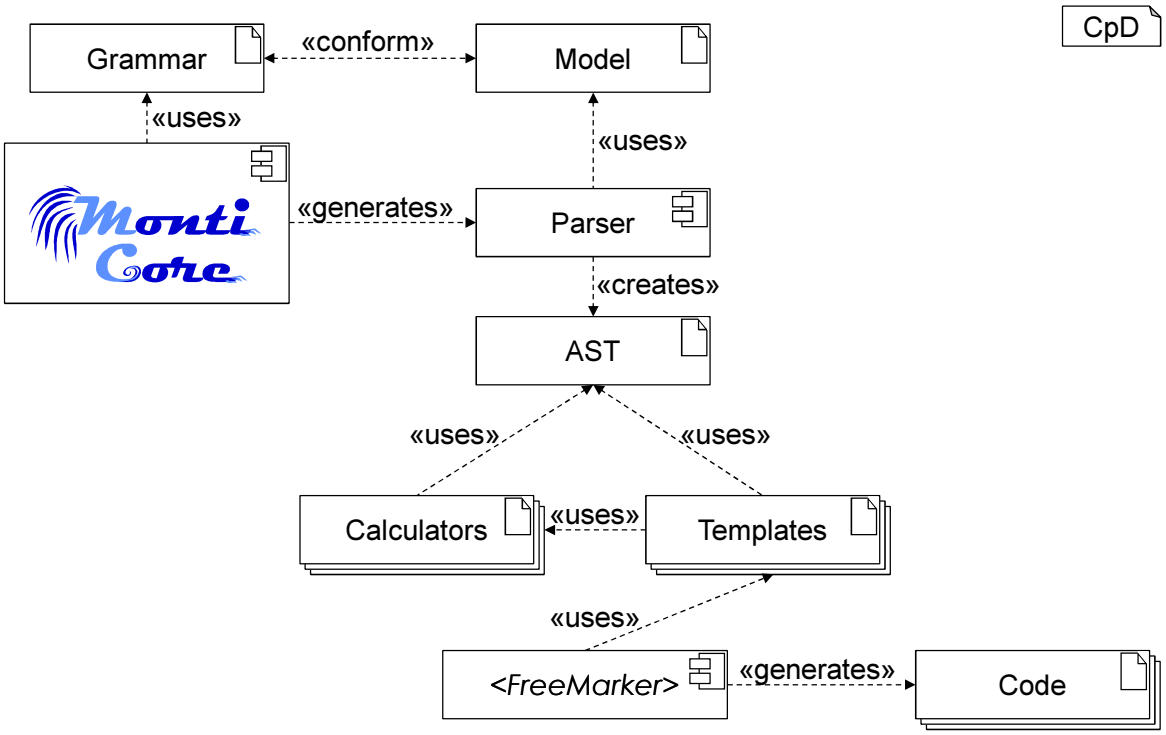


Abbildung 5.10: Übersicht der Komponenten bei der Generierung über das UML/P-Framework

Ein Generator setzt sich demnach im vorliegenden Ansatz aus einer Menge von Templates und ggf. einigen Kalkulatoren zusammen. Um die Notwendigkeit für den Einsatz von Kalkulatoren zu reduzieren, werden darüber hinaus im AST zusätzliche Methoden bereitgestellt. Diese entsprechen im Wesentlichen den vom MontiCore-Framework generierten get-Methoden für den Zugriff auf die Attribute eines AST-Knotens, liefern aber im Gegensatz dazu einen das Attribut repräsentierenden String zurück. Falls es sich bei dem Attribut um eine Menge von Knoten handelt, wird ein entsprechender Komma-separierter String oder durch Anhängen des Postfixes “List” eine Liste von Strings mit den Einzel-Elementen erzeugt. Gekennzeichnet sind diese Methoden durch das Präfix

"print", um sie von den normalen get-Methoden abzugrenzen. Auf diese Weise stehen sämtliche Daten des ASTs auch als Strings zur Verfügung, die sich leichter im Template verarbeiten lassen.

Bei der Codegenerierung entsteht häufig die Situation, dass für ein bestimmtes Element der Ausgangssprache ein oder mehrere Artefakte der Zielsprache erzeugt werden sollen. Ein einfaches Beispiel ist die Generierung von Java-Code aus den in Klassendiagrammen modellierten Klassen. Aus diesen muss nicht notwendigerweise jeweils nur eine entsprechende Klasse im Zielsystem entstehen, sondern es können unter anderem Factories für deren Instanziierung, Mockklassen, Codeinstrumentierung oder für den Zugriff auf Attribute zusätzlich get- und set-Methoden generiert werden. Auf diese Weise ist es möglich, die abstrakte Modellierung zu einem lauffähigen System zu ergänzen und andere Anforderungen wie Testbarkeit automatisch durch die Generatoren sicher zu stellen (siehe auch Kapitel 6). Vereinfacht ausgedrückt entsteht somit in den Generatoren eine Abbildung, die Elemente der Ausgangssprache in Elemente der Zielsprache überführt. Nicht zuletzt aus diesem Grund sind in der UML/P alle Sprachelemente als einzelne Regeln in der Grammatik gekapselt, für die das MontiCore-Framework jeweils einen eigenen Typ von AST-Knoten in der abstrakten Syntax erzeugt (siehe Abschnitt 5.1.2). Bezogen auf die abstrakte Syntax wird somit ein Elementtyp durch einen bestimmten Typ von AST-Knoten repräsentiert, auf den in den Generatoren reagiert werden kann. Vor dem Hintergrund eines solchen Szenarios besitzt die Idee, Codegeneratoren mit Hilfe von Visitoren zu entwickeln, einen deutlichen Vorteil gegenüber dem templatebasierten Ansatz (vgl. Abschnitt 5.1.3). Während in den Templates die notwendigen Elemente manuell aus dem AST extrahiert werden müssen, sind Visitoren auf eben diese Reaktion auf spezifische Typen von AST-Knoten ausgelegt und abstrahieren damit von der Traversierung des ASTs.

In dem als Teil dieser Arbeit entwickelten Codegenerierungsframework, das vom UML/P-Framework verwendet wird, wurde deshalb das Konzept der Visitoren mit dem templatebasierten Ansatz kombiniert. Dazu werden die Templates nicht über das FreeMarker-Framework direkt aufgerufen, sondern in Form von Parameter-Paaren dem UML/P-Framework übergeben (siehe Kapitel 7). Jedes Parameter-Paar besteht jeweils aus dem Namen eines Templates und einer AST-Klasse, die durch den Namen der zugehörigen Grammatikregel identifiziert wird. Liegt nach dem Parsen eines konkreten Modells eine entsprechende Instanz der Klasse im AST vor, werden die jeweils zugeordneten Templates vom Generierungsframework aufgerufen, wobei der AST-Knoten über das Datenmodell direkt abrufbar ist. Dadurch entfällt die Notwendigkeit, in den Templates über den gesamten AST zu den relevanten Teilen zu navigieren, sondern es kann jeweils ein geeigneter Einstiegspunkt gewählt werden. Gleichzeitig ermöglicht dies die Templateausführung, ohne vorab manuell in Java ein Datenmodell vorzubereiten und die FreeMarker-Engine aufzurufen (vgl. Abschnitt 5.1.3). Insgesamt reduziert sich damit die Programmierung eines Generators auf die Templates, mit der optionalen Möglichkeit, komplexe Berechnungen in Kalkulatoren auszulagern.

Das Generierungsframework sorgt dafür, dass in jedem Template zumindest folgende zwei Variablen automatisch zur Verfügung stehen:

- `ast`: Verweist auf den aktuellen AST-Knoten. Über dessen API, die sich anhand der in Abschnitt 5.1.2 beschriebenen Ableitung aus der Grammatikdefinition der Sprachen ergibt, kann auf den gesamten AST zugegriffen werden.
- `op`: Verweist auf eine Instanz der Klasse `TemplateOperator`, die u.a. Infrastruktur für den Aufruf von Kalkulatoren, die Definition von Variablen und den Aufruf von Templates enthält (siehe Abschnitt 7.4). Das Ergebnis von Templateaufrufen kann sowohl in andere Templates eingebettet als auch als eigenständige Dateien abgelegt werden.

Als ein erstes Anwendungsbeispiel wird zu jeder Klasse in einem Klassendiagramm neben einer entsprechenden Java-Klasse eine Factory generiert, über die Instanzen dieser Klasse erstellt werden können [GHJV95]. Quellcode 5.11 zeigt das Template für die Generierung der Factory-Klasse. Dieses wird auf AST-Knoten der Grammatikregel `CDClass` aufgerufen (siehe Quellcode C.5 im Anhang C.4). Mit Hilfe der print-Methoden kommt das Template allein mit dem AST als Datenmodell aus, ohne dass zusätzliche Berechnungen notwendig sind. Die Kontrollstrukturen beschränken sich auf die Iteration über Attribut-Listen. Insgesamt ist dadurch das gewünschte Ergebnis der Generierung im Template recht gut zu erkennen.

Das Ergebnis des Templates soll in einer Datei abgelegt werden. Dazu wird mit Quellcode 5.12 dem Template ein weiteres vorangestellt, das das beschriebene Factory-Template aufruft. Für solche Aufrufe innerhalb eines Templates bietet der Template-Operator zwei Methoden an:

- **Templateeinbettung (`includeTemplates`)**: Der Methode werden als Parameter der Name des aufzurufenden Templates und der AST-Knoten übergeben, der bei dessen Verarbeitung verwendet werden soll. Im Gegensatz zum `include`-Tag von FreeMarker verwendet der Aufruf den Modelpath von MontiCore [Kra10], so dass das Template über dessen voll-qualifizierten Namen statt des vollständigen Pfades referenziert werden kann (vgl. Abschnitt 5.1.3, Tabelle 5.7). Darüber hinaus kann es sich bei beiden Parametern jeweils um eine Liste von Templatenamen bzw. AST-Knoten handeln, über die automatisch iteriert wird. Auf diese Weise können mehrere Templateaufrufe durch eine Anweisung dargestellt werden. Das Ergebnis dieser Aufrufe wird an die Stelle der Anweisung zurück gegeben. Erfolgte der Aufruf in Form eines Einschubs, wird das Ergebnis im Template eingefügt, kann aber alternativ auch in einer Variablen gespeichert werden.
- **Templateaufruf (`callTemplate`)**: Legt im Gegensatz zur `includeTemplates`-Methode das Ergebnis des Templateaufrufs in einer Datei ab. Der Dateiname und optional dessen Endung werden der Methode als Parameter nach dem Templatenamen übergeben. Fehlt die Angabe einer Dateiendung, wird der Standardwert des Template-Operators verwendet. Als letzter Parameter folgt der zu verwendende AST-Knoten (siehe auch Abschnitt 7.4).

In Quellcode 5.12 wird in Zeile 5-8 die `callTemplate`-Methode verwendet, um das Factory-Template aufzurufen und als eigene Datei abzulegen. Dies zeigt, dass sich ein Template wiederum aus mehreren Subtemplates zusammensetzen kann. Das jeweils oberste Template einer solchen Hierarchie wird im Folgenden als Haupttemplate bezeichnet. Diese bilden den Startpunkt für die

```
package ${ast.printPackage()};

// Singleton
public class ${ast.printName()}Factory {

  protected static ${ast.printName()}Factory factory = null;

  protected ${ast.printName()}Factory() {
  }

  // public access through static method (Singleton)
  <#if ast.getCDConstructors()?size != 0>
    <#list ast.getCDConstructors() as constr>
      public static ${ast.printName()} create(${constr.printCDParameters()}) {
        if (factory == null) {
          factory = new ${ast.printName()}Factory();
        }
        return factory.doCreate(${constr.printCDParameterNames()});
      }

      protected ${ast.printName()} doCreate(${constr.printCDParameters()}) {
        return new ${ast.printName()}(${constr.printCDParameterNames()});
      }
    </#list>
  <#else>
    public static ${ast.printName()} create() {
      if (factory == null) {
        factory = new ${ast.printName()}Factory();
      }
      return factory.doCreate();
    }

    protected ${ast.printName()} doCreate() {
      return new ${ast.printName()}();
    }
  </#if>

}
```

FM

Quellcode 5.11: Factory-Template

```
<#if !ast.hasStereotype("singleton") && !ast.getModifier().isAbstract()>

  ${op.setValue("package", ast.printPackage())}

  ${op.callTemplate(
      "mc.umlp.cd.Factory",
      package + "." + ast.printName() + "Factory",
      ast)}
</#if>
```

FM

Quellcode 5.12: FactoryMain-Template

Generierung, indem sie als Parameter an das UML/P-Framework übergeben werden. Durch diese Form der Übergabe können die Haupttemplates prinzipiell unabhängig voneinander angegeben und vom Framework verarbeitet werden, so dass jedes Haupttemplate als eigener Generator betrachtet

werden kann. Dabei kapseln die Haupttemplates im Allgemeinen die Dateierzeugung, wobei nicht notwendigerweise nur ein `callTemplate`-Aufruf enthalten sein muss. Ein Template kann somit auch mehrere Dateien erzeugen und auf die gleiche Weise wiederum mehrere Generatoren zusammenfassen.

Neben der Dateierzeugung kapselt das Haupttemplate für die Factory-Generierung außerdem die Bedingungen, unter denen die Factory generiert werden soll (Zeile 1). Darüber hinaus wird in Zeile 3 eine neue Variable `package` über die Template-Operator-Methode `setValue` definiert, die im Gegensatz zum `assign`-Tag von FreeMarker ebenfalls in den Subtemplates gültig ist. Zeile 1 in Quellcode 5.11 könnte somit durch diese Variable ersetzt werden. Auf diese Weise können auch umfangreichere Vorberechnungen in einem Template gekapselt werden, um die Subtemplates aus den oben genannten Gründen auf möglichst wenige Kontrollstrukturen zu beschränken. Damit ist es prinzipiell möglich, lesbare und wartbare Generatoren nur auf Basis von Templates zu programmieren. Auch eigene Bibliotheken können mit Hilfe von Funktionen und Makros aufgebaut werden.

Für komplexe Berechnungen bietet das Framework dem Entwickler die Möglichkeit, selbst geschriebene Kalkulatoren zu nutzen. Mit diesen erschließt sich den Templates der vollständige Sprachumfang und die Infrastruktur von Java. Im Gegensatz zu der Templatesprache von FreeMarker ist es so möglich, beliebige neue Objekte zu erzeugen und in den Templates zur Verfügung zu stellen. Selbst Modell-zu-Modell-Transformationen auf dem AST als Vorbereitung für die Templates sind auf diese Weise möglich. Bei den Kalkulatoren handelt es sich um gewöhnliche Java-Klassen, die die abstrakte Klasse `TemplateCalculator` implementieren (siehe Abschnitt 7.4). Diese wird vom Codegenerierungsframework zur Verfügung gestellt. Ein konkreter Kalkulator muss diese Klasse mit dem erwarteten AST-Typ parametrisieren sowie die boolesche Methode `calc` implementieren.

Quellcode 5.13 zeigt die Anwendung eines Kalkulators. Diese werden über die `callCalculator`-Methode des Template-Operators aufgerufen, indem der voll-qualifizierte Name der Kalkulator-Klasse übergeben wird (Zeile 1). Das Generierungsframework ruft daraufhin die `calc`-Methode des entsprechenden Kalkulators auf und übergibt dieser den aktuellen AST-Knoten sowie den Template-Operator. Dadurch kann im Kalkulator auf dieselben Daten wie im Template zugegriffen werden. Auch die Erstellung neuer Variablen ist über den Template-Operator auf die gleiche Weise möglich, so dass kein Umdenken beim Wechsel zwischen Template- und Kalkulatorprogrammierung notwendig ist. Die Variablen stehen nach dem Kalkulatoraufruf im Template wie gewohnt zur Verfügung.

Das Template in Quellcode 5.13 generiert zu einem Attribut einer Typdefinition in einem Klassendiagramm entsprechende get- und set-Methoden im Zielsystem. Wie ein Ausschnitt des aufgerufenen Kalkulators in Quellcode 5.14 zeigt, ist diese Aufgabe komplexer als es auf den ersten Blick erscheinen mag. So muss berücksichtigt werden, ob nicht bereits entsprechende Methoden in dem Typ selbst oder einem der Supertypen enthalten sind. Darüber hinaus darf für finale Attribute oder Konstanten keine set-Methode generiert werden und boolesche Attribute werden nicht über eine get- sondern eine is- Methode abgerufen, um nur einige der Voraussetzungen zu nennen. Die

```
1  <#if op.callCalculator("mc.umlp.cd.codegen.CDGetterSetterCalculator")>
2    <#if addGetter>
3      ${ast.printModifier()} ${ast.printType()} ${getterName}() {
4        return ${ast.printName()};
5      }
6    </#if>
7
8    <#if addSetter>
9      ${ast.printModifier()} void ${setterName}(${ast.printType()} ${ast.printName()}) {
10       this.${ast.printName()} = ${ast.printName()};
11     }
12   </#if>
13 </#if>
```

FM

Quellcode 5.13: GetterSetter-Template

Ausführung von Templates oder Teilen davon kann somit von bestimmten Konstellationen im Ausgangsmodell abhängen. Da es sich oft anbietet, eine entsprechende Analyse zusammen mit der Datenaufbereitung durchzuführen, ist der Rückgabewert der `calc`-Methode von Kalkulatoren ein boolescher Wert, der vom Framework an die Stelle des Aufrufs im Template zurückgegeben wird. Wird der `callCalculator`-Aufruf wie im Beispiel eingebunden, kann dieser Rückgabewert genutzt werden, um die weitere Ausführung des Templates von dem erfolgreichen Abschluss des Kalkulators abhängig zu machen. Auf diese Weise wird gleichzeitig sichergestellt, dass die im Kalkulator definierten Variablen bei der Ausführung des Templates vorliegen.

In den bisherigen Beispielen wurden die Namen von Templates und Kalkulatoren direkt bei den Aufrufen des Template-Operators angegeben. Es ist jedoch auch möglich, diese offen zu lassen, indem stattdessen eine Variable eingesetzt wird. Diese Variablen können in einem beliebigen übergeordneten Template mit dem gewünschten Namen belegt werden. Auf diese Weise entstehen Erweiterungspunkte in Templates, wofür sich insbesondere die Methode `inlcudeTemplates` anbietet. Quellcode 5.15 zeigt dies am Beispiel des Templates für die Generierung von Klassen aus einem Klassendiagramm. Hier wurde in den Zeilen 15-19 für jedes Attribut einer Klasse ein Erweiterungspunkt definiert. Welche Subtemplates an dieser Stelle jeweils integriert werden, wird im Haupttemplate in Quellcode 5.16 in den Zeilen 1-4 vor dem Aufruf des Klassentemplates festgelegt. Laut Zeile 1 soll z.B. für Attribute ein Template für die Ausgabe des Attributs selbst und das bereits vorgestellte Template für die get- und set-Methoden aufgerufen werden. Dieses Vorgehen ermöglicht die flexible Zusammenstellung und Erweiterung von Templates. Im Grunde stellt dabei das Haupttemplate eine Art Konfigurationsdatei des Generators dar, die die Zusammensetzung der Subtemplates und damit des zu generierenden Artefakts bestimmt. Soll etwa ein anderer Generator Klassen ohne die automatische Erzeugung von get- und set-Methoden erstellen, genügt es, ein neues Haupttemplate auf Basis von Quellcode 5.16 zu schreiben und die 1. Zeile entsprechend zu modifizieren.

Das oben beschriebene Vorgehen erlaubt es somit, konfigurable Generatoren auf Basis von Templates zu erstellen. Im Sinne der Wiederverwendbarkeit können die Templates darüber hinaus

Java

```java
package mc.umlp.cd.codegen;

public class CDGetterSetterCalculator extends TemplateCalculator<ASTCDAttribute> {

  @Override
  public boolean calc(ASTCDAttribute ast, TemplateOperator op) {
    Boolean addGetter = false, addSetter = false;
    String getPrefix = "get";

    // getter- and setter-generation only for non-static/-private attributes
    if (!ast.getModifier().isPrivate() && !ast.getModifier().isStatic()) {
      // modify getPrefix for boolean type
      if (ast.getType() instanceof ASTPrimitiveType
          && ((ASTPrimitiveType) ast.getType()).isBoolean()) {
        getPrefix = "is";
      }
      // check if getter or setter already exist in Classdiagram
      ...
    }
    // extend template-datamodel
    if (addGetter || addSetter) {
      op.setValue("addGetter", addGetter);
      op.setValue("addSetter", addSetter);
      op.setValue("getterName", getPrefix + ast.getName().toUpperCase());
      op.setValue("setterName", "set" + ast.getName().toUpperCase());
      return true;
    }
    return false;
  }

}
```

Quellcode 5.14: Kalkulator des GetterSetter-Templates (Auszug)

FM

```
package ${ast.printPackage()};

<#list ast.printImportList() as i>
  import ${i};
</#list>

${ast.printDefaultModifier()} class ${ast.printName()}${ast.printTypeParameters()}
<#if ast.printSuperclassList()?size != 0>
  extends ${ast.printSuperclassList()[0]} <#-- only single-inheritance -->
</#if>
<#if ast.printInterfaces() != "">
  implements ${ast.printInterfaces()}
</#if>
{
  ${op.includeTemplates(t_class, ast)}
  ${op.includeTemplates(t_attr, ast.getCDAttributes())}
  ${op.includeTemplates(t_constr, ast.getCDConstructors())}
  ${op.includeTemplates(t_meth, ast.getCDMethods())}
  ${op.includeTemplates(t_assoc, ast.getCDAssociations())}
}
```

Quellcode 5.15: Class-Template

```
1 ${op.setValue("t_attr", ["mc.umlp.cd.Attribute", "mc.umlp.cd.GetterSetter"])}
2 ${op.setValue("t_meth", "mc.umlp.cd.Method")}
3 ${op.setValue("t_constr", "mc.umlp.cd.Constructor")}
4 ${op.setValue("t_assoc", "mc.umlp.cd.Association")}
5
6 ${op.callTemplate(
7     "mc.umlp.cd.Class",
8     ast.printPackage() + "." + ast.printName(),
9     ast)}
```

FM

Quellcode 5.16: ClassMain-Template

in Form von Bibliotheken gesammelt werden, um diese für verschiedene Projekte zu neuen Generatoren zu kombinieren. Das Hinzufügen, Modifizieren oder Entfernen einzelner Templates reicht jedoch nicht immer aus, um die gewünschte Funktionalität im generierten Code umzusetzen. Dies betrifft Fälle, in denen sich dadurch der Zugriff auf den generierten Code ebenfalls ändert. Ein Beispiel dafür ist die Generierung von get-/set-Methoden. Im Modell sind diese Methoden nicht aufgeführt, so dass danach auf die Attribute direkt zugegriffen werden kann. Auf diesen direkten Zugriff können sich andere Modelle wie z.B. Statecharts beziehen, aber auch der an verschiedenen Stellen in den Modellen eingebettete Code. Dennoch ist die Verwendung solcher Methoden im Zielsystem durchaus sinnvoll, um darüber etwa Zugriffe auf Attribute für Testfälle zu protokollieren (siehe Abschnitt 6.4). Wird nun das Template aus Quellcode 5.13 für die Bereitstellung von get- und set-Methoden dem Generator hinzugefügt, müssen diese Stellen mit berücksichtigt werden. Dafür gibt es verschiedene Möglichkeiten:

- **Konvention in den Modellen**: Der Zugriff auf die Attribute könnte schon in den Modellen in Form von get- und set-Methoden ausgedrückt werden. Dadurch wären die Modelle aber von Generatoren abhängig, die diese Methoden erstellen, und dies somit weiterhin kein konfigurabler Aspekt des Generators. Darüber hinaus ist für Erstellung und Verständnis der Modelle die Kenntnis dieser Konvention erforderlich.
- **Modifikation der Templates**: Neben der Hinzunahme des Templates für die Generierung der get- und set-Methoden müssten auch gleichzeitig die Templates ausgetauscht werden, die Zugriffe auf Attribute realisieren. Dies kann aber unter Umständen sehr aufwändig sein. So sind in dem Beispiel nicht nur die Templates für die Generierung aus Klassendiagrammen betroffen, sondern auch die aus den anderen Sprachen der UML/P. Darüber hinaus wird der in den Modellen eingebettete Code oft direkt für das generierte Zielsystem übernommen, sofern es sich bei beiden um die gleiche Sprache handelt. In diesen Fällen existieren für den eingebetteten Code keine Templates, so dass erst ein entsprechender (vollständiger) Satz erstellt werden muss. Um am Ende die Generierung der get- und set-Methoden möglichst einfach an- und abschalten zu können, muss bei der Erstellung der Templates darauf geachtet werden, die Generierung von Attributzugriffen in nur einem Subtemplate zu kapseln.

- **Transformation der Ausgangsmodelle**: Anstatt die Generierung der get- und set-Methoden als Template umzusetzen, könnten die Modelle entsprechend transformiert werden. Dabei kann es jedoch vorkommen, dass verschiedene Sprachen und damit ASTs berücksichtigt werden müssen, so dass im Prinzip mehrere Transformationen zu implementieren sind. Diese können entweder während der Templateverarbeitung über einen Kalkulator oder noch vor der Codegenerierung als eigener Workflow ausgeführt werden. Bei letzterem handelt es sich um einen speziellen Erweiterungsmechanismus von MontiCore [Kra10].
- **Transformation des Generats**: Als Alternative zur Transformation der Modelle kann auch das Ergebnis der Codegenerierung transformiert werden. Der Vorteil ist, dass es sich hier oft nur noch um eine Sprache handelt und sich dadurch die Transformation unabhängig von der Sprache der Ausgangsmodelle anwenden lässt. Darüber hinaus wird dabei der eingebettete Code und die Ergebnisse anderer Templates automatisch mit transformiert. Dieses Vorgehen setzt allerdings voraus, dass ein Parser für die Zielsprache existiert.

Die Generierung von get- und set-Methoden ist nicht das einzige Beispiel, bei dem dieses Problem auftritt. Weitere Beispiele sind die Verwendung von Factories für die Typinstanziierung oder auch einfach nur die Änderung von voll-qualifizierten Typnamen, wenn die Paketstruktur der Modelle nicht der des Generats entspricht. In solchen Fällen ist oft die nachträgliche Transformation des Generats das einfachste und effektivste Vorgehen, um einen Generator für das gewünschte Ergebnis zu implementieren. Aus diesem Grund wurde im Rahmen dieser Arbeit dafür zusätzliche Infrastruktur geschaffen, zumal die bisher vorgestellte Infrastruktur die anderen der oben aufgeführten Lösungsvarianten prinzipiell nicht ausschließt. Da bei dieser abschließenden Transformation der generierte Code nur noch angepasst werden soll, ohne dass dabei zusätzliche Funktionalität entsteht, wird dieser Schritt im Folgenden als Refactoring[2] bezeichnet [OJ90, Fow99].

Abbildung 5.17 zeigt die wichtigsten Komponenten, die an dem Refactoring des Generats beteiligt sind. Der Vorgang schließt sich direkt an die in Abbildung 5.10 dargestellte Codegenerierung aus den Templates an, die hier nur vereinfacht dargestellt ist. Der vom Generator erzeugte Code wird noch vor der Ablage im Dateisystem an den Parser weitergereicht, der den entsprechenden AST aufbaut. Für die Umsetzung der Refactorings bietet sich der visitorbasierte Ansatz an, wobei das Framework prinzipiell beliebige Lösungen zulässt. Im Detail wird dies in Abschnitt 7.5 beschrieben. Der AST wird nach der Modifikation durch die `Refactorizer`-Komponente über einen Pretty-Printer schließlich ausgegeben.

Das abschließende Refactoring des Generats ist wohlgemerkt nur eine weitere Möglichkeit des Frameworks, die in manchen Fällen schneller zum gewünschten Ergebnis führen kann. Bei einem vollständigen und gut strukturierten Satz von Templates, bei dem auch der eingebettete Code in Form von Templates in das Generat überführt wird, können solche Modifikationen auch nur auf Basis der Templates effizient durchgeführt werden.

[2] Im Deutschen sind in einigen Fällen auch die Begriffe “Refaktorierung” oder “Restrukturierung” anzutreffen. Im Allgemeinen wird jedoch die englische Bezeichnung verwendet.

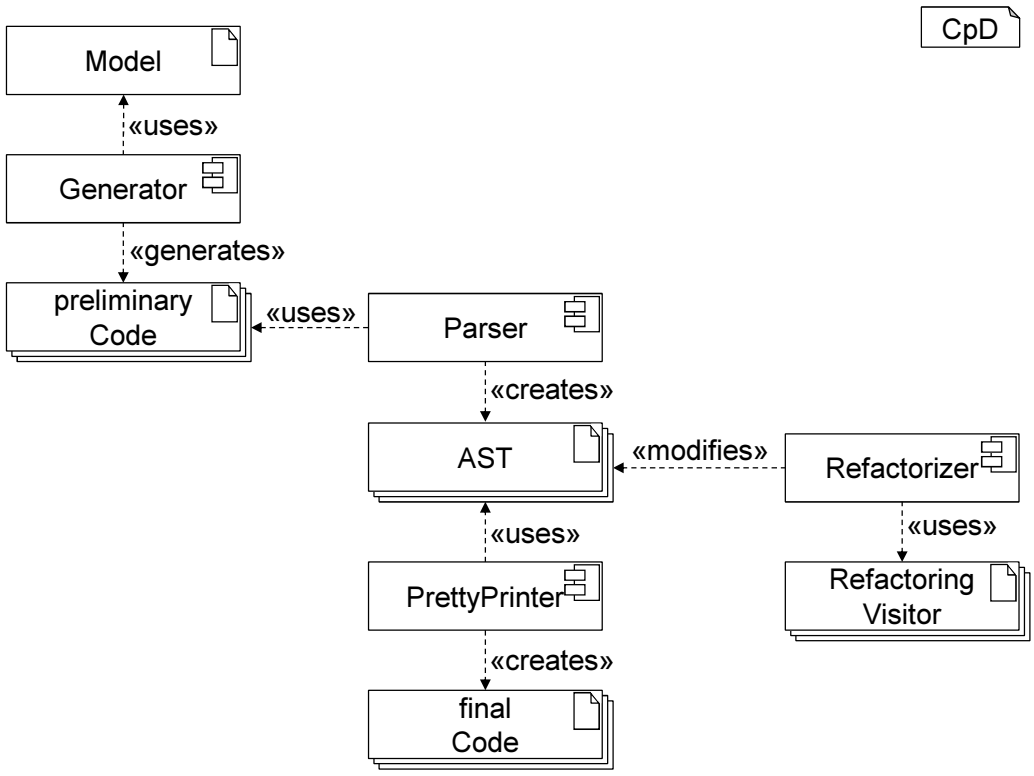


Abbildung 5.17: Refactoring-Komponenten bei der Generierung über das UML/P-Framework

5.3 Vorgehensweise bei der Implementierung eines Generators

In den meisten Template-Engines wie Velocity [GC03] oder FreeMarker [FM] werden die festen Anteile ohne weitere Überprüfung auf syntaktische Korrektheit übernommen, so dass sie in einer beliebigen Sprache formuliert werden können. Einerseits ist dadurch ein flexibler Einsatz dieser Engines unabhängig von der zu generierenden Zielsprache möglich. Andererseits fehlt es aber auch an Editorunterstützung in Entwicklungsumgebungen und damit an Komfortfunktionen wie Autovervollständigung oder Syntaxhervorhebung innerhalb der festen Anteile. Dies führt dazu, dass selbst einfachste syntaktische Fehler erst nach der Verarbeitung sichtbar werden. Einschübe und Kontrollstrukturen sind hingegen in einer eigenen Sprache formuliert, für die oft eine Editorunterstützung existiert.

Abgesehen von geringfügigen Modifikationen eines Generators ist es neben der Fehleranfälligkeit auch im Allgemeinen schwierig, das gewünschte Generat direkt in den Templates zu verallgemeinern. Stattdessen hat sich insbesondere für die Erstellung neuer sowie für umfangreiche Erweiterungen bestehender Generatoren folgendes Vorgehensmodell bewährt:

1. **Erstellung eines Beispielmodells**: Es wird ein Beispielmodell erstellt, das der geplante Generator künftig verarbeiten können soll. Dabei kann es erforderlich sein, das Beispiel in einen Kontext aus weiteren Modellen einzubetten, um ein vollständiges Szenario zu

erhalten. Dies ist notwendig, wenn das gewünschte Generat mit dem anderer Generatoren zusammen arbeiten soll, wie es z.B. bei der Generierung von Testfällen aus Objekt- und Sequenzdiagrammen der Fall ist (siehe Abschnitt 6.5). Insbesondere das eigentliche Beispielmodell sollte aber möglichst einfach sein und nur so viele Elemente enthalten, wie für das gewünschte Ziel erforderlich sind.

2. **Manuelle Implementierung des Generats**: Das gewünschte Generat wird anhand des Beispielmodells aus Schritt 1 manuell implementiert. Dabei können bereits existierende Generatoren genutzt werden. Auch die Modifikation des generierten Codes ist in diesem Fall erlaubt, sollte aber gut dokumentiert werden, um die Änderungen später entsprechend in den Templates umzusetzen. Generell ist es zu empfehlen, manuell erstellte bzw. modifizierte Dateien getrennt vom generierten Code in einem eigenen Ordner abzulegen.
3. **Erstellung von Testfällen**: Die Funktionalität des manuell implementierten Generats wird anhand von automatisierten Tests überprüft. Diese können mit Hilfe von Modellen über bereits vorhandene Testfallgeneratoren oder ebenfalls manuell in der Sprache des Zielsystems umgesetzt werden. Entsprechend dem "Test-First"-Ansatz [BA04] kann dieser Schritt auch vor der Umsetzung des Generats in Schritt 2 erfolgen.
4. **Anlegen der Templates**: Die in Schritt 2 manuell erstellten Dateien werden jeweils als ein Template gespeichert. Diese bestehen somit nur aus festen Anteilen der Zielsprache. Die dokumentierten Änderungen an bereits generierten Dateien werden in den entsprechenden Templates in einfachen Fällen direkt umgesetzt. Für komplexere Änderungen kann der manuell geänderte Code ebenfalls vorerst als fester Anteil in die Templates kopiert werden. Wird der so erstellte Generator ausgeführt, sollte das Generat exakt der manuellen Implementierung entsprechen und die Testfälle erfolgreich durchführbar sein. Änderungen an den Modellen haben jedoch noch keine Auswirkungen auf das Generat.
5. **Sukzessive Verallgemeinerung der Templates**: In den erstellten bzw. geänderten Templates werden schrittweise die aus den Beispielmodellen stammenden Informationen durch entsprechende Einschübe ersetzt, deren Inhalt auf Basis der abstrakten Syntax bestimmt wird. Dies kann auch den Einsatz von Kontrollstrukturen notwendig machen. Darüber hinaus können Berechnungen an Kalkulatoren oder Teile eines Templates in Subtemplates ausgelagert werden. Nach jeder Iteration sollte das Generat neu erstellt und die Testfälle ausgeführt werden.
6. **Erstellung weiterer Beispielmodelle und Testfälle**: Nach Abschluss des Generators können komplexere Beispielszenarien und entsprechende Testfälle erstellt werden, um die korrekte Arbeitsweise des Generators sicher zu stellen und ggf. durch weitere Iterationen anhand dieses Vorgehensmodells zu verfeinern.

Das obige Vorgehen hat gegenüber der direkten Implementierung von Templates verschiedene Vorteile. Die Umsetzung eines Beispiels ist weniger abstrakt als die eines auf beliebige Modelle anwendbaren Templates. Dabei kann sich der Entwickler auf die Verwendung der Zielsprache

konzentrieren und muss nicht zwischen dieser und der Templatesprache wechseln. Handelt es sich bei der Zielsprache um eine höhere Programmiersprache wie Java, kann für die Implementierung im Allgemeinen auf komfortable Entwicklungsumgebungen zurückgegriffen werden, die die Arbeit deutlich vereinfachen. Die damit insgesamt verbundene Effizienzsteigerung und Fehlervermeidung relativiert den geringen Mehraufwand durch die Beispielimplementierung, zumal Teile davon in den Templates direkt wieder zu verwenden sind.

Die zusätzlichen Tests dienen nicht nur der Überprüfung der Beispielimplementierung, sondern können ebenso für die Qualitätssicherung des Generators verwendet werden. Laufen die Testfälle auch später gegen das aus den Beispielszenarien erzeugte Generat fehlerfrei, sind darüber mehrere Rückschlüsse auf die Korrektheit des Generators möglich. Zum einen können die Testfälle nur ausgeführt werden, wenn der generierte Code kompilierbar und damit lauffähig ist. Zum anderen wird zumindestens die durch die Testfälle abgedeckte Funktionalität korrekt generiert. Durch die Ausweitung dieser Testfälle im letzten Schritt des Vorgehensmodells wird somit indirekt auch die Testabdeckung des Generators erhöht.

Wie bereits erwähnt, können auch aus Modellen generierte Testfälle zum Testen des Generats eingesetzt werden. In der UML/P steht hierfür eine eigene Sprache zur Verfügung, die die Spezifikation von Testfällen mit Hilfe von Objektdiagrammen in Kombination mit Java oder Sequenzdiagrammen erlaubt (siehe Abschnitt 3.5 und 6.5). Andererseits müssen auch diese Testgeneratoren entwickelt und erweitert werden. Um diese ebenfalls durch Ausführung des Generats zu testen, ist ein entsprechender Systemcode erforderlich. Dabei reicht es aus, sogenannte Mockklassen zu erstellen, die die von den Testfällen erwarteten Ergebnisse zurück geben. Genauso können aber auch wiederum Modelle eingesetzt werden, aus denen der Systemcode generiert wird. Dabei entsteht in der UML/P die interessante Situation, dass sich Testfallgeneratoren und Generatoren für das Produktivsystem gegenseitig testen.

Ein sonst übliches Verfahren, um Generatoren zu testen, ist der Vergleich der Ausgabe mit dem erwarteten Ergebnis. Im einfachsten Fall wird dies anhand eines String-Vergleichs durchgeführt. Dieses Vorgehen hat jedoch den Nachteil, dass Abweichungen in der Formatierung, die im Allgemeinen keinen Einfluss auf die Funktionalität haben, zu einem Fehlschlagen der Testfälle führt. Ist ein Parser für die Zielsprache vorhanden, ist es deshalb ratsamer, das erwartete und tatsächliche Ergebnis jeweils einzulesen und den Vergleich auf Basis der abstrakten Syntax durchzuführen. Die von MontiCore generierte AST-Struktur bietet hierfür entsprechende Infrastruktur [Kra10]. Beide Verfahren haben jedoch den Nachteil, dass Fehler, die bereits in der Referenzimplementierung bestehen, nicht erkannt werden. Ändert sich darüber hinaus durch eine Erweiterung des Generators die Zusammensetzung des Generats, müssen ggf. alle bisherigen Referenzimplementierungen entsprechend angepasst werden. Dies ist z.B. der Fall, wenn die generierten Klassen durch die Erweiterung eine neue Methode erhalten. Bei der hier vorgeschlagenen indirekten Testmethode reicht es hingegen aus, einen weiteren Testfall zu implementieren, der die Funktionalität der neuen Methode überprüft. Dadurch müssen bestehende Testfälle deutlich seltener angepasst werden. Der AST-Vergleich bietet sich hingegen an, um einmalig die korrekte

Übertragung der manuellen Implementierung auf die Templates in Schritt 4 des Vorgehensmodells zu überprüfen oder um Generatoren zu testen, die keinen ausführbaren Code erzeugen.

Die Qualität des indirekten Testens von Generatoren, indem nicht die Generatoren selbst, sondern deren Generat getestet wird, hängt somit sowohl vom Umfang der Beispielszenarien als auch von den Testfällen ab. Die Testabdeckung des Generators ergibt sich damit aus dem Durchschnitt der ausgeführten Zeilen des Generatorcodes bei der Generierung und der Testabdeckung des Generats.

Bei der manuellen Implementierung des Generats oder dessen Umsetzung als Templates kann sich herausstellen, dass größere Teile des Codes nicht von den Modellen abhängen. In solchen Fällen ist es nicht unbedingt notwendig, diese Teile bei jedem Generatoraufruf neu zu erstellen, wenn sich stattdessen der Modell-unabhängige Code sinnvoll in eigenen Klassen kapseln lässt. Da die Klassen im generierten Code verwendet werden, bilden diese damit dessen Laufzeitsystem, wozu u.a. auch Frameworks gehören, gegen die generiert wird. Der Generator ist von diesem Laufzeitsystem im Allgemeinen nur insofern abhängig, als dass es für die Ausführung der Testfälle nach dem hier beschriebenen Verfahren notwendig ist. Der Vorteil ist, dass sich das Laufzeitsystem unabhängig von Modellen implementieren und testen lässt. Dazu kommen die bereits erwähnten Vorteile, die Entwicklungsumgebungen für höhere Programmiersprachen bieten. Oft existiert auch eine Unterstützung für die Umstrukturierung von Code, die insbesondere bei der Weiterentwicklung umfangreicherer Laufzeitsysteme hilfreich ist. Bei solchen automatisierten Refactorings ist jedoch zu beachten, dass die Templates im Allgemeinen nicht berücksichtigt werden. Eine Ausnahme bildet die in [KR06] vorgestellte Template-Engine. Bei diesem Ansatz werden Templates als kompilierbare Java-Klassen erstellt, die Anweisungen der Templatesprache als Java-Kommentare enthalten. Auf diese Weise werden die Templates von der Infrastruktur der Entwicklungsumgebungen mit unterstützt. Andererseits schränkt dies die Codegenerierung auf Java ein, so dass der flexible Einsatz der Template-Engine für verschiedene Zielsprachen nicht möglich ist. Darüber hinaus stellt jedes Template immer eine vollständige Java-Klasse dar. Die dadurch fehlende Möglichkeit einer Strukturierung in Subtemplates, wie sie in Abschnitt 5.2 dargestellt wurde, führt zu einer schlechten Kapselung und damit zu Codeduplizierungen in den Templates.

Aus den bisherigen Ausführungen und Erläuterungen des im Rahmen dieser Arbeit entwickelten Ansatzes für die Entwicklung eines Codegenerators auf Templatebasis ergeben sich insgesamt folgende Möglichkeiten und Empfehlungen für eine Strukturierung:

1. **Haupttemplates**: Dabei handelt es sich um die Templates, die den Ausgangspunkt für die Codegenerierung bilden. Diese sollten möglichst die Angaben für die Dateierzeugung sowie die Zusammensetzung erweiterbarer Subtemplates kapseln. Darüber hinaus können ggf. einfache Berechnungen durchgeführt werden, die für die eingebundenen Subtemplates notwendig sind.
2. **Subtemplates**: Die Subtemplates kapseln die Elemente der Zielsprache und sollten möglichst frei von Kontrollstrukturen und Berechnungen in der Templatesprache gehalten

werden. Für die Vermeidung von Redundanz und zur Vereinfachung der Templates ist die Einbettung von weiteren Subtemplates möglich. Die Erzeugung von Dateien sollte hingegen vermieden werden.

3. **Kalkulatoren**: Kalkulatoren können sowohl in Haupt- als auch in Subtemplates verwendet werden. Sie dienen der Kapselung komplexer Berechnungen, die in der Templatesprache nur unter erhöhtem Aufwand zu implementieren wären. Zusätzlich können sie neue Hilfsobjekte instanziieren und den Templates zur Verfügung stellen. Auf diese Weise ist es möglich, den Inhalt von Templates auf Elemente der Zielsprache zu fokussieren.
4. **Laufzeitsystem**: Modell-unabhängiger Code, der sich sinnvoll in eigenen Klassen kapseln lässt.
5. **Refactoring des Generats**: Diese abschließende Transformation des durch die Templates erstellten Generats sollte nur in seltenen Ausnahmefällen eingesetzt werden, in denen eine Umsetzung allein auf Templatebasis zu aufwändig wäre (siehe Abschnitt 5.2).

Eines der wichtigsten Ziele des vorliegenden Ansatzes ist es, möglichst lesbare Templates zu erhalten, die das Generat erkennen lassen. Neben den obigen Empfehlungen kann dies durch die Verwendung von Kommentaren unterstützt werden. In Templates ist es dabei möglich, Kommentare sowohl in der Template- als auch in der Zielsprache zu verfassen. Der Unterschied besteht darin, dass letztere auch im Generat auftauchen. Dies kann durchaus sinnvoll sein, da insbesondere die Erweiterung eines bestehenden Generators oder die Fehlersuche häufig vom Generat ausgeht. Kommentare, die gleichzeitig das Verständnis des generierten Codes unterstützen, sollten demnach in der Zielsprache angegeben werden. Darüber hinaus erzeugt das Generierungsframework automatisch für jedes Haupt- und Subtemplate einen Kommentar, über den sich im Generat für jeden Abschnitt das entsprechend verantwortliche Template sowie das Quellmodell ermitteln lässt. Auf diese Weise kann im Generat die zugehörige Templatehierarchie abgelesen und damit die möglichen Ansatzpunkte für Erweiterungen und Modifikationen identifiziert werden.

5.4 Modifikation und Erweiterung eines Generators

In dem in dieser Arbeit entwickelten Codegenerierungsansatz sind die Templates das zentrale Artefakt eines Generators (siehe Abschnitt 5.2). Dementsprechend bilden diese auch den Ansatzpunkt für Modifikationen und Erweiterungen der Generierung. Inwieweit dabei neue Templates zu erstellen oder bestehende zu modifizieren sind, hängt von Art und Umfang der Änderungen sowie der bereits vorhandenen Codegenerierung ab. Kleinere Änderungen und insbesondere Korrekturen können häufig direkt in den bestehenden Templates umgesetzt werden. Für neue Generatoren oder umfangreichere Erweiterungen, die auch Verhalten bzw. Funktionalität des generierten Systems betreffen, sind im Allgemeinen zusätzlich neue Templates zu erstellen. Dabei kann es sinnvoll sein, notwendige Änderungen ebenfalls in neuen Templates abzulegen, falls die bisherigen Templates und damit der Generator erhalten bleiben soll.

Neue Templates können als Haupttemplates dem UML/P-Framework in Form von Parametern übergeben werden und bilden damit einen eigenständigen Generator. Dieses Vorgehen empfiehlt sich, wenn der bestehenden Generierung weitere Dateien hinzugefügt werden sollen, um das Ergebnis der neuen Templates zu speichern. Erweiterungen bestehender Dateien können hingegen als Subtemplates in die entsprechenden Generatoren integriert werden. Möglich ist dies über die in Abschnitt 5.2 vorgestellten Mechanismen der Templateeinbettung oder der Verwendung von Erweiterungspunkten in den vorhandenen Templates.

Quellcode 5.18 zeigt ein Beispiel für eine Erweiterung der in Abschnitt 5.2 behandelten Klassengenerierung. In diesem Fall wurde nicht nur die Generierung selbst erweitert, sondern auch der Sprachumfang von Klassendiagrammen durch die Einführung eines neuen Stereotyps `<<singleton>>`. Für eine damit ausgezeichnete Klasse soll im Zielsystem eine dem Singleton-Entwurfsmuster entsprechende Infrastruktur erzeugt werden, die dafür sorgt, dass im System nur eine Instanz dieser Klasse existiert [GHJV95].

FM

```
1  <#if ast.hasStereotype("singleton")>
2    // Singleton
3    protected static ${ast.printName()} _instance = null;
4
5    // public access through static method (Singleton)
6    public static ${ast.printName()} getInstance() {
7      if (_instance == null) {
8        _instance = new ${ast.printName()}();
9      }
10     return _instance;
11   }
12 </#if>
```

Quellcode 5.18: Singleton-Template

Eingebunden wird das Template in den Erweiterungspunkt `t_class` des Klassentemplates in Quellcode 5.15, indem das Haupttemplate der Klassengenerierung entsprechend erweitert wird (siehe Zeile 1 in Quellcode 5.19). Das Klassentemplate selbst muss dazu nicht verändert werden. Dadurch lässt sich die alte Form der Codegenerierung weiterhin verwenden, indem nur das Haupttemplate modifiziert bzw. ausgetauscht wird. Bei den Erweiterungspunkten handelt es sich somit um eine Parametrisierung der Generatoren, die wie deren Implementierung allein auf Basis der Templates stattfindet. Das dabei verwendete Verfahren der offenen Variablen, die in übergeordneten Templates einer Templatehierarchie zu besetzen sind, lässt sich beliebig auf weitere Templateinhalte wie Aufrufe von Kalkulatoren oder Kontrollstrukturen ausweiten und ist damit äußerst flexibel. So lassen sich ebenfalls etwa Kalkulatoren auswählen oder bedingte Inhalte bestimmen. Auf diese Weise entsteht eine konfigurable Codegenerierung, deren Parameter in den Haupttemplates gekapselt sind. Durch die Auswahl der Haupttemplates beim Generatoraufruf wird schließlich die Codegenerierung gesteuert. Mit dem hier vorgestellten Verfahren lassen sich somit Generatoren auf Templatebasis flexibel steuern, erweitern bzw. neu kombinieren und ein Portfolio an Generatoren für unterschiedliche Zwecke entwickeln. Wie das Beispiel zeigt, kann

das Generat dabei nicht nur über Auswahl und Konfiguration der Codegeneratoren beeinflusst werden, sondern auch über die Modelle mit Hilfe von Spracherweiterungen wie Stereotypen.

FM

```
1  ${op.setValue("t_class", "mc.umlp.cd.pattern.Singleton")}
2  ${op.setValue("t_attr", ["mc.umlp.cd.Attribute", "mc.umlp.cd.GetterSetter"])}
3  ${op.setValue("t_meth", "mc.umlp.cd.Method")}
4  ${op.setValue("t_constr", "mc.umlp.cd.Constructor")}
5  ${op.setValue("t_assoc", "mc.umlp.cd.Association")}
6
7  ${op.callTemplate(
8      "mc.umlp.cd.Class",
9      ast.printPackage() + "." + ast.printName(),
10     ast)}
```

Quellcode 5.19: ClassMain-Template aus Quellcode 5.16 erweitert um die Singleton-Generierung aus Quellcode 5.18

Das bisher vorgestellte Verfahren ist wohlgemerkt nur eine vom Generierungsframework zur Verfügung gestellte Möglichkeit, wie Templates aufgebaut und genutzt werden können. Die Einbettung von Templates ist auch auf direktem Wege über den `includeTemplates`-Aufruf möglich. Allerdings kann die Definition von Erweiterungspunkten und deren Konfiguration in den Haupttemplates dazu beitragen, Codeduplizierungen zu vermeiden, wenn Subtemplates in unterschiedlichen Kombinationen verwendbar und austauschbar sein sollen.

Der Vollständigkeit halber sei noch erwähnt, dass das obige Beispiel für die Umsetzung des Singleton-Musters nicht ganz ausreicht. Zusätzlich muss noch die Erzeugung von Instanzen der entsprechenden Klasse verhindert werden. Dies geschieht im Allgemeinen durch eine eingeschränkte Sichtbarkeit der Konstruktoren, was sich leicht durch eine entsprechende Erweiterung des Konstruktor-Templates umsetzen lässt, das in Quellcode 5.19 über den Erweiterungspunkt `t_constr` eingebunden wird. Eventuell in den Modellen verwendete und damit unerlaubte Konstruktoraufrufe können entweder über die Definition zusätzlicher Kontextbedingungen verboten oder mit Hilfe eines abschließenden Refactorings in `getInstance()`-Aufrufe umgewandelt werden. Letzteres lässt sich alternativ auch über die Anpassung des Templates für Konstruktoraufrufe umsetzen - sofern vorhanden (siehe dazu auch Abschnitt 5.2). Erläuterungen zur Erweiterung von Kontextbedingungen sowie der Verwendung des Refactoringframeworks finden sich in den Abschnitten 7.3 und 7.5.

5.5 Komposition von Generatoren

Nicht immer werden Sprachen als eine Einheit entwickelt, sondern entstehen als Erweiterung oder Kombination einzelner Sprachen. Auch die UML/P setzt sich aus mehreren Teilsprachen zusammen, die sowohl parallel als auch in Form von Einbettung genutzt werden (siehe Kapitel 3). Werden für diese Teilsprachen jeweils einzelne Generatoren entwickelt, stellt sich die Frage nach deren Zusammenspiel und Integration. Aber auch für eine einzige Sprache ist es oft sinnvoll,

gewisse Aspekte des Zielsystems als separate Generatoren zu entwickeln, um sie so nur bei Bedarf hinzunehmen zu können. Beispiele hierfür sind zusätzliche, nur während der Entwicklung genutzte Testinfrastruktur oder Anteile, die je nach der Zielplattform des generierten Systems unterschieden werden müssen. Im Hinblick auf die Codegenerierung ist dabei das eigentliche Ziel, ein funktionierendes Gesamtsystem aus den einzelnen Generaten zu erhalten. Aus diesem Grund lässt sich die Komposition von Generatoren auf zwei Ebenen betrachten:

1. **Integration der Generatoren**: Die Generatoren werden nicht unabhängig ausgeführt sondern stehen in einer Beziehung zueinander und können sich so gegenseitig beeinflussen.
2. **Integration der Generate**: In diesem Fall müssen die einzelnen Generatoren jeweils so umgesetzt werden, dass das Ergebnis zusammen mit dem der anderen Generatoren lauffähig ist. Die Komposition findet demnach auf Ebene der Generate zur Compile- oder Laufzeit des Zielsystems statt. Dabei macht es keinen Unterschied, ob die Generatoren gleichzeitig oder nacheinander ausgeführt werden.

Im vorliegenden Fall werden Generatoren mit Hilfe von Templates bezogen auf ein bestimmtes Element der abstrakten Syntax implementiert. Bei der Verarbeitung wird jedem Template die Instanz eines entsprechenden AST-Knotens als Datenmodell zugewiesen. Auch alle anderen Aspekte wie Vorberechnungen oder Bedingungen sind in den Templates selbst gekapselt oder werden von dort aufgerufen. Aus diesem Grund bildet jedes Template im Prinzip einen eigenen Generator . Durch Einbettung und Referenzierung der Templates können daraus wiederum nach dem Baukastenprinzip umfangreichere Generatoren zusammengesetzt werden.

Wie in Abbildung 5.20 dargestellt, können für die Umsetzung kompositionaler Generatoren auch beide der oben genannten Integrationsformen gemeinsam eingesetzt werden. Die erste Integration findet auf Ebene der Templates `T1` bis `T5` statt, die sich wiederum aus weiteren Templates zusammensetzen können. Dies zeigt, dass Generatoren sowohl durch die Einbettung der Templates, als auch durch deren gemeinsame Ausführung auf den Modellen komponierbar sind. Dabei ist es irrelevant, ob es sich bei den Modellen um Instanzen derselben oder unterschiedlicher Sprachen handelt. Die zweite Ebene ist durch die aus den Templates erzeugten Generate `G1` bis `G4` dargestellt. Von der Templateeinbettung abgesehen, handelt es sich dabei um einzelne Dateien, die sich durch entsprechende Referenzierung sowie die Nutzung von Schnittstellen integrieren lassen. Eine nachträgliche Integration auf Dateiebene ist von dem in dieser Arbeit entwickelten Codegenerierungsframework nicht vorgesehen, da die Modelle sich aus Effizienzgründen jeweils unabhängig voneinander verarbeiten lassen sollen. Die beiden in Abbildung 5.20 dargestellten ASTs müssen demnach nicht gleichzeitig vorliegen.

Die Notwendigkeit einer Integration auf Ebene der Generate hängt von den Anforderungen der Ausgangssprache sowie den Möglichkeiten der Zielsprache ab. Für die UML/P und die Zielsprache Java werden entsprechende Konzepte in Kapitel 6 behandelt.

Die Spezifikation von Templates für bestimmte Typen von AST-Knoten eignet sich insbesondere im Hinblick auf die Kompositionalität von Sprachen, wie sie in MontiCore zur Verfügung steht [KRV10, GKR+08]. Dessen Spracheinbettung, die in der UML/P für die Nutzung der

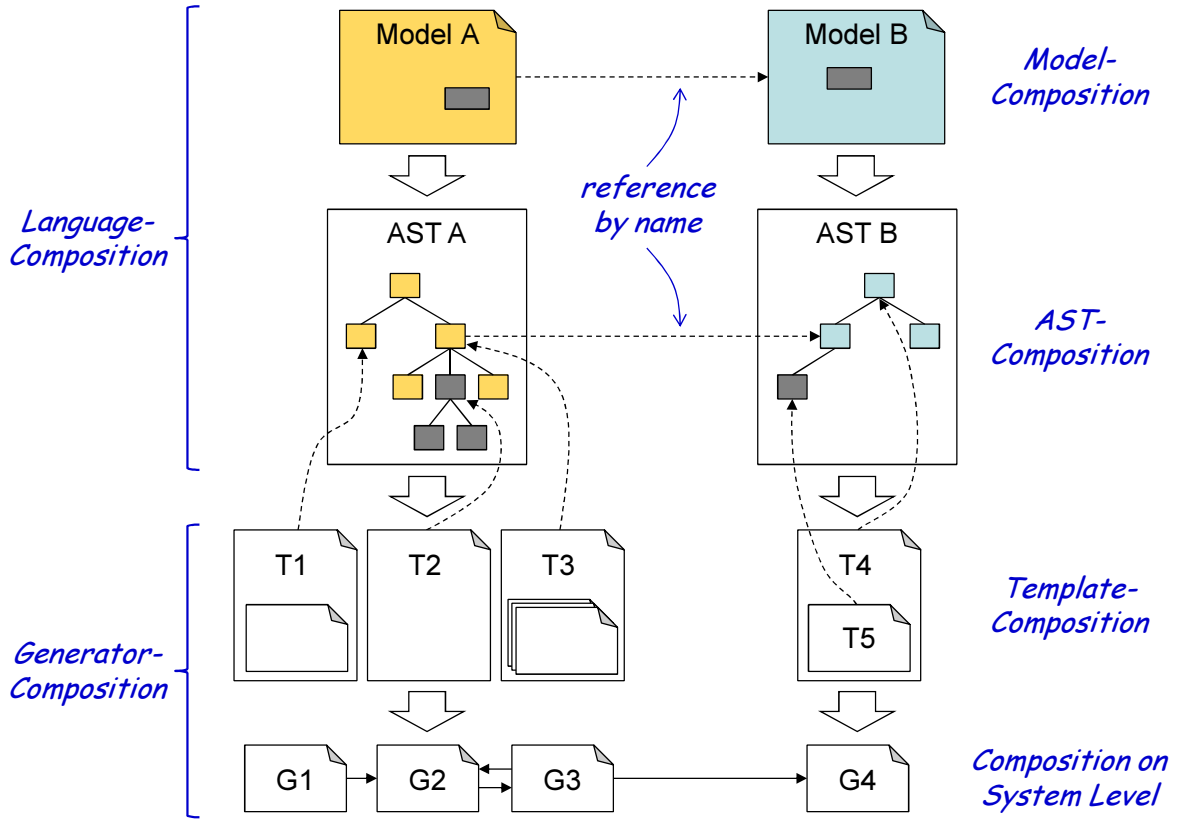


Abbildung 5.20: Komposition von Generatoren analog zur Sprachkomposition (T1-T5 symbolisieren Templates, G1-G4 Generate)

OCL/P und Java/P in den Modellen verwendet wird, basiert u.a. auf einer Zusammenführung der ASTs der einzelnen Sprachen. Dabei wird jede Sprache durch separate Parser verarbeitet, die jeweils einen AST für die Fragmente einer Sprache erstellen. Diese werden von MontiCore zu einem vollständigen AST für das vorliegende Modell integriert. In Abbildung 5.20 ist dieser Aspekt anhand der dunklen Elemente in den Modellen bzw. den ASTs dargestellt. Referenzen zu anderen Modellen werden in MontiCore hingegen nicht durch eine AST-Integration aufgelöst, sondern bleiben als einfache Namensreferenzen erhalten. Auf diese Weise lassen sich die Modelle getrennt voneinander verarbeiten und referenzierte Modelle nur bei Bedarf einlesen.

Dieses Prinzip lässt sich auf die Codegenerierung mit dem hier vorgestellten Templateansatz übertragen. Da sich die Templates jeweils auf ein bestimmtes Element der abstrakten Syntax beziehen, können diese für die Sprachen entsprechend getrennt spezifiziert und mit Hilfe der Templateeinbettung oder separaten Aufrufen in die Codegenerierung integriert werden. Diese Möglichkeit wurde in den bisherigen Beispielen bereits genutzt, um Elemente in Klassendiagrammen wie Klassen, Attribute oder Methoden mittels einzelner Templates in Java-Code zu übersetzen (vgl. Quellcode 5.16). Dadurch kann die Übersetzung für ein Element leicht und gezielt ausgetauscht oder flexibel erweitert werden. Die Trennung der Templates nach Sprachelementen hat bei entsprechender Spezifikation insbesondere aber auch dann Vorteile, wenn Sprachen

neu oder mehrmals kombiniert werden. Beispielsweise werden für Bedingungen in Statecharts und Sequenzdiagrammen `OCLExpressions` eingebettet (siehe Tabelle C.1 im Anhang C). Ein entsprechendes Template, das `OCLExpressions` nach Java überführt, kann für Statecharts und Sequenzdiagramme somit gleichermaßen eingesetzt werden. Soll hingegen die Formulierung von Bedingungen in den Modellen in einer anderen Sprache als der OCL/P erfolgen, müssen nur für die neue Sprache entsprechende Templates entworfen und in die bestehenden Templates für die Modelle integriert bzw. gemeinsam mit diesen aufgerufen werden. Insbesondere ist dies unabhängig davon, ob die neue Sprache anstatt oder alternativ zur OCL/P eingesetzt werden soll. Das Konzept der Erweiterungspunkte bietet dabei die Möglichkeit, die notwendigen Änderungen an den bestehenden Templates auch bei der Einbettung auf die Haupttemplates zu beschränken, indem Subtemplates ausgetauscht oder hinzugefügt werden.

In einigen Fällen kann es erforderlich sein, dass Generatoren untereinander Nachrichten austauschen. Bei der Templateeinbettung ist dies mit Hilfe von Variablen möglich, die automatisch ebenfalls in den Subtemplates zur Verfügung stehen. So wird im Beispiel in Quellcode 5.12 das verwendete Paket für die Factory-Generierung als Variable `package` gespeichert und könnte in Quellcode 5.11 in Zeile 1 verwendet werden. Auf diese Weise würde allein im Haupttemplate das Paket der Factory bestimmt und diese Information an die Subtemplates weitergereicht. Neben dieser Definition einfacher FreeMarker Basistypen in den Templates bieten darüber hinaus die Kalkulatoren vielfältige Möglichkeiten über die Erzeugung von Variablen beliebiger Objekte, Transformationen des ASTs, bis hin zur Ablage von Informationen im Dateisystem, um einen solchen Informationsaustausch auch jenseits der Templateeinbettung zu realisieren. Schließlich können ebenfalls im AST Informationen in Form von Annotationen abgelegt und so anderen Templates zugänglich gemacht werden.

Eines der Ziele der vorliegenden Arbeit war es, eine flexible und erweiterbare Codegenerierung zu schaffen, mit deren Hilfe sich die UML/P an die mitunter stark variierenden Anforderungen von Unternehmen, Projekten und Zielplattformen anpassen lässt. Nur so kann der breite Einsatz der UML/P als Modellierungssprache gewährleistet werden. Dazu wurde ein Konzept und eine entsprechende Infrastruktur für eine templatebasierte Codegenerierung geschaffen. Im Gegensatz zu anderen Ansätzen bilden dabei die Templates Ausgangspunkt und zentrales Artefakt für die Definition eines Generators. Die Templates selbst werden auf Basis der abstrakten Syntax der Ausgangssprache spezifiziert, die im vorliegenden Fall vom MontiCore-Framework aus den jeweiligen Grammatikbeschreibungen der Sprachen der UML/P erzeugt wird. Anhand der abstrakten Syntax baut der Parser mit dem AST eine das Modell repräsentierende Objektstruktur auf, die automatisch in den Templates zur Verfügung steht. Gleichzeitig wurde der Ansatz mit den Vorteilen visitorbasierter Generatoren kombiniert. Diese folgen der allgemeinen Funktionsweise von Codegeneratoren, bestimmte Elemente der Eingabe ggf. unter zusätzlichen Bedingungen in ein entsprechendes Artefakt der Zielsprache zu überführen. Entsprechend können die Templates im vorliegenden Ansatz für einen bestimmten Typ von AST-Knoten spezifiziert werden. Mit Hilfe der Templateeinbettung entsteht dadurch eine dem AST ähnliche hierarchische Strukturierung

der Templates, bei deren Verarbeitung gleichzeitig der AST traversiert wird. Dieser Aufbau erleichtert nicht nur Austausch und Erweiterung einzelner Teile der Generierung, sondern eignet sich insbesondere auch für die Komposition von Generatoren. Mit den Kalkulatoren stehen darüber hinaus in den Templates Hilfsobjekte zur Verfügung, über die die Vorteile und Möglichkeiten der Programmiersprache Java für die Datenaufbereitung oder die Überprüfung von Bedingungen genutzt werden können. Zusammen mit einem Vorgehensmodell für die Entwicklung und Erweiterung von Generatoren sowie einem Test-Konzept steht damit eine umfangreiche Infrastruktur zur Verfügung, die nicht nur von der UML/P genutzt werden kann.

Kapitel 6

Konzepte der Codegenerierung

Der Nutzen von Modellen lässt sich durch den Einsatz von Codegeneratoren erheblich steigern. Diese stellen nicht nur die Konsistenz zwischen den Modellen und der Implementierung sicher, sondern tragen auch zur Vermeidung von Fehlern und zu einer effizienten Entwicklung bei. Die Effizienzsteigerung liegt vor allem in der höheren Abstraktion der Modelle begründet, deren fehlenden technischen Details für die Umsetzung in ein lauffähiges System in den Codegeneratoren gekapselt werden können. Damit überbrücken die Generatoren den Abstand zwischen den abstrakten Modellen und der konkreten Implementierung. Hinzu kommt, dass auf diese Weise die technische Umsetzung für eine bestimmte Plattform unabhängig von den Modellen entwickelt und verbessert werden kann. Auch eine Portierung des modellierten Systems auf eine andere Plattform lässt sich so allein durch einen Wechsel der Generatoren realisieren.

In Kapitel 5 wurde der im Rahmen dieser Arbeit entwickelte Ansatz für eine templatebasierte Codegenerierung vorgestellt, der eine unabhängige Entwicklung und Komposition von Generatoren ermöglicht. Eine Form der Komposition stellt dabei die Integration der Templates durch Einbettung dar und wurde bereits in Abschnitt 5.5 behandelt. In diesem Kapitel wird hingegen die Umsetzung der Modelle auf eine objektorientierte Zielplattform wie Java betrachtet, wobei der Fokus auf der Architektur des Generats im Hinblick auf die Kompositionalität liegt. Dies stellt die zweite Möglichkeit der Generatorkomposition dar, bei der das Zielsystem durch die unabhängige Ausführung mehrerer Generatoren erstellt wird. Dazu müssen diese jedoch so umgesetzt sein, dass sich die einzelnen Generate zu einem funktionierenden Gesamtsystem vereinen. Da in diesem Kapitel weniger der konkrete Code, als vielmehr die dahinter stehenden Konzepte im Vordergrund stehen, werden diese im Folgenden ebenfalls anhand von UML-Modellen erläutert. Auch hier zeigen die Modelle ihre Stärke in der Konzentration auf das Wesentliche durch Abstraktion, wodurch gleichzeitig die Erläuterungen unabhängig von der konkreten Sprache der Zielplattform werden. Kurz gesagt wird das Ergebnis eines Generators der UML/P wiederum durch Modelle der UML/P abstrakt dargestellt. Dafür wird in diesem Fall auf die graphische Notation zurückgegriffen, deren Vorteile in einer schnelleren Erfassung des Dargestellten liegen, während sich die textuelle Fassung besser für die effiziente Erstellung insbesondere in agilen Entwicklungsprozessen eignet (siehe Abschnitt 2.3 und [GKR+07]).

In [Rum04b] wurden bereits allgemeine Konzepte für die Codegenerierung aus UML/P-Modellen anhand der Sprache Java ausführlich behandelt. Die dort vorgestellten Grundprinzipien wurden auch in dieser Arbeit verwendet. Allerdings konzentriert sich die Darstellung in [Rum04b] auf die Generierung kleiner Einheiten aus den Modellen. Deren Integration zu einem Gesamtsystem wird im Wesentlichen offen gelassen oder nur angedeutet. Dabei scheint es naheliegend, den generierten Code aus einem Methodenstatechart oder eines konstruktiv verwendeten Objektdiagramms in die entsprechende Klasse direkt einzufügen. Im Klassendiagramm ist jedoch nicht ersichtlich, welche zusätzlichen Modelle weitere Details zu einer modellierten Klasse beisteuern. Dies führt dazu, dass bei einer Änderung nur eines Modells potentiell alle Modelle geladen werden müssen, um die notwendigen Codefragmente der von der Änderung betroffenen Klassen neu zu erstellen. Ein solches Vorgehen bringt jedoch erhebliche Performance-Nachteile hinsichtlich der Codegenerierung mit sich und behindert insbesondere agile Vorgehensweisen wie XP [BA04] oder Scrum [Sch10a]. Diese sind auf eine häufige Ausführung des generierten Systems sowie der Testfälle während der Entwicklung angewiesen, um möglichst jederzeit die Funktionalität des Systems sicherstellen zu können [BBB+].

Es gibt verschiedene Möglichkeiten, dieses Problem zu umgehen oder zumindest zu reduzieren. So könnten Namenskonventionen der Modelle, zusätzliche Annotationen in Form von Stereotypen oder die Beschränkung auf Modelle desselben Pakets die Suche nach den zu einer Klasse beitragenden Details einschränken. All diese Varianten ließen sich prinzipiell mit dem vorliegenden Generierungsansatz z.B. in Form entsprechender Kalkulatoren oder Refactorings umsetzen (siehe Abschnitt 5.2). Allerdings führt dies gleichzeitig zu Einschränkungen oder erhöhtem Aufwand bei der Erstellung der Modelle, was vermieden werden sollte. Alternativ könnte eine Infrastruktur geschaffen werden, die es ermöglicht, nur Teile einer generierten Datei neu zu erstellen. Dazu müsste ein Generator den vom ihm erstellten Anteil im Generat markieren[1] und die Infrastruktur sicher stellen, dass bei einer erneuten Ausführung nur dieser Teil ersetzt wird. Auf diese Weise entständen für jeden Generator geschützte Regionen im Generat, die von anderen Generatoren nicht überschrieben werden. Somit genügt es, nur die geänderten Modelle erneut zu verarbeiten, mit dem Nachteil, dass zusätzlich die betroffenen Dateien des bereits generierten Codes eingelesen werden müssen. Der Performancevorteil gegenüber der vollständigen Generierung hängt demnach von der Größe des Projekts sowie dem Verhältnis zwischen Modellen und generierten Dateien ab, sollte aber in den meisten Fällen positiv ausfallen.

Die Infrastruktur für diese zweite Variante ließe sich ebenfalls auf Basis von Kalkulatoren oder Refactorings mit dem vorliegenden Generierungsframework umsetzen. Alternativ könnte diese auch über zusätzliche Operationen im Template-Operator in zukünftigen Versionen des Frameworks generell zur Verfügung gestellt werden. Der Vorteil der in diesem Kapitel vorgestellten Konzepte für eine Komposition auf Basis der Generate ist jedoch, dass diese unabhängig von den Möglichkeiten der Generator-Technologie anwendbar sind. Darüber hinaus können diese

[1] Dies ist im vorliegenden Generierungsframework anhand von Kommentaren bereits jetzt der Fall, um das Auffinden der entsprechenden Templates zu erleichtern.

ebenfalls genutzt werden, um den generierten Code manuell zu erweitern, ohne diesen selbst zu modifizieren.

Mit der Behandlung von Unterspezifikation und projektspezifischen Erweiterungen der Modelle beschäftigen sich die Abschnitte 6.1 und 6.2 zu Beginn noch mit zwei allgemeinen Aspekten der Codegenerierung aus UML/P-Modellen. Erst im Anschluss daran werden Architekturen und Konzepte für Generate vorgestellt, die auch dann ein funktionierendes Gesamtsystem ergeben, wenn die entsprechenden Generatoren oder die Modelle getrennt voneinander ausgeführt bzw. verarbeitet werden. In dieser Hinsicht wird in Abschnitt 6.3 auf die Generierung des Produktivcodes eingegangen. Für diesen wird im darauf folgenden Abschnitt 6.4 eine Möglichkeit zur Codeinstrumentierung vorgestellt, die als Grundlage für die Testfallgenerierung in Abschnitt 6.5 dient. Die Qualität der Testfälle kann schließlich mit Hilfe von Metriken gemessen werden. Das Besondere an dem im Abschnitt 6.6 beschriebenen Konzept ist, dass sich das Ergebnis der Metriken auf die Modelle und nicht auf den generierten Code bezieht. Dabei sind die Metriken selbst unabhängig von den Modellen bzw. deren Sprache implementiert, so dass sie für beliebige Sprachen wiederverwendet werden können.

6.1 Umgang mit Unterspezifikation

Als Unterspezifikation wird eine fehlende Festlegung oder das Auslassen von Details in den Modellen verstanden. Diese tritt in der UML/P in unterschiedlichen Formen auf:

1. **Abstraktion**: Bereits die abstrakte Sicht der Modelle auf das System kann als eine Form der Unterspezifikation betrachtet werden. Insbesondere abstrahieren die Modelle von den technischen Details und einer konkreten Zielplattform des modellierten Systems, die erst durch die Implementierung oder die Auswahl und Parametrisierung eines Generators festgelegt wird.
2. **Unvollständigkeit**: Die UML/P erlaubt an vielen Stellen in den Modellen das Auslassen von Details, indem bestimmte Angaben nur optional sind. Dazu gehören etwa fehlende Initialisierungen und Typangaben von Attributen in Klassen- und Objektdiagrammen oder Assoziationen ohne Kardinalitäten, Navigationsrichtungen und Rollennamen. Doch die Unvollständigkeit lässt sich nicht nur an den optionalen Elementen festmachen. So können in Klassen ganze Methoden und Attribute fehlen, etwa weil das Klassendiagramm nur eine spezielle Sicht auf das System darstellt oder sich das Modell noch in der Entwicklung befindet. Ähnliche Gründe können in Statecharts dazu führen, dass ein Zustand keine passenden Transitionen für alle möglichen Stimuli bereitstellt. Um solche weniger offensichtlichen Unvollständigkeiten aufzuzeigen, wurden in der UML/P mit den Repräsentationsindikatoren zwei spezielle Marker eingeführt (siehe Abschnitt 3.8). Andere Unvollständigkeiten lassen sich nicht an einem Modell festmachen, sondern ergeben sich erst aus der Gesamtheit der Modelle. Ein Beispiel hierfür ist die Spezifikation von Methodenverhalten. Wird diese nicht

als Methodenrumpf angegeben, ist im entsprechenden Klassendiagramm nicht ersichtlich, ob das Verhalten z.B. bereits als Statechart vorliegt oder noch unterspezifiziert ist.

3. **Interpretationsspielraum**: Die Interpretation von Modellen der UML/P ist nicht immer eindeutig, sondern kann je nach Zielsetzung und Verwendung unterschiedlich ausgelegt werden [Grö10]. Solche auch als semantische Variationspunkte bezeichneten Aspekte können etwa bei der Beurteilung eines Objektdiagramms als finales Ergebnis eines Testfalls auftreten. Hier stellt sich die Frage, wie exakt die dargestellte Situation dem System entsprechen soll. So können beliebige weitere Objekte oder auch Links der modellierten Objekte beim Vergleich zugelassen oder jeweils verboten sein. Ähnliche Fragestellungen entstehen bei Interaktionen in Sequenzdiagrammen (siehe Abschnitt 3.4). Auch ein Nichtdeterminismus in Statecharts kann unterschiedlich interpretiert werden, etwa als unvollständige oder abstrakte Darstellung des Statecharts oder als echter Nichtdeterminismus im System. Allgemein handelt es sich bei einem Nichtdeterminismus um eine Wahlmöglichkeit, bei der nicht spezifiziert ist, ob diese im Laufe der Entwicklung, bei der Implementierung bzw. Codegenerierung oder zur Laufzeit vom System oder dessen Benutzer getroffen wird.

Die Gründe für eine Unterspezifikation liegen demnach vor allem in der Abstraktion der Modelle von technischen Details, in noch auszuarbeitenden Verhalten und Struktur des Systems sowie in festzulegenden Entscheidungen bezüglich Verwendung und Interpretation der Modelle. Entsprechend sind insbesondere die Modelle in der frühen Entwicklungsphase eines Softwaresystems im Allgemeinen noch stark unterspezifiziert. Im Laufe der Entwicklung wird ein Teil dieser Unterspezifikationen durch das Hinzufügen von Details und weiteren Modellen aufgelöst. Dazu können auch Stereotypen verwendet werden. Bereits in [Rum04a] wurden eine Reihe von Stereotypen für die UML/P eingeführt, die der Vervollständigung der Modelle oder der Festlegung von semantischen Variationspunkten dienen. So ist die Aufhebung der Unvollständigkeit eines Statecharts in Bezug auf nicht angegebene Stimuli durch die Verwendung der Stereotypen `<<completion_ignore>>`, `<<completion_chaos>>` oder `<<error>>` möglich. Die ersten beiden werden am Statechart angegeben und stehen für keinen bzw. einen zufälligen Zustandswechsel, falls ein Stimulus in einem Zustand nicht verarbeitet werden kann. Der dritte Stereotyp kennzeichnet einen expliziten Fehlerzustand, der in solch einem Fall eingenommen wird. Ähnlich lassen sich semantische Variationspunkte bei Interaktionen in Sequenzdiagrammen auflösen (siehe Abschnitt 3.4). Dies sind nur Beispiele für die Behandlung von Unterspezifikationen mit Hilfe von Stereotypen, die sich durch die Definition eigener Stereotypen beliebig erweitern lassen.

Die noch offenen Unterspezifikationen müssen schließlich bei der Umsetzung in den Zielcode behandelt werden. Erfolgt die Implementierung manuell, ist dies Teil der intellektuellen Arbeit der Programmierer. Beim Einsatz von Codegeneratoren muss dieses Wissen hingegen in den Generatoren selbst enthalten sein. Eine Steuerung erfolgt hier über die Auswahl und Parametrisierung der Generatoren. Im vorliegenden templatebasierten Ansatz werden die Generatoren durch Übergabe der Haupttemplates an das Generierungsframework ausgewählt (siehe Kapitel 5). Die Parametrisierung wiederum findet durch die Auswahl von Subtemplates und das Besetzen

von Variablen in den Templates selbst statt. Wie Abbildung 6.1 zeigt, entsteht auf diese Weise eine schrittweise Verfeinerung des Systems zuerst auf Basis der Modelle und schließlich durch die Umsetzung der Generatoren. Dabei ist es sinnvoll, die von den verwendeten Generatoren nicht behandelbaren Unterspezifikationen anzuzeigen, um sie auf der Modellierungsebene zu beheben. Hierfür wurde im Rahmen dieser Arbeit ein anpassbares und erweiterbares Framework für Kontextbedingungen geschaffen, das solche Analysen noch vor Aufruf der Generatoren erlaubt (siehe Abschnitt 7.3). Diese lassen sich in die Editoren der einzelnen Sprachen einbinden, so dass der Modellierer auf Fehler oder fehlende Spezifikationen in den Modellen selbst aufmerksam gemacht wird.

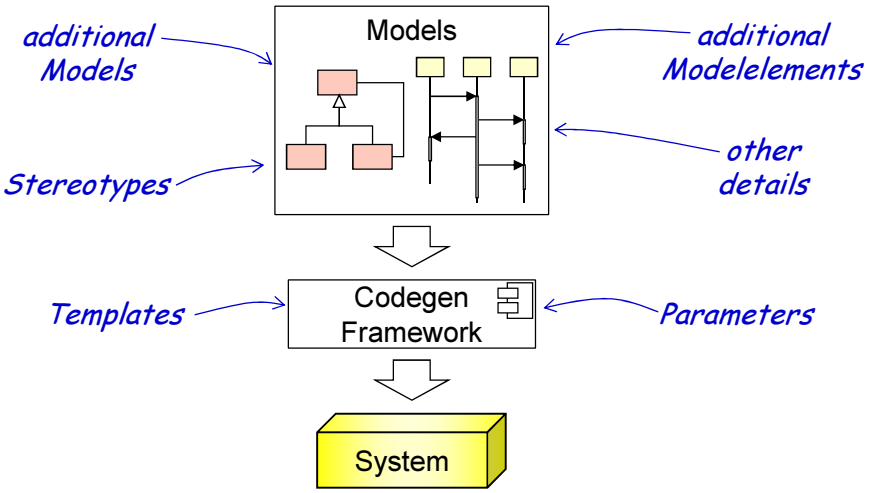


Abbildung 6.1: Behandlung von Unterspezifikation durch schrittweise Verfeinerung

Die Generatoren konkretisieren die abstrakte Sicht der Modelle durch die Kapselung der technischen Details, die für die Umsetzung des Systems auf der Zielplattform notwendig sind. Durch deren Auswahl und Parametrisierung wird demnach die Abstraktion als eine Form der Unterspezifikation überwunden. Bei der Ausführung reagiert ein Generator letztendlich auf die Elemente und Informationen, die in den Modellen vorliegen. Wie bereits an den obigen Beispielen gezeigt wurde, können insbesondere Stereotypen verwendet werden, um Elementen oder ganzen Modellen auf abstrakte Weise besondere Eigenschaften zuzuordnen. Auch hierbei handelt es sich um eine Abstraktion von einer konkreten technischen Umsetzung. Codegeneratoren werden deshalb häufig auf genau diese Stereotypen reagieren, um eine entsprechend angepasste Generierung auszuführen. Auf diese Weise erfolgt eine Art Steuerung der Codegenerierung auf der Modellebene für die auch eigene Stereotypen definiert werden können (siehe auch Abschnitt 6.2). Dabei ist jedoch darauf zu achten, dass die Stereotypen auf den Modellen selbst interpretierbar sind und nicht als reine Parameter für einen speziellen Codegenerator missbraucht werden. Nur so bleiben die Modelle in sich verständlich und es entsteht keine Abhängigkeit zu einem bestimmten Codegenerator. Zur Festlegung Generator-spezifischer Parameter sollten stattdessen die Haupttemplates verwendet werden (siehe Abschnitt 5.2).

Codegeneratoren können nicht nur die technischen Aspekte für eine bestimmte Zielplattform kapseln. Verbleiben darüber hinaus noch Unterspezifikationen in den Modellen, existieren verschiedene Lösungsansätze, wie diese ebenfalls in den Generatoren behandelt werden können:

1. **Standardumsetzung**: Für viele Unterspezifikationen existieren bereits auf den Modellen bestimmte Standardannahmen (siehe Kapitel 3), die von den Generatoren entsprechend umgesetzt werden können. So werden etwa nicht spezifizierte Navigationsrichtungen von Assoziationen mit einer bidirektionalen Navigation oder fehlende Kardinalitätsangaben mit der Kardinalität `1` gleichgesetzt. Aber auch andere fehlende Elemente können als spezielle Eigenschaft eines Generators automatisch ergänzt werden. Beispiele hierfür sind die Generierung von get- und set-Methoden oder von Konstruktoren mit und ohne Parameter in Klassen, für die solche Angaben im Modell fehlen.

   Selbst Aspekte wie Nichtdeterminismen in Statecharts lassen sich durch bestimmte Annahmen im Generator in ein deterministisches System überführen. Eine Strategie könnte etwa sein, jeweils die erste passende Transition auszuführen. In dieser Hinsicht bietet die textuelle Notation die Besonderheit, dass durch die zeilenweise Angabe eine natürliche Reihenfolge der Transitionen existiert, die sich in der graphischen Fassung nur schwer definieren lässt. Aus diesem Grund sieht [Rum04a] eine Gewichtung der Transitionen über zusätzliche Stereotypen vor, die sich auch für die textuelle Notation verwenden lässt. Andere Ansätze wie Stateflow nutzen Konventionen wie die Reihenfolge im Uhrzeigersinn [HR04a]. Die Gewichtung entlang der Reihenfolge der Transitionen löst auch den Nichtdeterminismus in Quellcode 3.15 aus Abschnitt 3.3 und entspricht der dort diskutierten Lösung über die Angabe zusätzlicher Vorbedingungen.

2. **Bereitstellung von Schnittstellen**: Unterspezifikationen wie das fehlende Verhalten von Methoden lassen sich nicht oder nur schwer mit Hilfe von Standardannahmen ergänzen. Stattdessen können in solchen Fällen Schnittstellen generiert werden, die eine spätere Ergänzung durch manuelle Implementierung oder Generate anderer Generatoren ermöglichen. Dieser Punkt wird in Abschnitt 6.3 genauer behandelt.

3. **Intelligenter Generator**: Dieser Ansatz geht deutlich über die in Punkt 1 beschriebene Umsetzung von Standards hinaus. Dabei werden die Modelle vom Generator auf das Auftreten von bestimmten Strukturen oder Namen untersucht. Anhand des Ergebnisses dieser Analyse werden schließlich Annahmen über die Funktionalität des Zielsystems getroffen. Mit Hilfe dieses Verfahrens kann zum Beispiel die Verwendung von Entwurfsmustern [GHJV95] in Klassendiagrammen erkannt und im Zielsystem entsprechend umgesetzt werden. Setzt sich der Name einer Klasse etwa aus einem existierenden Typnamen und der Endung "Factory" zusammen, kann allein diese Angabe zur vollständigen Generierung einer Factory für den Typ führen. Ähnlich ließe sich das Observer-Muster anhand einer entsprechend benannten sowie gerichteten Assoziation oder das Composite-Muster an seiner speziellen Struktur erkennen und die zugehörigen Klassen im System automatisch mit den notwendigen Methoden ergänzen. Auf diese Weise sind insbesondere auch Domänen-spezifische

Umsetzungen denkbar. So könnte etwa die Angabe einer Klasse “Person” zur Generierung einer Adressverwaltung mit Datenbankanbindung führen. Dadurch können sehr komplexe aber auch sehr mächtige Generatoren entstehen, die eine hohe Abstraktion der Modelle ermöglichen.

Wie die Beispiele und Erläuterungen zeigen, lassen sich selbst aus unterspezifizierten Modellen lauffähige Systeme generieren. Die genaue Umsetzung ist dabei eine spezifische Eigenschaft des jeweiligen Generators, die sich gegebenenfalls über Parameter steuern lässt. Mit Hilfe entsprechender Testfälle kann sicher gestellt werden, dass trotz der vom Generator getroffenen Annahmen das generierte System dem erwarteten Ergebnis entspricht.

6.2 Projektspezifische Variabilität

Die UML/P bietet wie der OMG-Standard der UML [OMG10d] die Möglichkeit, die Sprache über die Definition eigener Stereotypen zu erweitern. Auf diese Weise entstehen Varianten der UML/P, die an spezifische Anforderungen von Projekten, Unternehmen oder Domänen angepasst sind. Die Definition einer solchen Variante wird als Profilbildung bezeichnet und die Variante selbst als Profil. Doch auch die unterschiedliche Auslegung der bereits in Abschnitt 6.1 angesprochenen semantischen Variationspunkte können Teil eines Profils sein. Entsprechend stellen die in Kapitel 4 aufgeführten Kontextbedingungen nur einen sinnvollen Basissatz dar, der sich für Profile, aber auch für spezielle Anforderungen von Codegeneratoren beliebig anpassen und erweitern lässt. Um dies auch technisch zu unterstützen, wurde im Rahmen dieser Arbeit ein Framework entwickelt, das eine flexible Auswahl und Erweiterung der Kontextbedingungen ermöglicht (siehe Abschnitt 7.3). Dies schließt ebenfalls eine abweichende Priorisierung in Bezug auf die Fehlergrade mit ein. Letzteres eignet sich unter anderem auch für die Anpassung der Kontextbedingungen an die einzelnen Entwicklungsphasen, in denen sich die Anforderungen an die Modelle insbesondere hinsichtlich des Abstraktionsgrades unterscheiden. Ein Profil kann demnach auch mehrere Sätze an Kontextbedingungen enthalten, die je nach Entwicklungsphase eingesetzt werden.

Die unterschiedliche Interpretation der Modelle und die zusätzlichen Stereotypen müssen ebenfalls von den eingesetzten Codegeneratoren berücksichtigt werden, so dass diese in gewisser Weise ebenfalls als Teil eines Profils betrachtet werden können. Die Umsetzung der Generatoren unterscheidet sich hingegen nicht von den in diesem und in Kapitel 5 diskutierten Ansätzen. Je nach Grad der Abweichung des Profils von der allgemeinen UML/P können dabei ebenfalls bestehende Generatoren wiederverwendet werden. So kann es bei der Einführung zusätzlicher Stereotypen ausreichen, nur für die damit markierten Elemente neue Generatoren zu entwerfen und diese mit existierenden Generatoren zu kombinieren. Dieses Verfahren wurde bereits anhand von Quellcode 5.18 in Abschnitt 5.4 diskutiert.

Abbildung 6.2 fasst die einzelnen Aspekte der Profilbildung in der UML/P zusammen. Dabei wird angedeutet, dass auf die zusätzlichen Stereotypen eines Profils ebenfalls in Kontextbedin-

gungen und Templates Bezug genommen wird. Die unterschiedlichen Auslegungen semantischer Variationspunkte finden sich hingegen nur indirekt in den Templates wieder.

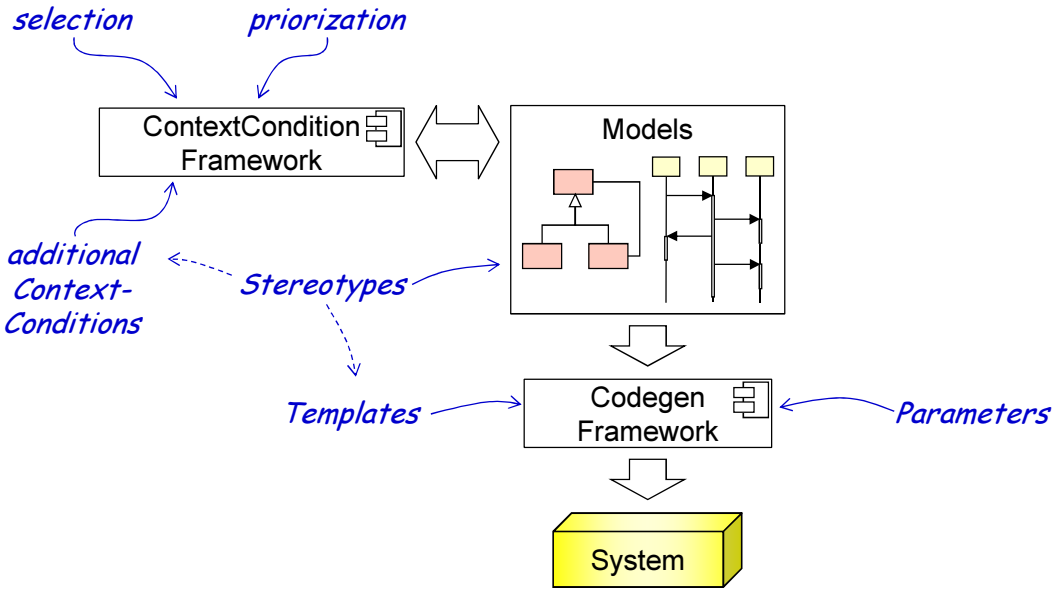


Abbildung 6.2: Profilbildung in der UML/P

In einigen wenigen Fällen kann die textuelle Notation der UML/P semantische Variationspunkte bieten, die in der graphischen Fassung nicht bestehen. Dies betrifft Elemente, die graphisch mit Hilfe von Linien miteinander in Beziehung gesetzt werden. Dieser direkte Bezug fehlt in den textuellen Modellen und wird stattdessen mit Hilfe von Namen hergestellt. Identifizieren letztere ein Element nicht eindeutig, sind unterschiedliche Interpretationen möglich. Ein Beispiel hierfür sind die Links in Objektdiagrammen. Entsprechende Überlegungen lassen sich aber auch für Transitionen in Statecharts[2] und Interaktionen in Sequenzdiagrammen anstellen. So können in Objektdiagrammen mehrere anonyme Objekte gleichen Typs angegeben werden. Wie Abbildung 6.4 für das Objektdiagramm in Quellcode 6.3 zeigt, kann ein Link zwischen solchen Objekten dadurch auf die folgenden zwei Arten ausgelegt werden:

1. **Strikte Interpretation**: Der Link ist nicht eindeutig und stellt somit einen Fehler dar. Um diesen zu korrigieren, müssen für die Objekte Namen vergeben und anstelle der Typbezeichnungen im Link verwendet werden.
2. **Variable Interpretation**: Der Link ist formal richtig aber unterspezifiziert. Um die Unterspezifikation zu beheben, könnten folgende Stereotypen eingeführt und am Link oder Objektdiagramm angegeben werden:

   a) `<<apply_single>>`: Das Objektdiagramm entspricht einer Situation im Zielsystem genau dann, wenn nur ein Paar der möglichen Objekte verlinkt ist.

[2]Für Zustände dürfen hier gleiche Namen vergeben werden, wenn sie sich in verschiedenen Subzuständen befinden.

b) `<<apply_multiple>>`: Es wird gefordert, dass Links zwischen allen möglichen Objekten im Zielsystem existieren.

Wird ein Objektdiagramm für die Erzeugung eines Systemzustands eingesetzt, bezieht sich die variable Interpretation jeweils nur auf die neu angelegten Objekte, sodass im Fall von `<<apply_single>>` ein zufälliges Objektpaar vom Codegenerator ausgewählt wird.

```
1  objectdiagram ControllerConnections {
2
3    :Controller;
4    :Animal {species = "Bird";}
5    :Animal {species = "Turtle";}
6
7    link Animal -- Controller;
8
9  }
```

Quellcode 6.3: Beispiel für einen semantischen Variationspunkt in textuellen Objektdiagrammen

In der graphischen Darstellung sind die Links per Definition eindeutig, so dass insbesondere die vorgeschlagenen Stereotypen der variablen Interpretation hier nicht sinnvoll anwendbar sind. Am nächsten an der graphischen Notation ist daher die strikte Interpretation. Von dieser wird deshalb auch im Basissatz der Kontextbedingungen in Kapitel 4 ausgegangen, so dass diese für eine variable Interpretation innerhalb eines Profils angepasst werden müssten. Darüber hinaus ist zu überlegen, wie in diesem Fall eine graphische Darstellung der Modelle zu realisieren ist, wenn diese ebenfalls genutzt werden soll. Wird die `<<apply_single>>`-Variante etwa als Nachbedingung eines Testfalls verwendet, müssen im Grunde für jede mögliche Verlinkung ein einzelnes Objektdiagramm entworfen und die einzelnen Varianten mit "oder" zu einer Gesamtaussage verknüpft werden, um eine graphische Entsprechung des textuellen Modells zu erhalten (siehe Abbildung 6.4).

Abgesehen von dieser unterschiedlichen Interpretation von Beziehungen zwischen Elementen wurden andere semantische Variationspunkte bereits in [Rum04a] ausführlich behandelt und entsprechende Stereotypen für deren Konkretisierung eingeführt. Diese lassen sich äquivalent ebenfalls in der textuelle Notation verwenden (siehe dazu auch Abschnitt 6.1). Darüber hinaus beschäftigt sich [Grö10] mit der Semantikdefinition von objektbasierten Modellierungssprachen und betrachtet dabei eingehend die UML/P und deren semantische Variabilität.

Wie das obige Beispiel zeigt, müssen die Bedeutung und Konsequenzen von zusätzlichen Variationspunkten für Profile der UML/P gut überlegt sein. Die meisten Profile werden sich hingegen auf die Einführung spezifischer Stereotypen beschränken, die den damit ausgezeichneten Elementen eine spezielle Bedeutung innerhalb einer Domäne oder eines anderen Kontextes zuordnen. Für diese Form der semantischen Erweiterung der UML/P genügt im Allgemeinen die Definition zusätzlicher Kontextbedingungen und Generatoren, die sich auf die Behandlung der ausgezeichneten Elemente konzentrieren.

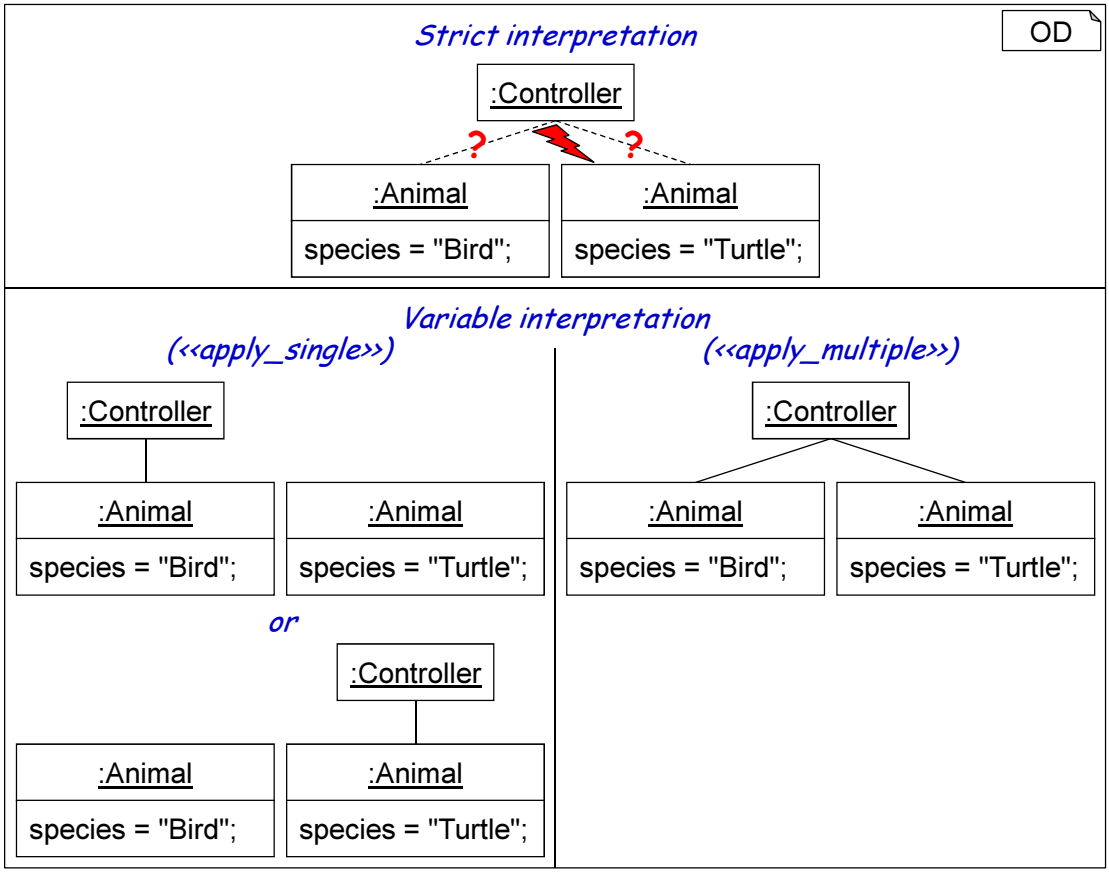


Abbildung 6.4: Mögliche Interpretationen des semantischen Variationspunkts aus Quellcode 6.3

6.3 Generierung von Produktivcode

Die in den Klassendiagrammen beschriebene Architektur spielt eine zentrale Rolle bei der Modellierung eines Softwaresystems. Andere Modelle bauen darauf auf und ergänzen die Architektur um Verhalten, Bedingungen und Testfälle, indem sie sich auf die in den Klassendiagrammen definierten Typen sowie deren zugeordneten Attribute, Konstruktoren, Methoden und Assoziationen beziehen. Erst durch diesen Bezug ergeben die einzelnen Modelle ein Gesamtsystem. In den Klassendiagrammen selbst ist hingegen nicht ersichtlich, welche anderen Modelle weitere Details den definierten Typen hinzufügen. Um eine Klasse in ihrer Gesamtheit zu betrachten, müssen demnach alle Modelle durchsucht werden, da sich in jedem zusätzliche Anteile der Klasse befinden können. Dies ist weniger ein Problem bei Testfällen, die einen rein überprüfenden aber keinen spezifizierenden Charakter haben und dementsprechend das Systemverhalten nicht beeinflussen. Statecharts, Objektdiagramme, Java/P und OCL/P können hingegen konstruktiv Aspekte dem System hinzufügen. Im Vergleich zu objektorientierten Programmiersprachen ergibt sich hier demnach eine andere Kapselung, indem Struktur und Verhalten einer Klasse nicht

notwendigerweise in einer Datei definiert werden, sondern sich auf mehrere Modelle aufteilen können. Diese Anteile im System wieder zu vereinen ist Aufgabe des Codegenerators.

Abbildung 6.5 veranschaulicht diesen Sachverhalt, indem es die referenziellen Beziehungen zwischen den einzelnen Modellarten aufzeigt. Der Übersichtlichkeit halber ausgenommen ist dabei die Referenzierung von Java/P-Typen, die in allen Modellarten möglich ist. Im Unterschied zu Klassendiagrammen können in Java/P definierte Typen jedoch nicht um Verhalten ergänzt werden, so dass es sich hier um eine reine Nutzung der Typen handelt.

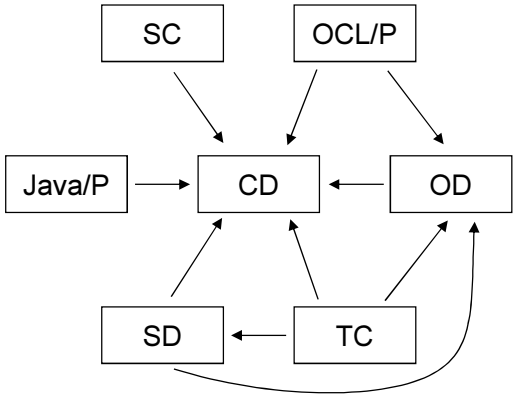


Abbildung 6.5: Referenzielle Beziehungen in der UML/P (ausgenommen der Nutzung von in Java/P definierten Typen)

Ziel der im Folgenden vorgestellten Codegenerierung ist es, die Kapselung der Modelle auf eine objektorientierte Zielplattform zu übertragen. Insbesondere soll es dabei möglich sein, trotz der in Abbildung 6.5 dargestellten Beziehungen aus jedem Modell getrennt Code zu generieren, ohne bereits generierten Code anderer Modelle zu modifizieren oder alle Modelle einlesen zu müssen. Dies trägt zu einem effizienten Entwicklungsprozess bei, da nur das Generat geänderter Modelle neu erstellt werden muss. Vor allem für die agile Entwicklung mit der UML/P ist dies essenziell, da so auch geringe Modifikationen an den Modellen ohne längere Wartezeiten direkt anhand des generierten Systems überprüft werden können. Werden dazu ebenfalls aus Modellen generierte Testfälle verwendet, kann der Modellierer trotz der Abstraktion mit der UML/P wie mit einer ausführbaren Sprache arbeiten. So ist ein iteratives Vorgehen bei der Modellierung möglich, indem kurze Zyklen der Modellierung von Test- und Produktivcode sowie der Testausführung aufeinander folgen. Entsprechend erlaubt dies auch eine testgetriebene Entwicklung nach dem Test-first Ansatz auf Modellbasis [Bec02].

Insgesamt wurden die folgenden drei Varianten untersucht, um Generate auf Systemebene zu komponieren:

1. Delegator-basierte Komposition mit Factories (Abschnitt 6.3.1)
2. Delegator-basierte Komposition per Namenskonvention (Abschnitt 6.3.2)
3. Aspektorientierte Komposition (Abschnitt 6.3.3)

Die Erweiterung des aus Klassendiagrammen erzeugten Generats wird am Beispiel von Statecharts erläutert, lässt sich aber analog auch für Ergänzungen des Produktivcodes durch die Generate von OCL/P, Java/P oder Objektdiagrammen anwenden. Die Generierung von Testcode wird hingegen in Abschnitt 6.5 gesondert behandelt.

6.3.1 Delegator-basierte Komposition mit Factories

Bei diesem Ansatz wird eine erweiterbare Struktur aus Klassendiagrammen generiert, die durch Generate anderer Modelle ergänzt werden kann. Die Grundidee der nachträglichen Verhaltensergänzung von Klassen basiert dabei auf dem Konzept der Delegation und der Methodensignatur als innerhalb der UML/P bekannte Schnittstelle. Dazu werden folgende Elemente für jede Klasse eines Klassendiagramms generiert:

1. *Interface*, das hinsichtlich Name und Methoden den Angaben der modellierten Klasse entspricht.
2. *Implementierung des Interfaces* aus Punkt 1, die die eigentliche Ausführungslogik enthält. Der Name kann beliebig gewählt werden, da er ausschließlich dem Generator bekannt sein muss und auf der Modellierungsebene keine Rolle spielt. Im Folgenden wird er aus dem Namen der Klasse im Klassendiagramm mit der zusätzlichen Endung “Impl” abgeleitet. Methoden mit Methodenrumpf werden aus dem Klassendiagramm übernommen. Alle anderen Methoden werden wie folgt umgesetzt:
   a) Methode wie im Klassendiagramm, wobei intern ein Delegator aufgerufen wird, der die Ausführungslogik der Methode enthält.
   b) Methode zum Setzen des Delegators.
3. *Factory* zur Instanziierung der Klassenimplementierung und der Delegatoren.

Abbildung 6.6 und 6.7 zeigen das Generat für einen Ausschnitt der TripleLogo-Applikation aus Quellcode 3.4. Diese Umsetzung wird erst durch die Entkopplung der modellierten von der generierten Architektur möglich. Die Klassendiagramme geben somit nur den von außen beobachtbaren Rahmen vor, der von den Generatoren für das Zielsystem für unterschiedliche Zwecke wie Testbarkeit oder, wie in diesem Fall, Erweiterbarkeit angepasst werden kann.

Durch die in Abschnitt 3.7 diskutierte Anhebung von Java/P auf die Modellierungsebene kann die Struktur des Generats im Grunde beliebig von den Vorgaben der Modelle abweichen, solange die Funktionalität des generierten Systems den Modellen nicht widerspricht (siehe auch Abschnitt 3.9). Das Generat selbst spielt nur noch für die Systemausführung und für die Entwicklung der Generatoren eine Rolle. Spezifiziert und implementiert wird das System hingegen mit der UML/P, ohne dass sich der Modellierer mit dem generierten Code auseinander setzen muss. Damit lösen Modelle den Systemcode als Hauptartefakt der Softwareentwicklung ab [MB02, Rum02]. Allerdings unterstützt eine der modellierten Architektur ähnliche Struktur des Generats dessen Verständlichkeit und damit die Wart- und Weiterentwickelbarkeit der Generatoren, für die

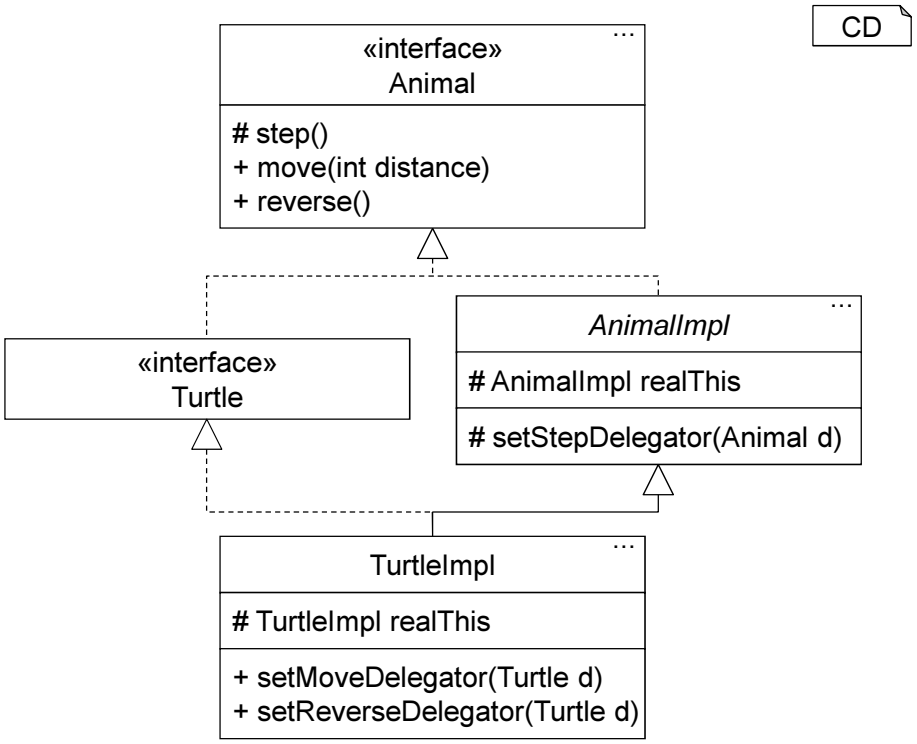


Abbildung 6.6: Architekturübersicht des Generats aus Klassendiagrammen. Dargestellt ist ein Teil der Klassen des Zielsystems, die aus Quellcode 3.4 generiert werden.

die Betrachtung des generierten Codes als Produkt der Generierung essentiell ist. Die strukturelle Ähnlichkeit ist auch dann von Vorteil, wenn das Generat als Framework außerhalb der UML/P eingesetzt werden soll. Aus diesem Grund wird im vorliegenden Ansatz für jede Klasse ein Interface generiert. Wie Abbildung 6.6 zeigt, bildet deren Aufbau, Vererbungsstruktur und Assoziationen die modellierte Architektur nach. So entsteht eine dem Klassendiagramm entsprechende Schnittstelle zu dem generierten System, die die darunter liegende, vom Klassendiagramm abweichende Architektur verbirgt. Dies wird durch die in Abbildung 6.7 dargestellten Factories unterstützt, die nach außen nur den Typ des Interfaces bei der Instanziierung bekannt geben.

Ist im Klassendiagramm nur die Signatur einer Methode, nicht aber deren Implementierung spezifiziert, wird innerhalb des generierten Methodenrumpfes stattdessen ein Delegator aufgerufen. Dieser wird bei der Instanziierung der Klasse durch die Factory besetzt. Dazu setzt sich jede Factory wiederum aus mehreren Teil-Factories zusammen (`PartFactory`, siehe Abbildung 6.7), die für die Erzeugung der Delegatoren zuständig sind. Diese werden erst zur Laufzeit des Systems ermittelt. Auf diese Weise muss zum Zeitpunkt der Generierung aus Klassendiagrammen nicht feststehen, welche Delegatoren das Verhalten einer Klasse vervollständigen. Die Klassen `Factory` und `PartFactory` stehen als Framework innerhalb einer mit der UML/P ausgelieferten Laufzeitumgebung zur Verfügung (siehe Abschnitt 7.1). Von diesen werden generierte Factories

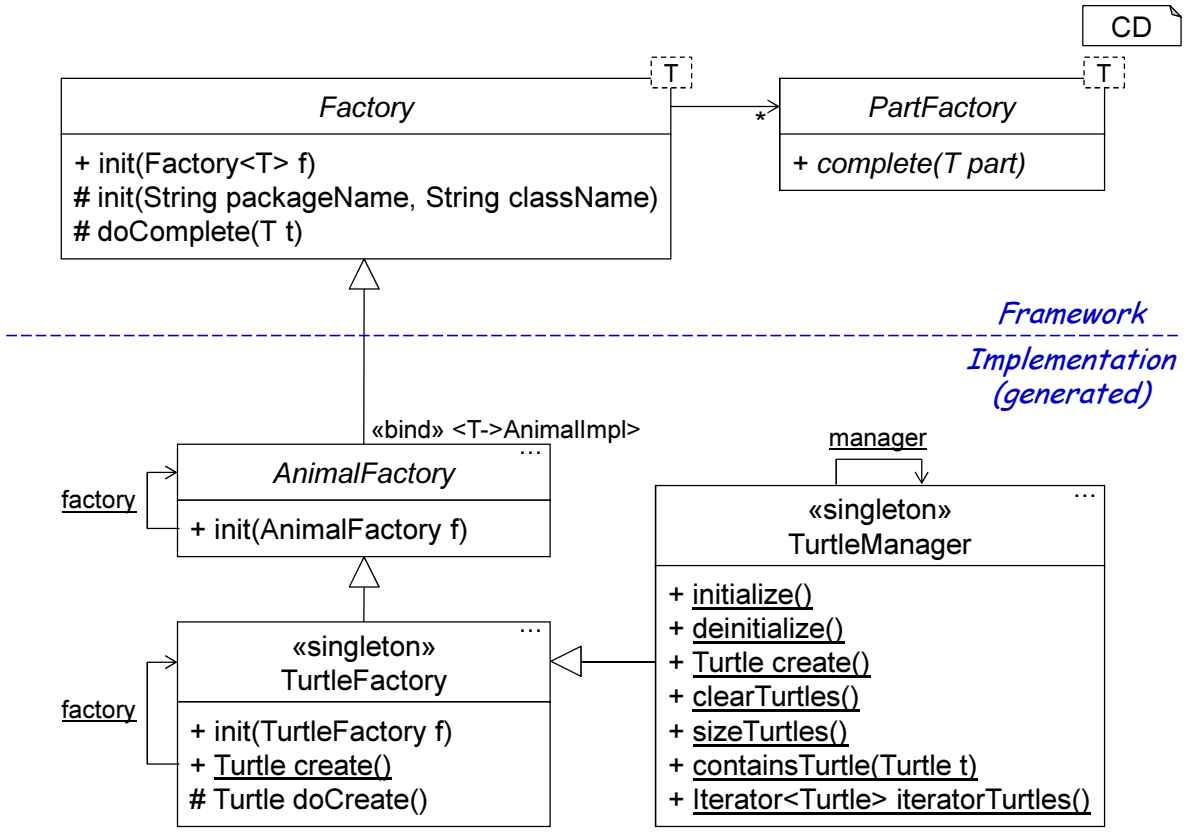


Abbildung 6.7: Architekturübersicht des Generats von Factories aus Klassendiagrammen. Dargestellt ist ein Teil der Klassen des Zielsystems, die aus Quellcode 3.4 generiert werden.

und Teil-Factories abgeleitet, so dass nur der für die jeweilige Objekterzeugung spezifische Anteil generiert werden muss.

Abbildung 6.8 zeigt das Generat aus dem Statechart in Quellcode 3.15, das das Verhalten der Klasse `Turtle` spezifiziert. Durch den oben beschriebenen Aufbau kann dieses nun ebenfalls allein aus den Informationen des Statecharts erstellt werden, ohne dass eine zeitgleiche Verarbeitung des entsprechenden Klassendiagramms notwendig ist. Das im Statechart spezifizierte Verhalten wird in die als Delegator verwendete Klasse `TurtleBehavior` generiert. Die genaue Umsetzung ist an dieser Stelle nicht relevant. Die Literatur bietet verschiedene Ansätze für die Codegenerierung aus Statecharts wie das Statepattern [GHJV95] oder die in [Rum04b] aufgeführten leichtgewichtigen Ansätze, die hier verwendet werden können.

Der Delegator kapselt somit das Verhalten einer oder mehrerer Methoden einer Klasse. Da dafür auch häufig der Zugriff auf deren Attribute notwendig ist, besitzt jede generierte Klasse außerdem mit `realThis` einen Zeiger auf die eigene Instanz. Im Fall der Delegatoren wird dieser auf die Hauptinstanz gesetzt, wobei der Generator sicherstellt, dass Zugriffe auf geerbte Attribute ausschließlich über den Zeiger erfolgen.

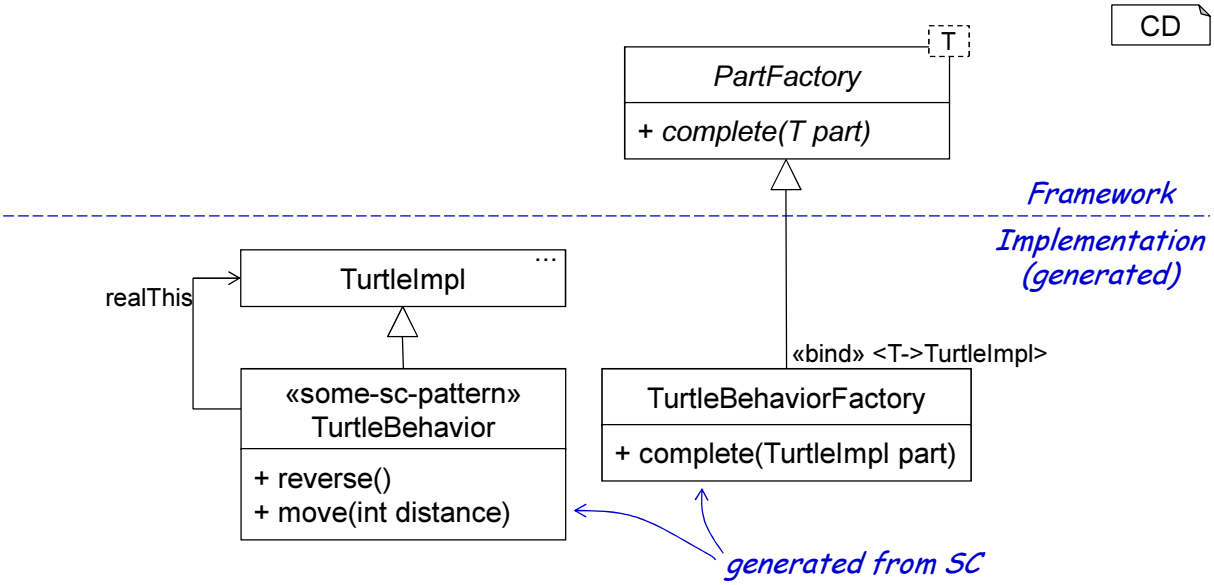


Abbildung 6.8: Architekturübersicht des Generats aus Statecharts am Beispiel von Quellcode 3.15

Den Ablauf bei der Zusammenstellung von Hauptinstanz und ergänzenden Delegatoren während der Objekterzeugung zeigt Abbildung 6.9 am Beispiel des Typs `Turtle`. In einem ersten Schritt wird die intern verwendete Instanz der Factory initialisiert und die notwendigen Teil-Factories ermittelt. Durch Nutzung der Vererbungsstruktur der Factory wird dabei ebenfalls geerbtes Verhalten von `Animal` berücksichtigt. Diese Initialisierung erfolgt nur beim ersten Aufruf der statischen Methode `create()` und ist danach für alle folgenden Objekterzeugungen festgelegt. Aus Effizienzgründen wurde in der vorliegenden Arbeit die Suche nach den Teil-Factories auf das Paket der Hauptinstanz und auf Klassen, die einem bestimmten Namensschema folgen, eingeschränkt. Als Namensschema dient der Name des zu erzeugenden Typs als Prefix und das Postfix "Factory". Da sich diese Einschränkung in den Generatoren manifestiert, handelt es sich dabei nur um einen Vorschlag, der für andere Codegenerierungen jederzeit abgewandelt werden kann.

Nach erfolgter Initialisierung der Factoryinstanz wird auf dieser die Methode `doCreate()` aufgerufen, die die Hauptinstanz `TurtleImpl` erzeugt und an die Teil-Factories weiterreicht. Diese fügen die Delegatoren hinzu und setzen bei diesen gleichzeitig den Zeiger `realThis` auf die Hauptinstanz.

Neben der flexiblen Erweiterung von generierten Klassen um Verhalten bietet die Instanziierung über Factories weitere Vorteile. Abbildung 6.6 zeigt hierfür beispielhaft die Klasse `TurtleManager`. Diese verhält sich ähnlich wie die Klasse `TurtleFactory`, protokolliert jedoch zusätzlich jede erzeugte Instanz. Auf diese Weise entsteht ein zentraler Zugriffspunkt für alle Instanzen eines Typs, der z.B. für die Überprüfung von Klasseninvarianten der OCL/P verwendet werden kann. Um den Manager einzusetzen, genügt ein einmaliger Aufruf der statischen Methode `initialize()`. Dieser Aufruf setzt den `factory`-Zeiger von `TurtleFactory` auf den Manager.

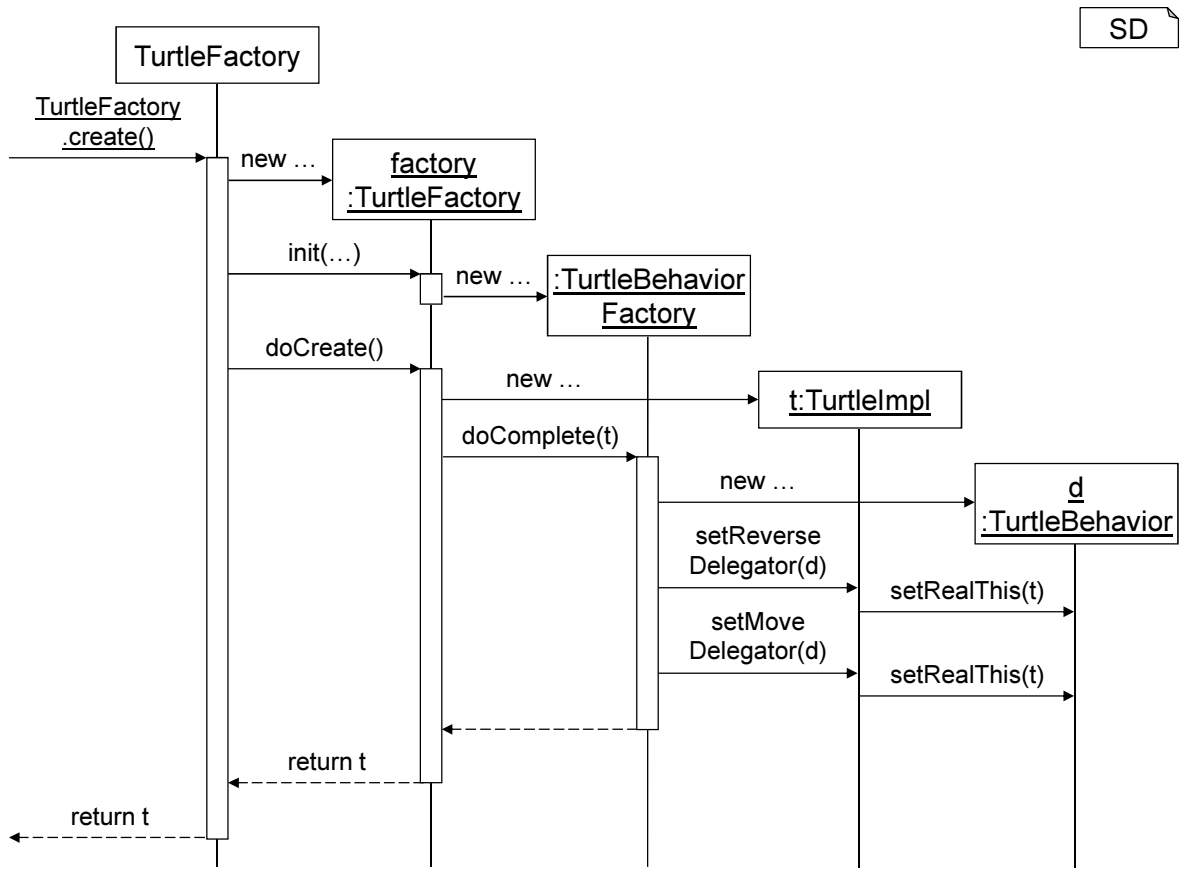


Abbildung 6.9: Klassenkomposition mit Hilfe von Factories: Vervollständigung von Objekten im generierten Zielsystem am Beispiel von Quellcode 3.4

Dadurch wird für alle zukünftigen Objektinstanziierungen über die Factory intern automatisch der Manager verwendet und die Instanzen protokolliert. Ein ähnliches Verfahren wird auch im Rahmen von Tests für die Codeinstrumentierung eingesetzt, die in Abschnitt 6.4 behandelt wird. Dabei handelt es sich um eine Variante zu dem hier vorgestellten Erweiterungskonzept von Klassen, das ebenfalls auf die Komposition über Factories aufsetzt, aber ohne festgelegte Delegatorschnittstellen auskommt.

6.3.2 Delegator-basierte Komposition per Namenskonvention

Wie die in Abschnitt 6.3.1 vorgestellte Codegenerierung basiert auch dieser Ansatz darauf, die separat erstellten Anteile einer Klasse per Delegation aufzurufen. Der Unterschied besteht darin, dass das Zusammensetzen dieser Anteile nicht mit Hilfe einer Factory erfolgt, sondern die Typnamen der Delegatoren anhand einer Namenskonvention festgelegt werden. Dadurch vereinfacht sich die Architektur des Generats etwas im Vergleich zu Abschnitt 6.3.1, so dass nur folgende Elemente aus einer Klasse generiert werden müssen:

1. *Default-Delegatoren* für jede Methode einer Klasse. Diese werden nur erzeugt, wenn sie nicht bereits im System vorliegen und müssen sich über die Namenskonvention eindeutig einer Methode zuordnen lassen. Die Default-Delegatoren können von anderen Generatoren mit einer konkreten Implementierung überschrieben werden.
2. *Klasse*, deren Aufbau im Wesentlichen den Angaben im Klassendiagramm entspricht. Zusätzlich wird das Attribut `realThis` für den Zugriff der Delegatoren auf die Attribute der Hauptklasse erzeugt. Funktion und Anwendung von `realThis` entsprechen demnach dem Ansatz in Abschnitt 6.3.1. Für nicht-abstrakte Methoden ohne Rumpf im Klassendiagramm wird im Generat ein Rumpf ergänzt, in dem der zugehörige Delegator instanziiert und aufgerufen wird. Die Default-Delegatoren sorgen somit dafür, dass das generierte System in jedem Fall kompilierbar ist, auch wenn noch kein Verhalten für eine Methode spezifiziert wurde.

Abbildung 6.10 zeigt das Generat eines Ausschnitts der TripleLogo-Applikation. Die Namen der Delegatoren wurden hier anhand der Klassen- und Methodensignatur festgelegt, die bei der Generierung der Methodenimplementierung aus den Statecharts ebenfalls eingehalten werden müssen.

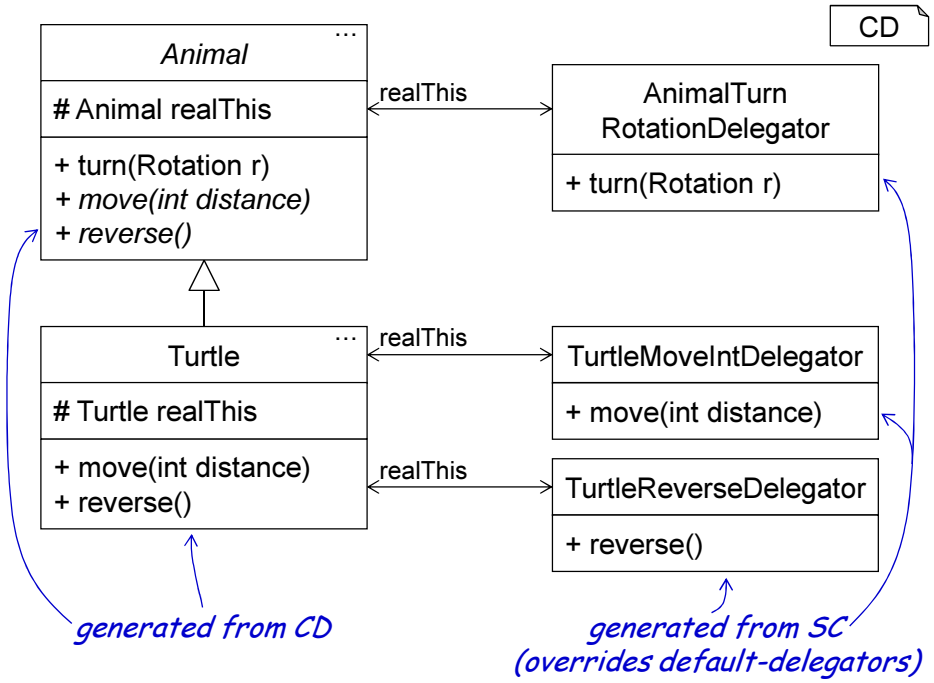


Abbildung 6.10: Komposition des Generats mit festgelegten Delegatoren. Dargestellt ist ein Teil der Klassen des Zielsystems, die aus Quellcode 3.4, 3.15 und 3.17 generiert werden.

Die im Vergleich zu Abschnitt 6.3.1 vereinfachte Architektur des Generats steht dem Nachteil gegenüber, dass generierte Dateien von anderen Generatoren überschrieben werden müssen. Dies

kann zu Problemen bei der Generierung führen. Eine Überprüfung auf Existenz verhindert zwar, dass der aus einem Statechart generierte Delegator von einer nachfolgenden Generierung aus den Klassendiagrammen mit dem Default-Delegator überschrieben wird. Andererseits müssen bei Änderungen der Methodensignaturen oder des Generators ebenfalls die Default-Delegatoren neu angelegt werden. Dies automatisch zu erkennen, erfordert komplexere Überprüfungen und führt damit zu einem gewissen Performance-Verlust bei der Generierung.

6.3.3 Aspektorientierte Komposition

Wurde in den bisherigen Abschnitten 6.3.1 und 6.3.2 bei der Generierung das objektorientierte Paradigma für die Strukturierung des Zielcodes genutzt, wird im Folgenden auf das Konzept der *aspektorientierten Programmierung* (AOP, [KLM+97]) aufgesetzt. Entwickelt wurde diese Idee aus der Beobachtung heraus, dass sich manche Anforderungen an objektorientierte Systeme nicht ohne Codeduplizierung umsetzen lassen. Als ein Standardbeispiel wird häufig die Protokollierung von Methodenaufrufen etwa für Logdateien oder Testfälle genannt. Für diese sogenannte Instrumentierung des Codes muss in jeder Methode ein Protokollaufruf eingefügt werden[3]. Solche Anforderungen werden auch als *Crosscutting Concerns* bezeichnet. Mit Hilfe der aspektorientierten Programmierung können diese Crosscutting Concerns in eigenen Dateien, den *Aspekten* gekapselt werden. Diese bestehen aus dem Code für die Anforderung und Regeln, wann dieser im Programmablauf aufzurufen ist.

Auch wenn modelliertes Verhalten und Bedingungen in der UML/P auf konkrete Typen oder Methoden Bezug nimmt und es sich damit nicht um Crosscutting Concerns handelt, können die Mechanismen von AOP doch dazu eingesetzt werden, um separat generierte Anteile verschiedener Modelle im Zielsystem zu vereinen. Dafür werden die in den Klassendiagrammen spezifizierten Typen und Methoden als Ankerpunkte für die als Aspekte abgelegten Verhaltensbeschreibungen oder Bedingungen verwendet.

Für Java existiert mit AspectJ eine Implementierung des aspektorientierten Programmierparadigmas [KHH+01], das im Folgenden als Zielplattform für die Generierung eingesetzt wird. Mit Hilfe dieser Technologie kann die in den Klassendiagrammen modellierte Architektur für das Zielsystem vollständig übernommen werden. Einzig fehlende Methodenrümpfe nicht abstrakter Methoden sind mit einer Default-Implementierung zu ergänzen, damit der Zielcode kompilierbar ist. Insgesamt kann dabei das Generat aus den Klassendiagrammen ohne AspectJ-spezifischen Code in reinem Java umgesetzt werden. Dieser kommt erst für Ergänzungen der Architektur um Bedingungen oder Verhalten zum Einsatz, die außerhalb des Klassendiagramms modelliert wurden.

Quellcode 6.11 zeigt eine Umsetzung des Statechart aus Quellcode 3.15 als Aspekt in AspectJ. Um das Verhalten für die Methode `reverse` der Klasse `Turtle` hinzuzufügen, wird in den Zeilen 5 und 6 ein sogenannter *Pointcut* definiert. Dieser kann mehrere Einzelereignisse wie den Aufruf einer Methode, die Verwendung einer Variable oder die Instanziierung eines Objekts

[3] Ein ohne AOP auskommendes Konzept für die Instrumentierung von Code wird in Abschnitt 6.4 behandelt.

umfassen. Die Ereignisse sind von AspectJ vorgegeben und werden als *Join Points* bezeichnet, die im Pointcut konkretisiert werden. Ein Pointcut stellt somit eine Art Ereignismuster dar, das während der Ausführung eines Programms auftreten kann. In diesem Fall ist es die Ausführung der Methode `reverse` auf einer Instanz des Typs `Turtle`. Letztere wird über die `target`-Anweisung an die Variable `realThis` gebunden, die später für den Zugriff auf die Instanz innerhalb des generierten Statechart-Codes genutzt wird. Die Variable erfüllt somit denselben Zweck wie schon das gleichnamige Attribut in den Abschnitten 6.3.1 und 6.3.2. Die zusätzliche Anweisung `within(Turtle)` ist notwendig, um ein Überschreiben der Methode in einer Subklasse von `Turtle` zu ermöglichen. Sie sorgt dafür, dass der Pointcut nur dann gültig ist, wenn der in `Turtle` spezifizierte Methodenrumpf ausgeführt werden soll.

AspectJ

```
1  package mc.tl;
2
3  public aspect TurtleBehavior {
4
5    pointcut pcReverse(Turtle realThis):
6      execution(void Turtle.reverse()) && target(realThis) && within(Turtle);
7
8    void around(Turtle realThis): pcReverse(realThis) {
9      reverse(realThis);
10   }
11
12   public void reverse(Turtle realThis) {
13     ... // generated code for reverse-behavior
14   }
15
16   pointcut pcMove(Turtle realThis, int distance):
17     execution(void Turtle.move(int)) && target(realThis) && within(Turtle)
18     && args(distance);
19
20   void around(Turtle realThis, int distance): pcMove(realThis, distance) {
21     move(realThis, distance);
22   }
23
24   public void move(Turtle realThis, int distance) {
25     ... // generated code for move-behavior
26   }
27
28 }
```

Quellcode 6.11: Statechart-Generat als Aspekt am Beispiel von Quellcode 3.15

Ein Pointcut erlaubt das Hinzufügen von *Advices*, die auszuführenden Code bei Auftreten des Pointcuts enthalten. Dieser wird entweder vor (`before`), nach (`after`) oder anstelle (`around`) des auslösenden Ereignisses ausgeführt. Um die Default-Implementierung im Methodenrumpf zu ersetzen, wird in Zeile 8-10 die `around`-Anweisung verwendet, die auf den zuvor definierten Pointcut Bezug nimmt.

Der Code für das Verhalten der Methode `reverse` wird in den Zeilen 12-14 als eigene Methode abgelegt. Dieser hätte auch direkt anstelle des Aufrufs in Zeile 9 eingefügt werden können. Stattdessen wurde eine bis auf den zusätzlichen Parameter `realThis` dem Klassendiagramm

entsprechende Methodensignatur gewählt, um die Lesbarkeit und Verständlichkeit sowohl des generierten Codes als auch der entsprechenden Templates zu verbessern. Um den Aspekt-spezifischen Code noch stärker zu kapseln, könnten auf die gleiche Weise die Methodenimplementierungen auch in eine eigene Klasse ausgelagert werden.

Die Umsetzung des Verhaltens der Methode `move` in den Zeilen 16-26 erfolgt analog mit dem einzigen Unterschied, dass zusätzlich dessen Argument durch die `args`-Anweisung in Zeile 18 gebunden werden muss.

Der Vorteil der aspektorientierten Programmierung liegt in der getrennten Generierung von Struktur und Verhalten des modellierten Systems, ohne dass wie in den Abschnitten 6.3.1 und 6.3.2 eine von den Klassendiagrammen abweichende Architektur oder ein Überschreiben von generierten Dateien notwendig ist. Dem gegenüber steht die Abhängigkeit von einer Technologie wie AspectJ, die nicht für jede Zielsprache zur Verfügung steht. So erfordert AspectJ den Einsatz eines eigenen Compilers, um Java-Code und Aspekte in Java-konformen Bytecode[4] zu übersetzen. Dieser ersetzt somit den Java-Compiler und fügt die Aspekte an die entsprechenden Stellen im Bytecode ein. Der Bytecode kann schließlich in einer regulären Java-Laufzeitumgebung ausgeführt werden, so dass die Abhängigkeit zu AspectJ nicht mehr zur Laufzeit des Systems besteht.

Die Umsetzung des aspektorientierten Programmierparadigmas über einen eigenen Compiler wie bei AspectJ bringt eine weitere Einschränkung mit sich, die bei den beiden Delegator-basierten Ansätzen nicht besteht. So muss bei der Kompilierung der Code, auf den sich Aspekte beziehen, vollständig vorliegen. Dadurch ist es nicht möglich, generierten Code aus einzelnen Modellen nur in kompilierter Form zur Verfügung zu stellen. Hinzu kommt, dass beim Generieren von Aspekten die davon betroffenen Klassen ebenfalls neu kompiliert werden müssen, auch wenn sich diese selbst nicht geändert haben. Die Nutzung von AspectJ bietet demnach keine vollständige Kompositionalität des Generats, da diese nur auf Ebene des generierten Codes, nicht aber auf Bytecode-Ebene besteht.

Alle drei vorgestellten Varianten für die Generierung von Systemcode aus UML/P-Klassendiagrammen ermöglichen die nachträgliche Ergänzung von Klassen um Verhalten und Bedingungen. Die Schnittstelle einer Klasse (auch Klassensignatur genannt), die sich aus den Attributen sowie Methoden- und Konstruktorsignaturen zusammensetzt, kann damit jedoch nicht erweitert werden. Um dennoch jedes Modell separat durch den Generator verarbeiten zu können, muss das Sichtenkonzept der UML/P so eingeschränkt werden, dass nur eine Definitionsstelle für jeden Typ existiert, die deren Signatur vollständig darstellt. Dies wurde bereits in in Abschnitt 4.3 diskutiert.

Bei der Generierung von Interfaces ist noch zu beachten, dass die vorliegende Fassung der UML/P Attribute als Teil der Schnittstellenbeschreibung erlaubt (siehe Abschnitt 3.9). Diese können bei der Verwendung von Java als Zielsprache nicht übernommen werden, da Java nur

[4]Beim Bytecode handelt es sich um das von der Hardware-Plattform unabhängige Ergebnis der Kompilierung. Dieser wird von der Java-Laufzeitumgebung bei der Systemausführung in den Hardware-abhängigen Maschinencode übersetzt.

Konstanten in Interfaces zulässt. Allerdings ist eine indirekte Umsetzung durch die Generierung entsprechender get- und set-Methoden im zugehörigen Interface des Zielsystems möglich.

6.4 Codeinstrumentierung

Die Instrumentierung von Code spielt eine wichtige Rolle bei der Überprüfung von Abläufen in einem Softwaresystem. Dabei soll der Aufruf von Attributen und Methoden, sowie die Instanziierung von Objekten bei der Systemausführung protokolliert werden, um anhand dieser Daten fehlerhafte Abläufe erkennen zu können. In der UML/P werden entsprechende Testfälle mit Hilfe von Sequenzdiagrammen modelliert. Deren Umsetzung in Testfälle des Zielsystems und die damit notwendige Instrumentierung des generierten Codes dient im Folgenden als Beispiel.

Die Codeinstrumentierung ist in diesem Fall nur notwendig, um die in den Sequenzdiagrammen modellierten Abläufe mit den tatsächlichen Abläufen bei der Ausführung des Systems im Rahmen von Testfällen zu vergleichen. Für die Nutzung des Systems ist dieser Vergleich nicht erforderlich. Nicht zuletzt aus Performance-Gründen soll deshalb die Codeinstrumentierung getrennt vom Systemcode generiert und nur bei Bedarf in diesen integriert werden.

Eine Möglichkeit für die Komposition von Systemcode und Instrumentierung ist das in Abschnitt 6.3.3 beschriebene aspektorientierte Programmierparadigma. Beim Einsatz von AspectJ wäre dadurch jedoch eine unterschiedliche Kompilierung von Test- und Produktivsystem erforderlich, um letzteres ohne die Protokollierung der Abläufe zu nutzen. Aus diesem Grund wird im Folgenden ein auf Factories basierendes Konzept vorgestellt, das auch eine Komposition bereits kompilierter Generate erlaubt. Das Konzept baut auf der in Abschnitt 6.3.1 beschriebenen Generierung des Systemcodes auf, lässt sich aber auch mit anderen Varianten umsetzen, wenn entsprechende Factories für die Objektinstanziierung eingesetzt werden.

Abbildung 6.12 zeigt die für die Codeinstrumentierung notwendige Architektur am Beispiel der TripleLogo-Applikation und des Sequenzdiagramms aus Quellcode 3.24. Für die Instrumentierung werden folgende Klassen zusätzlich zum Systemcode generiert:

- *Instrumentierte Subklassen* der Systemklassen. Dabei werden alle geerbten Methoden überschrieben. Diese rufen im Wesentlichen die jeweilige Methode der Superklasse auf, legen aber vor und nach diesem Aufruf einen entsprechenden Eintrag im `Logger` an. Attributzugriffe können über get- und set-Methoden und Instanziierungen über Konstruktoren in gleicher Weise protokolliert werden.
- *Factories* zur Aktivierung der Instrumentierung und Instanziierung der instrumentierten Subklassen.

Die Codeinstrumentierung wird als zusätzlicher Generator den Klassendiagrammen hinzugefügt, da dafür insbesondere Wissen über die Signatur der Klassen notwendig ist, die sich in den Sequenzdiagrammen nur indirekt ablesen lässt. Darüber hinaus kann auf diese Weise die Instrumentierung auch unabhängig von Sequenzdiagrammen eingesetzt werden.

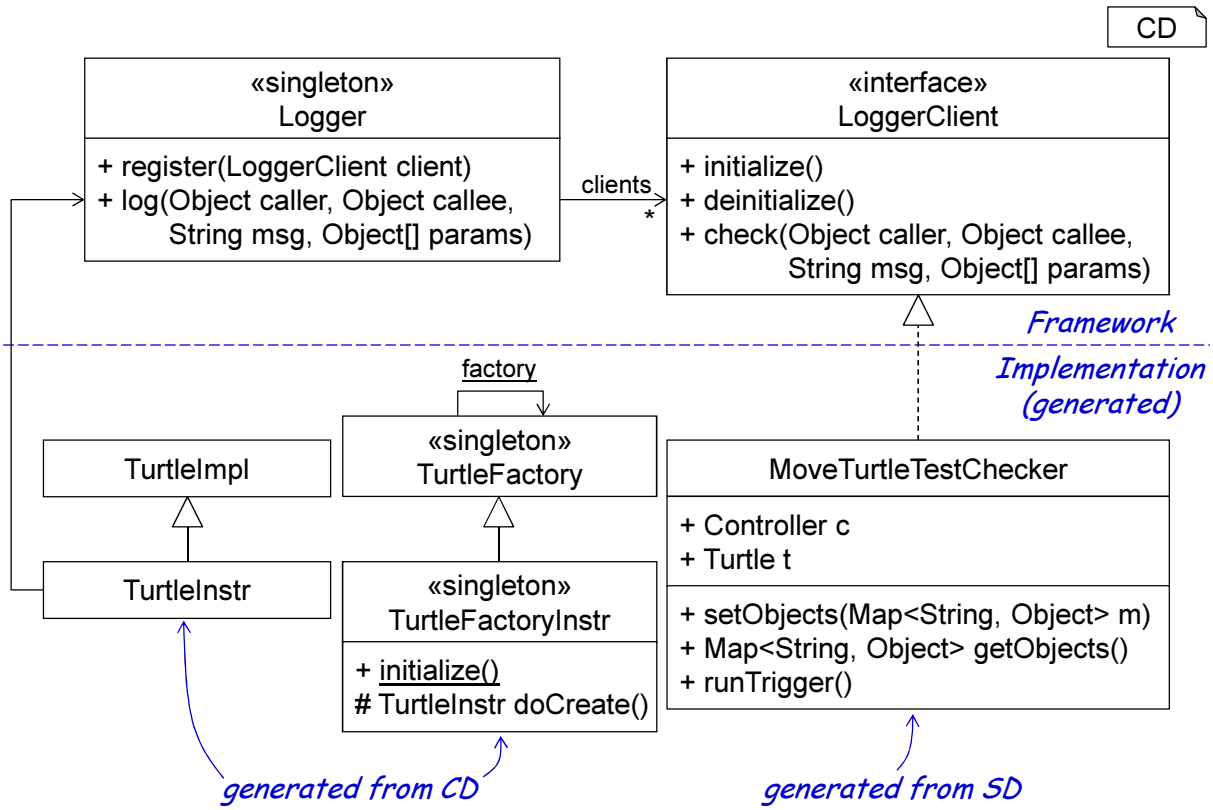


Abbildung 6.12: Architekturübersicht des Generats für die Codeinstrumentierung am Beispiel des Sequenzdiagramms aus Quellcode 3.24

Der aus einem Sequenzdiagramm generierte Code für den Vergleich der Log-Einträge mit den modellierten Abläufen wird als Implementierung des Interfaces `LoggerClient` umgesetzt. `Logger` und `LoggerClient` stehen dabei als zusätzliche Laufzeitumgebung im UML/P-Framework zur Verfügung (siehe Kapitel 7). Die dem `Logger` zugeordneten Clients werden bei Auftreten eines neuen Log-Eintrags aufgerufen. Dadurch kann auf Log-Einträge direkt reagiert und gleichzeitig mehrere Algorithmen pro Systemausführung ausgeführt werden. Darüber hinaus ermöglicht es diese offene Architektur, den Logging-Mechanismus flexibel einzusetzen. So sind auf diese Weise nicht nur Vergleiche für Sequenzdiagramme, sondern z.B. auch die Protokollierung von Systemabläufen in Dateiform, die Überprüfung von OCL/P-Bedingungen oder die Berechnung von Statistiken allein durch die Generierung entsprechender Clients möglich (siehe auch Abschnitt 6.6).

Um die Instrumentierung für eine Klasse zu aktivieren, genügt ein Aufruf der statischen Methode `initialize()` der Factory, die die entsprechenden instrumentierten Subklassen erzeugt. Am Beispiel in Abbildung 6.12 ersetzt dies die intern verwendete Instanz von `TurtleFactory` durch `TurtleFactoryInstr`, so dass künftig Objekte vom Typ `TurtleInstr` erzeugt werden. Der Einsatz der Instrumentierung erfolgt demnach ohne dass bestehender Systemcode angepasst werden muss. Dies entspricht dem Verfahren, dass auch schon beim Manager in Abschnitt 6.3.1 angewendet wurde. Der genaue Ablauf wird in Abschnitt 6.5 beschrieben.

Insgesamt ermöglicht es die hier vorgestellte Architektur, die Instrumentierung allein mit Hilfe objektorientierter Konzepte in Klassen zu kapseln. Dadurch lässt sie sich nicht nur bei der Ausführung des Systems bedarfsorientiert verwenden. Auch die Generatoren von Systemcode und Instrumentierung sind eigenständig und können unabhängig voneinander ausgeführt werden. Gleiches gilt für die Codegenerierung der Sequenzdiagramme, die auf die Infrastruktur der Instrumentierung aufsetzt. Die Zusammensetzung des Generats aus Sequenzdiagrammen und wie dieses im Rahmen von Testfällen mit Objektdiagrammen und der Testspezifikationssprache verwendet werden kann, wird in Abschnitt 6.5 diskutiert.

6.5 Generierung von Testcode

Testfälle können in der UML/P mit Hilfe von Objekt- und Sequenzdiagrammen spezifiziert werden. Aufgabe der Objektdiagramme ist es dabei, die Testdaten zu Beginn und das erwartete Ergebnis am Ende von Testfällen zu beschreiben. Sequenzdiagramme modellieren hingegen Ausführung und erwarteten Verlauf eines Tests. Um diese Teile zu einem vollständigen Testfall zu kombinieren, wurde die Testspezifikationssprache entwickelt (siehe Abschnitt 3.5). Wie bereits in den vorigen Abschnitten sollen sich auch hier die Modelle trotz dieser gemeinsamen Verwendung getrennt voneinander verarbeiten lassen, indem aus jedem Modell eigene Klassen generiert werden.

Abbildung 6.13 zeigt die Klassen, die aus dem Testfall in Quellcode 3.30 und den beiden Objektdiagrammen in Quellcode 3.12 und 3.31 generiert werden. Jedes Objektdiagramm wird dabei als eine Klasse umgesetzt, die sich sowohl zum Aufbau als auch zur Überprüfung der modellierten Objektstrukturen im Zielsystem einsetzen lässt. Auf diese Weise ist die Nutzung von Objektdiagrammen nicht nur im Rahmen von Testfällen möglich, sondern auch für den konstruktiven Einsatz im Produktivcode. So können etwa komplexe Objektstrukturen zu Beginn der Systemausführung aufgebaut oder innerhalb von OCL/P-Bedingungen verwendet werden, ohne dass andere Generatoren für Objektdiagramme notwendig sind. Demnach ist die Codegenerierung aus den Objektdiagrammen unabhängig von deren Verwendung.

Die in den Objektdiagrammen modellierten Objekte sind im Generat als Attribute einschließlich zugehöriger get- und set-Methoden umgesetzt. Durch einen Aufruf der `setup()`-Methode werden diese Attribute dem Modell entsprechend besetzt. Gleichzeitig wird eine Map bestehend aus den Objektnamen und den erstellten Objekten zurückgegeben, die alternativ zum Attributzugriff für die externe Verarbeitung verwendet werden kann. Entsprechend steht eine `setObjects`-Methode zur Verfügung, um Objekte der Klasse bekannt zu machen. Die `check()`-Methode vergleicht wiederum diese Objekte mit den modellierten Strukturen. Dabei auftretende Unterschiede werden als Nachrichten über den `MessageHandler` zur Verfügung gestellt.

Dieser Aufbau ähnelt dem Generat aus Sequenzdiagrammen in Abbildung 6.12. Auch hier werden die beteiligten Objekte als Attribute umgesetzt. Darüber hinaus realisieren die folgenden Methoden Ausführung und Überprüfung der modellierten Abläufe im System mit Hilfe der in Abschnitt 6.4 beschriebenen Codeinstrumentierung:

- `initialize()`: Einrichtung der erforderlichen Codeinstrumentierung

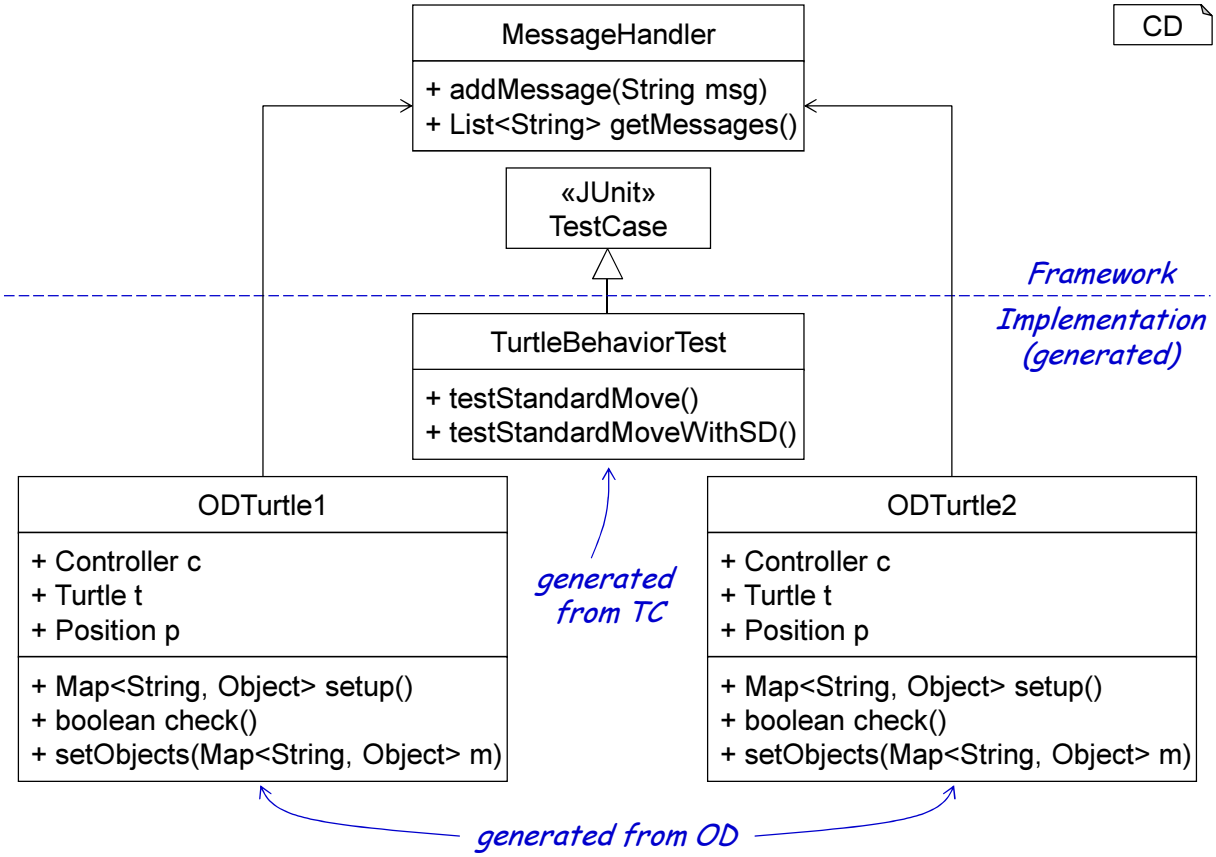


Abbildung 6.13: Architekturübersicht des Generats aus Objektdiagrammen und der Testspezifikationssprache am Beispiel von Quellcode 3.12, 3.30 und 3.31

- `runTrigger()`: Ausführung der Auslöser des modellierten Ablaufs (Trigger)
- `check(...)`: Vergleich des Modells mit den Abläufen bei Systemausführung

Die Referenzierung von Elementen wie Typen oder Objekte erfolgt in der vorliegen Fassung der UML/P über deren voll-qualifizierten Namen, die über Import-Angaben vereinfacht werden können (siehe Abschnitt 3.8). Anonyme Objekte sind demnach außerhalb des zugehörigen Objekt- bzw. Sequenzdiagramms nicht bekannt. Entsprechend stehen diese auch nicht als Attribute im Generat zur Verfügung, werden aber intern für Objektaufbau und -vergleich berücksichtigt, so dass darauf indirekt über Attribute oder Assoziationen benannter Objekte zugegriffen werden kann.

Auch in der Testspezifikationssprache wird auf Objekt- und Sequenzdiagramme sowie Objekte über deren Namen Bezug genommen (siehe Abschnitt 3.5). Dies ermöglicht nicht nur eine eindeutige und nachvollziehbare Referenzierung der in den modellierten Testfällen verwendeten Modelle und Modellelemente. Gleichzeitig können die Namen innerhalb der Codegenerierung als bekannte Schnittstellen vorausgesetzt und so die Testfälle unabhängig von den referenzierten Elementen vom Generator verarbeitet werden.

Ein Testfall in der Testspezifikationssprache wird vom Generator als Subklasse von `TestCase` umgesetzt. Diese Klasse ist Teil des JUnit-Frameworks, das in Java die Implementierung von Testfällen unterstützt [Lin05]. Für jeden Unit-Test wird in der Subklasse eine Methode erzeugt, die den Code für die Testausführung enthält. Den modellierten Tests entsprechend werden dabei die aus den referenzierten Objekt- und Sequenzdiagrammen erzeugten Klassen verwendet. Die Testausführung erfolgt schließlich über das JUnit-Framework allein auf Basis des generierten Codes. Daher sollte bei der Implementierung des Generators darauf geachtet werden, beim Fehlschlagen eines Testfalls auf die Modelle als die eigentlichen Quellen des Testfalls Bezug zu nehmen. Nur so kann der Modellierer anhand der Modelle das Testresultat nachvollziehen. Abbildung 6.14 zeigt dies anhand der in Quellcode 3.30 modellierten Tests am Beispiel einer unerwarteten Ausrichtung der Schildkröte nach Testausführung.

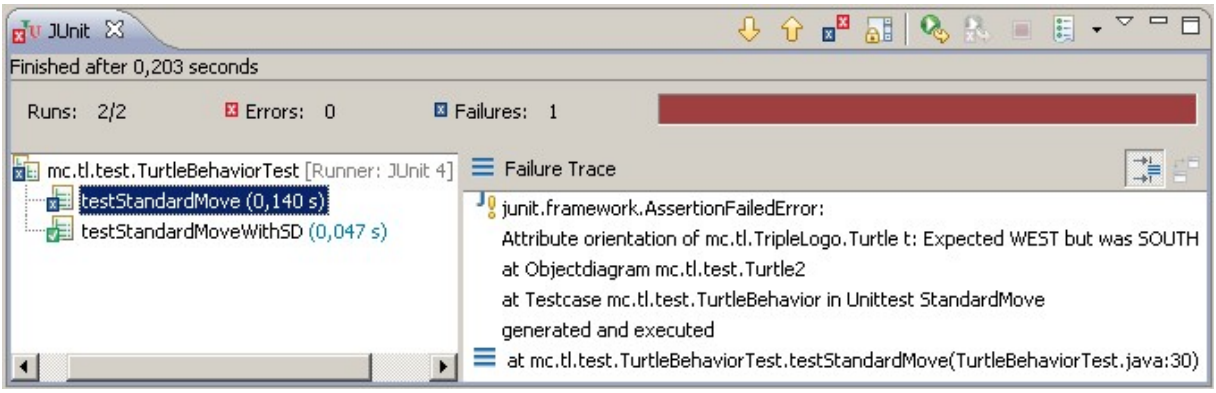


Abbildung 6.14: Screenshot eines fehlgeschlagenen JUnit-Tests auf Modellbasis

Der Ablauf bei Ausführung des generierten Testcodes im System wird in Abbildung 6.15 dargestellt. Dabei handelt es sich um einen Ausschnitt des zweiten in Quellcode 3.30 modellierten Unit-Tests, der neben den bereits erwähnten Objektdiagrammen das Sequenzdiagramm `MoveTurtleTest` aus Quellcode 3.24 als Ablaufbeschreibung einsetzt. Somit zeigt das Beispiel ebenfalls die Verwendung des aus Sequenzdiagrammen erstellten Generats (siehe Abbildung 6.12) sowie die in Abschnitt 6.4 diskutierte Codeinstrumentierung.

Nach dem Aufruf der Testmethode `testStandardMoveWithSD()` durch das JUnit-Framework wird die Protokollierung der für den Testfall notwendigen Systemteile gestartet, indem das Generat aus dem Sequenzdiagramm `MoveTurtleTest` als Client im `Logger` registriert und die Factories für die instrumentierten Klassen initialisiert werden. Letzteres ist in Abbildung 6.15 nur für den Typ `Turtle` aufgeführt, betrifft aber alle der im Sequenzdiagramm `MoveTurtleTest` verwendeten Typen. Durch die Initialisierung von `TurtleFactoryInstr` wird eine Instanz dieser Factory erstellt, die die ansonsten intern für die Objekterzeugung des Typs `Turtle` verwendete Instanz in `TurtleFactory` ersetzt (siehe Abschnitt 6.4). Wie der Aufbau der im Objektdiagramm `Turtle1` beschriebenen Objektstruktur durch den `setup()`-Aufruf zeigt, werden dadurch alle

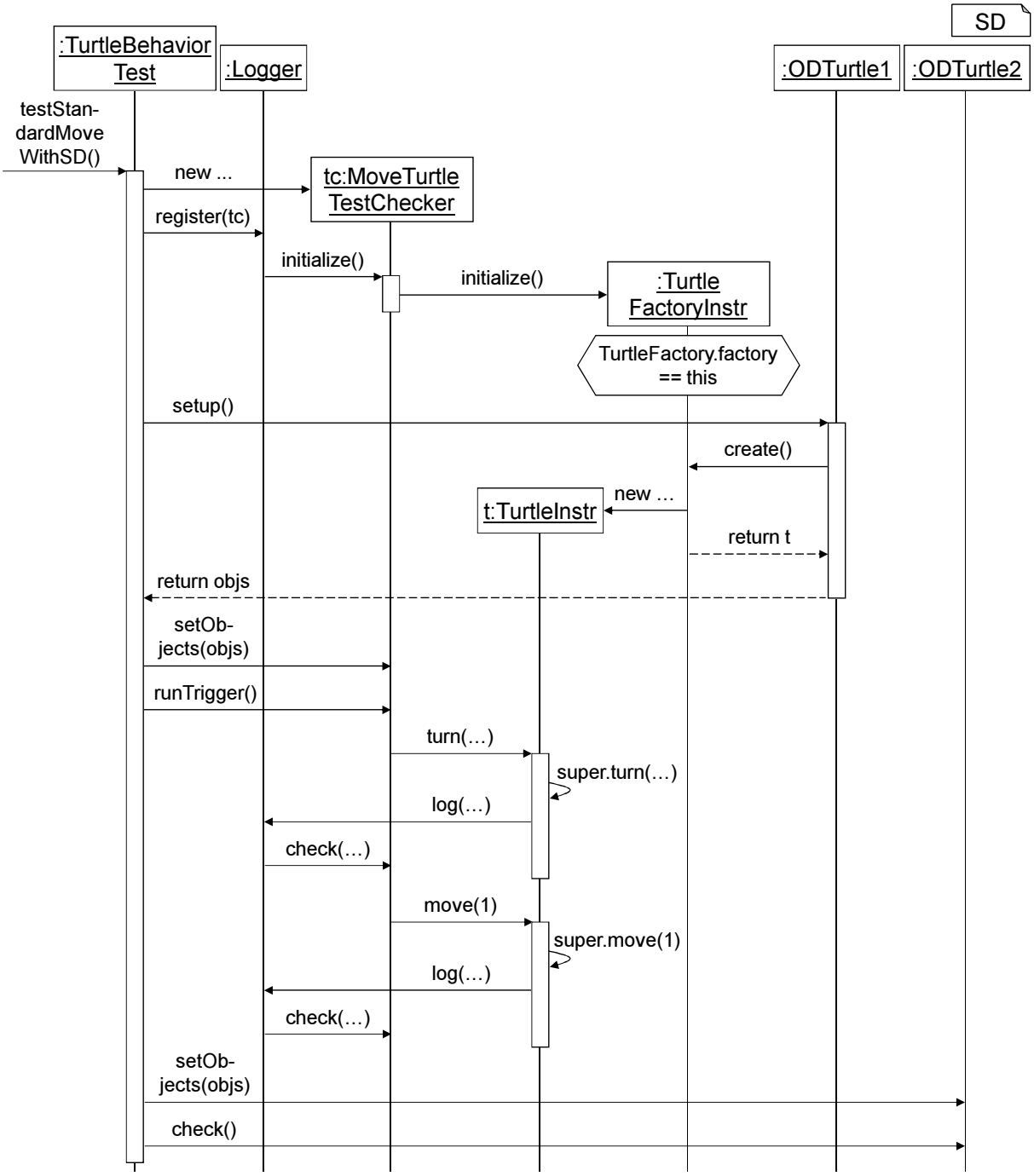


Abbildung 6.15: Ausschnitt eines Testfallablaufs im generierten Zielsystem

folgenden Aufrufe der statischen `create()`-Methode der `TurtleFactory` automatisch intern von `TurtleFactoryInstr` bearbeitet.

Die für den modellierten Testablauf benötigten Objekte werden schließlich an das Sequenzdiagramm-Generat weitergereicht und die Testtreiber aufgerufen, die in der Methode `runTrigger()`

gekapselt sind. Die Instrumentierung sorgt beim folgenden Systemablauf dafür, dass dieser mit dem im Sequenzdiagramm modellierten Ablauf verglichen werden kann. Auch wenn in Abbildung 6.15 nicht explizit aufgeführt, schließt dies die Kommunikation zwischen `Turtle` und `Controller` mit ein (vgl. Quellcode 3.24).

Der Vergleich der finalen Objektstruktur nach Abschluss der Systemaufrufe mit dem erwarteten Ergebnis im zweiten Objektdiagramm erfolgt nach der Übertragung der Objekte über dessen Generat `ODTurtle2`.

Obwohl sich ein Testfall in der Testspezifikationssprache aus Objekt- und Sequenzdiagrammen zusammensetzt, erlaubt die beschriebene Architektur eine unabhängige Verarbeitung der Modelle. Dies wurde durch die strikte Einhaltung zweier Prinzipien beim Entwurf der Generate erreicht:

1. Übertragung der Modularisierung der Modellebene auf die Ebene der Generate.
2. Schaffung von Schnittstellen zur Verbindung einzelner Generate, die sich allein aus den auf Modellebene bekannten Schnittstellen bzw. Referenzen ableiten.

Wie Abbildung 6.16 zeigt, können dabei durchaus mehrere Generate aus einem Modell entstehen (siehe auch Abschnitt 5.5). Ein Beispiel hierfür ist die aus den Klassendiagrammen generierte Systemarchitektur und die Codeinstrumentierung, die jeweils als eigene Generatoren umgesetzt wurden. Bei der Verarbeitung eines Modells werden jedoch keine Modellinhalte verwendet, die nicht in dem Modell selbst vorliegen. Stattdessen werden Schnittstellen in den Generaten zur Verfügung gestellt, die sich aus den schon auf Modellebene vorhandenen Referenzen ableiten lassen. Auf diese Weise sind die Modellinhalte auch für Generate anderer Modelle nutzbar, ohne die Inhalte selbst bekannt zu geben. So kapselt die aus Sequenzdiagrammen generierte Klasse die Trigger-Aufrufe, den Vergleich der modellierten Interaktionen mit den Systemabläufen sowie die dafür notwendige Initialisierung der Instrumentierung als eigene Methoden. Bei der Verarbeitung eines modellierten Testfalls wird aufgrund der Referenz auf ein Sequenzdiagramm auf das Vorhandensein einer entsprechenden Klasse im Zielsystem geschlossen. Das Wissen über die Architektur der aus den Modellen erstellten Generate ist demnach für Implementierung und Wartung der Generatoren notwendig und sollte entsprechend dokumentiert werden.

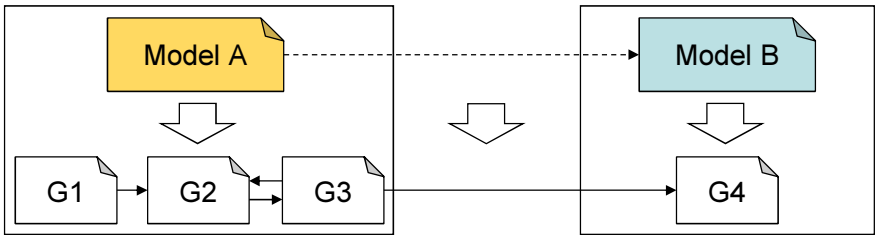


Abbildung 6.16: Übertragung von Modularisierung und Schnittstellen der Modell- auf Generatebene

Insgesamt führt dieses Vorgehen dazu, dass nur die seit der letzten Generierung geänderten Modelle erneut verarbeitet werden müssen, solange sich die Generatoren selbst nicht ändern. Die

Übertragung der Modularisierung verhindert, dass sich interne Änderungen eines Modells auf Generate anderer Modelle auswirken. Dies ist nur dann der Fall, wenn ebenfalls Schnittstellen betroffen sind. Da sich diese aber in den Referenzen auf der Modellebene widerspiegeln, werden durch die notwendigen Anpassungen der Modelle automatisch alle Generate neu erstellt, auf die sich die Schnittstellenänderung auswirkt. Die Konsistenz von Schnittstellen und Verwendung im Generat korrespondiert somit mit der Konsistenz von Modellelementen und deren Referenzierungen.

6.6 Testüberdeckung und -güte auf Modellen

Beim Einsatz von Testfällen für die automatisierte Qualitätssicherung eines Softwareprodukts stellt sich häufig die Frage nach der Qualität der Testfälle selbst. Um diese zu bestimmen, können sogenannte Softwarequalitätsmetriken eingesetzt werden. Dabei handelt es sich um Messungen, deren Ergebnisse die Beurteilung einer Software jenseits von funktionalen Eigenschaften zulassen. Allgemein können Metriken wie folgt definiert werden:

Definition 5 (Softwarequalitätsmetrik) *Funktion, die Daten über eine Software auf einen Zahlenwert abbildet. Dieser kann als Grad interpretiert werden, den die Software bezüglich einer gegebenen Qualitätseigenschaft erfüllt. (frei übersetzt nach [IEEE93])*

Als Qualitätseigenschaft im Sinne von Softwaretests wird in der Regel der prozentuale Anteil vom Systemcode betrachtet, der durch die Testfälle ausgeführt wird. Ziel dieser auch als Testüberdeckung oder Testabdeckung (engl. “Test coverage”) bezeichneten Eigenschaft ist es, den Systemcode möglichst vollständig zu testen. Der Grad der Überdeckung wird dabei anhand unterschiedlicher Überdeckungskriterien bestimmt:

- **Anweisungsüberdeckung** (engl. “Statement coverage”): Betrachtung der ausgeführten Anweisungen.
- **Zweigüberdeckung** (engl. “Branch coverage”): Betrachtung der bei der Ausführung verwendeten Verzweigungen im Systemcode. Im Gegensatz zur Anweisungsüberdeckung wird hierbei auch das Auslassen von bedingten Anweisungen mit einbezogen.
- **Bedingungsüberdeckung** (engl. “Decision coverage”): Betrachtung der bei der Testausführung genutzen möglichen Werte für in Bedingungen verwendete logische (Teil-)Ausdrücke.
- **Pfadüberdeckung** (engl. “Path coverage”): Betrachtung der in den Tests verwendeten möglichen Ausführungspfade durch den Systemcode.

Da in den meisten Softwaresystemen potentiell unendlich viele mögliche Pfade existieren, ist eine vollständige Pfadüberdeckung im Allgemeinen nicht möglich. Häufig wird deshalb die einfache Pfadüberdeckung angewendet, bei der jede Schleife nur mindestens einmal durchlaufen

werden muss. In der Literatur existieren weitere Varianten der oben aufgeführten Überdeckungskriterien [Lig09].

Die bisherigen Ausführungen beziehen sich auf die Qualitätsbeurteilung von codebasierten Tests. Beim Einsatz von modellbasierten Tests wie in der UML/P sollten die Überdeckungskriterien jedoch nicht auf dem generierten Code sondern auf den Modellen ermittelt werden. Andernfalls ist eine Beurteilung des Ergebnisses für den Modellierer, der im Allgemeinen keine Kenntnis über den generierten Code hat, nur schwer möglich. Vor diesem Hintergrund wurde bereits bei der Generierung des Testcodes darauf geachtet, dass sich Fehlerausgaben auf die Modelle beziehen (siehe Abschnitt 6.5). Mögliche Überdeckungskriterien für Tests von Klassendiagrammen und Statecharts können wie folgt aufgestellt werden:

1. **Klassendiagramme**:
   - *Typüberdeckung*: Anteil verwendeter Typen.
   - *Methoden- und Attributüberdeckung*: Anteil aufgerufener Methoden und Attribute.
   - *Assoziationsüberdeckung*: Anteil genutzter Assoziationen.
2. **Statecharts**:
   - *Zustandsüberdeckung*: Anteil besuchter Zustände.
   - *Transitionsüberdeckung*: Anteil genutzter Transitionen.
   - *Transitionspfadüberdeckung*: Anteil verwendeter Pfade von einem initialen zu einem finalen Zustand.

Daneben sind weitere Kriterien und Verfeinerungen denkbar. So kann die Testüberdeckung für Modelle oder deren Elemente jeweils getrennt ermittelt werden. Auch eine Bedingungsüberdeckung auf den Vorbedingungen der Transitionen ist möglich. Obwohl diese Überdeckungskriterien denen für Code recht ähnlich sind, müssen sie dennoch spezifisch für jede Sprache angepasst und implementiert werden. Um den dafür notwendigen Aufwand zu reduzieren, wird im Folgenden ein Konzept für eine sprachunabhängige und damit generische Definition von Metriken vorgestellt. Darauf aufbauend wurde das MBTM-Framework[5] entwickelt, das eine Wiederverwendung von Metrikberechnung und -auswertung ermöglicht. Dieses wurde im Detail bereits in [SV09] vorgestellt.

Das Konzept beruht im Wesentlichen auf einem gerichteten, hierarchischen Graphen als sprachunabhängige Zwischenstruktur. Die Berechnung und Auswertung der darauf definierten Metriken können für Sprachen wie die UML/P wiederverwendet werden, indem die Struktur jeder Sprache auf den Graphen übertragen wird. Anhand dieser im Folgenden als Mapping bezeichneten Abbildung wird während der Ausführung der aus den Modellen generierten Tests vom MBTM-Framework die Testüberdeckung berechnet und das Ergebnis in Form eines modellbezogenen Reports zur Verfügung gestellt (siehe Abbildung 6.17).

[5]Model-Based Testing Metrics

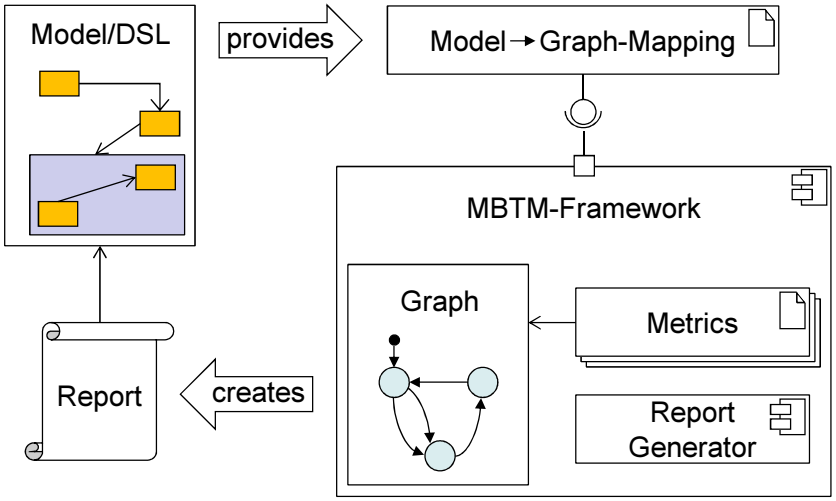


Abbildung 6.17: MBTM-Framework

Die im MBTM-Framework verwendeten Graphen eignen sich aus zwei Gründen als sprachunabhängige Struktur für die generische Definition und Berechnung von Metriken:

1. *Die Struktur einer Sprachinstanz (bzw. eines Modells) lässt sich als Graph abstrahieren*:
   Strukturell betrachtet besteht jede Instanz einer Sprache aus Elementen sowie Beziehungen zwischen diesen. Werden Elemente als Knoten und Beziehungen als Kanten dargestellt, entsteht eine abstrakte Darstellung der Struktur als Graph. Für eine flexiblere Darstellung setzen sich Graphen im MBTM-Framework aus hierarchischen Knoten und gerichteten Kanten zusammen. Darüber hinaus können mögliche Start- und Endknoten für Pfadberechnungen sowie unterschiedliche Arten von Knoten und Kanten definiert werden. Durch letzteres entsteht eine zusätzliche Typisierung und damit Unterscheidungsmöglichkeit innerhalb der Knoten- bzw. Kantenmenge. In der Graphentheorie werden diese auch als knoten- und kantengefärbte Graphen bezeichnet [Die10].
2. *Überdeckungskriterien lassen sich auf Graphen anwenden*:
   Das MBTM-Framework erlaubt es, auf erstellten Graphen sequenziell Knoten aufzurufen. Diese werden zusammen mit den zwischen zwei aufeinanderfolgenden Knotenaufrufen befindlichen Kanten markiert, wenn letztere eindeutig ist. Ansonsten können Kanten auch separat markiert werden. Auf diese Weise entstehen Mengen von markierten Knoten, Kanten und Pfaden innerhalb des Graphen. Anhand dieser können auf Graphen Überdeckungskriterien wie Knoten-, Kanten- und Pfadüberdeckungen berechnet werden, die den bereits diskutierten Kriterien ähneln. Die Ergebnisse lassen sich mit Hilfe des Mappings zwischen Sprach- und Graphstruktur auf der Sprachinstanz entsprechend interpretieren.

An Infrastruktur stellt das MBTM-Framework eine API-basierte Erstellung von Graphen, verschiedene Graphmetriken, parametrisierbare Report-Generatoren und eine Plugin-Architektur zur Anbindung von Editoren zur Verfügung [SV09]. Letzteres ermöglicht die Darstellung

und Auswertung der Metrikergebnisse innerhalb der Sprachinstanzen bzw. Modelle. Um diese Infrastruktur für die Qualitätsbeurteilung von Modell-bezogenen Tests einzusetzen, sind im Wesentlichen zwei Schritte erforderlich:

1. *Graph-Mapping*: Erstellung eines Graphen für jedes Modell.
2. *Graph-Markierung*: Markierung der Knoten im Graph, die den bei der Testausführung verwendeten Elementen entsprechen.

Im Hinblick auf den in dieser Arbeit verwendeten generativen Ansatz für die Umsetzung der Modelle in das Zielsystem können beide Schritte mit Hilfe von Generatoren umgesetzt werden. Wie Abbildung 6.18 zeigt, wird dabei der Code für die Erstellung des Graphen aus dem Modell generiert. Das Konzept für das Mapping zwischen Modell und Graph ist dementsprechend im Generator gekapselt. Die Markierung der Knoten während der Testausführung wird hingegen über eine Instrumentierung des generierten Systems vorgenommen. Wie sich diese unabhängig vom System selbst generieren und bedarfsorientiert einsetzen lässt, wurde bereits in Abschnitt 6.4 beschriebenen. Diese Infrastruktur kann hier wiederverwendet werden.

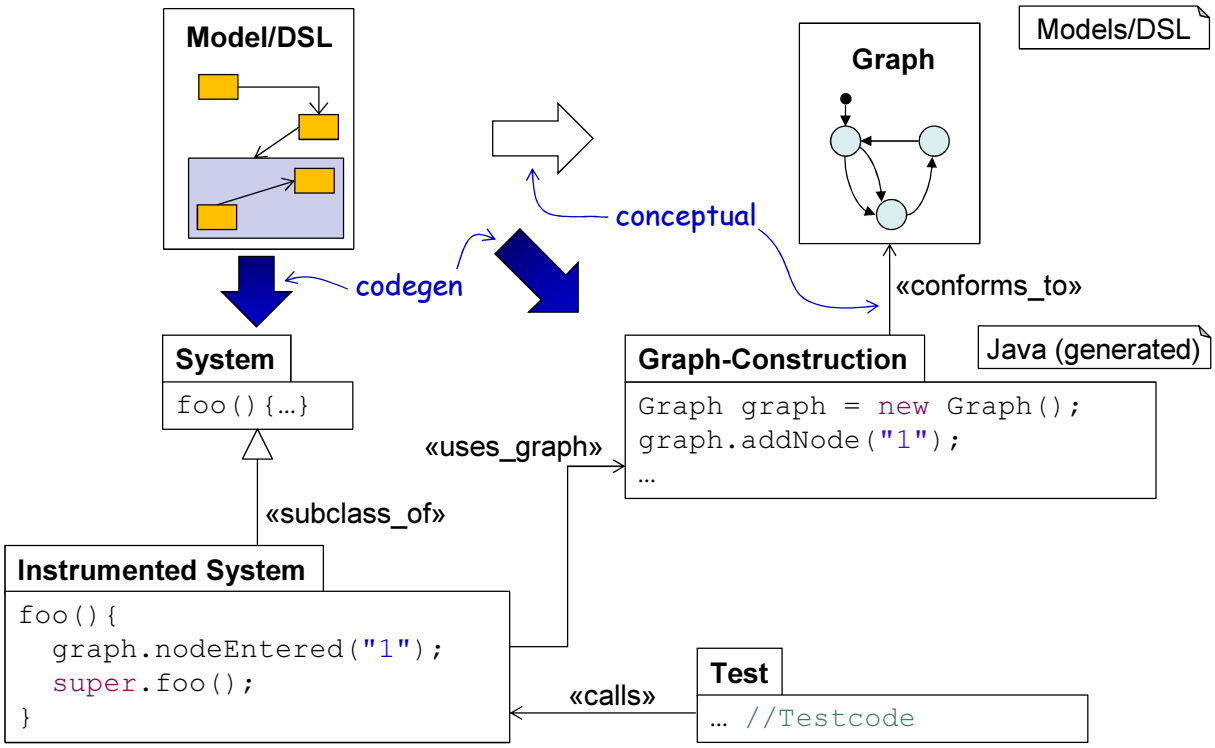


Abbildung 6.18: Testüberdeckung auf Modellen

Ein Mapping für Klassendiagramme zeigt Abbildung 6.19 am Beispiel eines Ausschnitts der TripleLogo-Architektur aus Abbildung 3.5. Dabei wurden Typdefinitionen, Attribute, Methoden und Konstruktoren jeweils als Knoten sowie Attribute und Vererbungsbeziehungen als Kanten

im Graphen dargestellt. Dieses Mapping stellt nur eine Möglichkeit dar, wie die Struktur eines Klassendiagramms auf einen Graphen übertragen werden kann. Attribute, Methoden und Konstruktoren könnten auch als innere Knoten des entsprechenden Typknotens dargestellt oder für die Elemente unterschiedliche Knotenarten eingesetzt werden. Handelt es sich bei der hierarchischen Darstellung um eine semantisch äquivalente Alternative zum flachen Graphen aus Abbildung 6.19, berechnet das Framework bei der Verwendung unterschiedlicher Knotenarten neben der Überdeckung aller Knoten zusätzlich eine Überdeckung für jede Art. Dasselbe gilt für die Definition von Kantenarten. Auf diese Weise kann etwa die Testüberdeckung von Methoden und Attributen getrennt ermittelt werden. Dementsprechend ist ein Mapping für eine Sprache nicht immer eindeutig und beeinflusst insbesondere den möglichen Detailgrad der Auswertung.

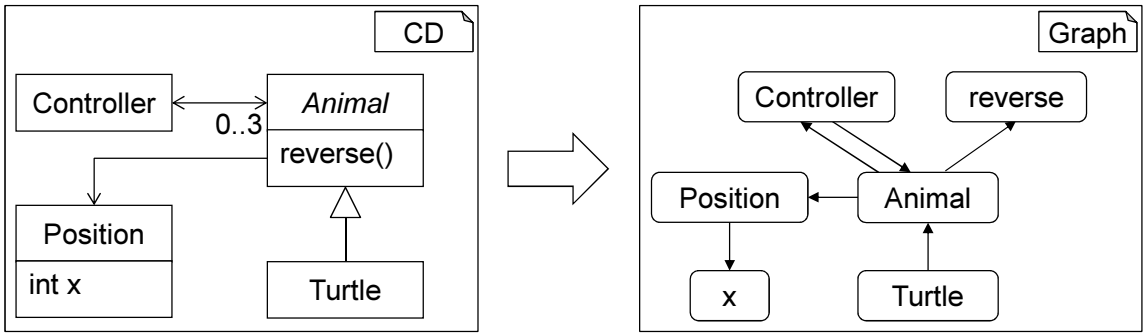


Abbildung 6.19: Abbildung von Modellen auf Graphen am Beispiel eines Klassendiagramms

Aufgrund ihrer ähnlichen Struktur lassen sich Statecharts mehr oder weniger direkt in Graphen übersetzen. Für eine Berechnung der Bedingungsüberdeckung auf den Transitionsvorbedingungen müssen diese jedoch ggf. in ihre Teilausdrücke zerlegt werden. Eine Bedingungsüberdeckung ergibt sich aus der Kantenüberdeckung, wenn für jede mögliche Belegung dieser Teilausdrücke eine Kante im Graphen angelegt wird (siehe Abbildung 6.20). In diesem Fall markiert die Instrumentierung nur die Kante, die der bei der Testausführung vorliegenden Belegung entspricht.

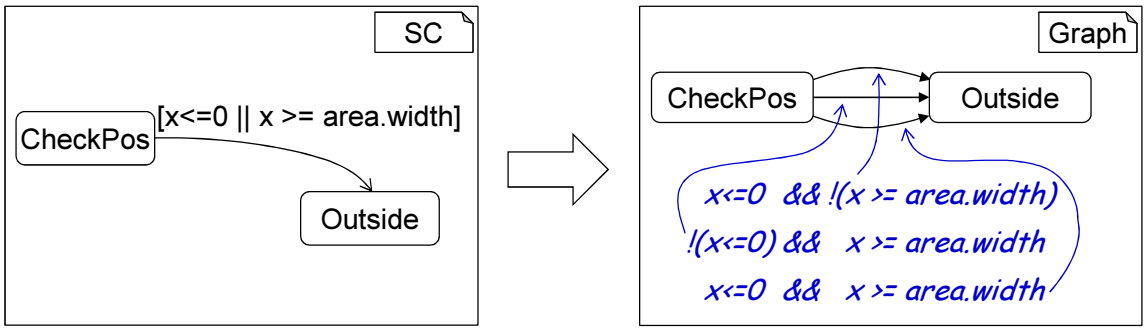


Abbildung 6.20: Bedingungsüberdeckung auf Statechart-Transitionen durch Graph-Mapping

In [SV09] werden mögliche Mappings für weitere Sprachen diskutiert und damit die breite Anwendbarkeit des Frameworks gezeigt. Für dessen Nutzung genügt es mit dem in der vorliegenden Arbeit verwendeten generativen Ansatz, einmalig Templates für Grapherstellung und Codeinstrumentierung für die Sprachen zu erstellen, auf denen Metriken berechnet werden sollen. Die dadurch generierte Infrastruktur kann schließlich bei der Ausführung der Testfälle gemeinsam mit dem MBTM-Framework genutzt werden, um automatisch die Testüberdeckung auf den entsprechenden Modellen zu bestimmen.

Für die Auswertung der Ergebnisse auf den Modellen bietet das MBTM-Framework wie bereits erwähnt eine Plugin-Architektur, die u.a. die Anbindungen von Editoren in Eclipse erlaubt. Dies eignet sich insbesondere für die Auswertung auf graphischen Modellen, wie Abbildung 6.21 am Beispiel eines Aktivitätsdiagramms zeigt. Nicht durch die Testfälle ausgeführte Knoten werden hier in rot direkt innerhalb des Modells angezeigt. Zusätzlich bietet das auf der rechten Seite in Abbildung 6.21 dargestellte Benutzerinterface die Möglichkeit, verwendete und nicht verwendete Transitionen und Pfade des Aktivitätsdiagramms oder nur bestimmte Typen von Knoten und Transitionen auszuwählen und entsprechend zu markieren. Auf diese Weise kann anhand der Metrikergebnisse die Testüberdeckung auf den Modellen untersucht und über die Erstellung zusätzlicher Testfälle gezielt verbessert werden.

Diese Anbindung an die Metrikberechnung des MBTM-Frameworks ist im Rahmen von [SV09] entstanden. Für die UML/P stehen noch keine graphischen Editoren zur Verfügung. Alternativ kann in den textuellen Editoren die Darstellung durch farbliche Hervorhebung von Wörtern erfolgen. Doch auch ohne eine Anzeige in den Editoren lassen sich die Ergebnisse anhand der vom MBTM-Framework generierten Reports auswerten. Die Abbildungen 6.22, 6.23 und 6.24 zeigen hierzu zum Vergleich die entsprechenden Reports aus der Testüberdeckung des Aktivitätsdiagramms. Da vom MBTM-Framework dabei ebenfalls der Bezug auf die Modelle eingehalten wird, lassen sich auch anhand der Liste in Abbildung 6.22 getestete und ungetestete Elemente identifizieren. Darüber hinaus bieten Diagramme wie in Abbildung 6.23 und 6.24 einen Überblick der aktuellen Testüberdeckung sowie deren zeitlicher Entwicklung. Diese dienen weniger der Verbesserung der Testqualität, sondern können innerhalb des Projektmanagements für Statusberichte und Monitoring eingesetzt werden.

Der Fokus der in diesem Kapitel vorgestellten Konzepte für die Architektur der Generate lag im Wesentlichen darauf, Modelle trotz gegenseitiger Verwendung und Referenzierung einzeln mit dem Generator verarbeiten zu können. Ohne bereits generierte Dateien zu verändern, findet die Komposition der modellierten Systemaspekte mit Hilfe objektorientierter Konzepte auf Ebene der Generate statt. Dies kommt einer Übertragung der Modularisierung der Modelle auf die der Generate gleich. Gleiches gilt für technische Aspekte wie die Codeinstrumentierung, die erst durch die Generierung hinzugefügt werden und sich durch die vorgestellten Konzepte ebenfalls als einzelne Generatoren umsetzen lassen. Auf diese Weise entstehen für Modellarten und technische Aspekte mehrere separate Generatoren, die sich bei Bedarf kombinieren oder austauschen lassen. Dies bietet nicht nur Vorteile für die Wartung und Weiterentwicklung der Codegenerierung,

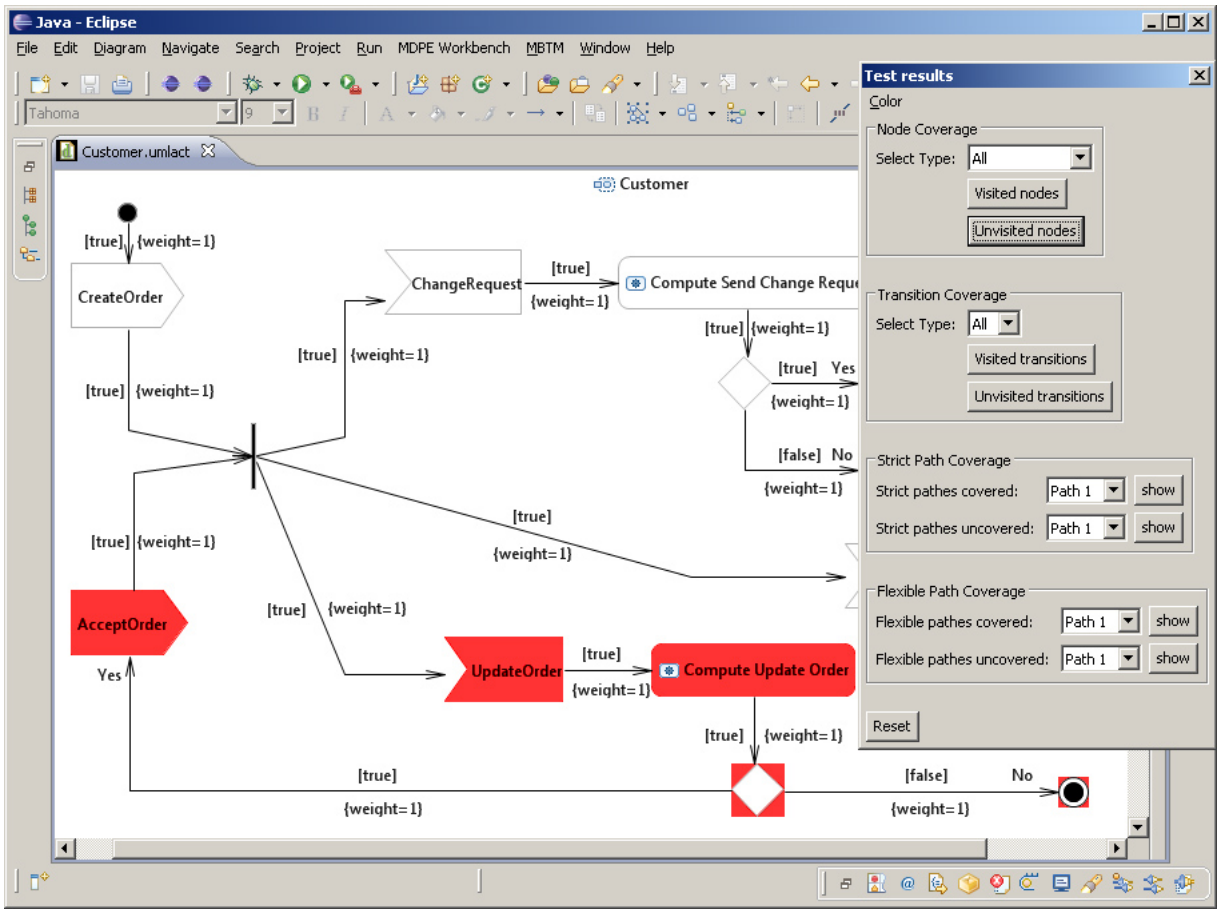


Abbildung 6.21: Plugin zur Evaluierung der Testüberdeckung (Screenshot)

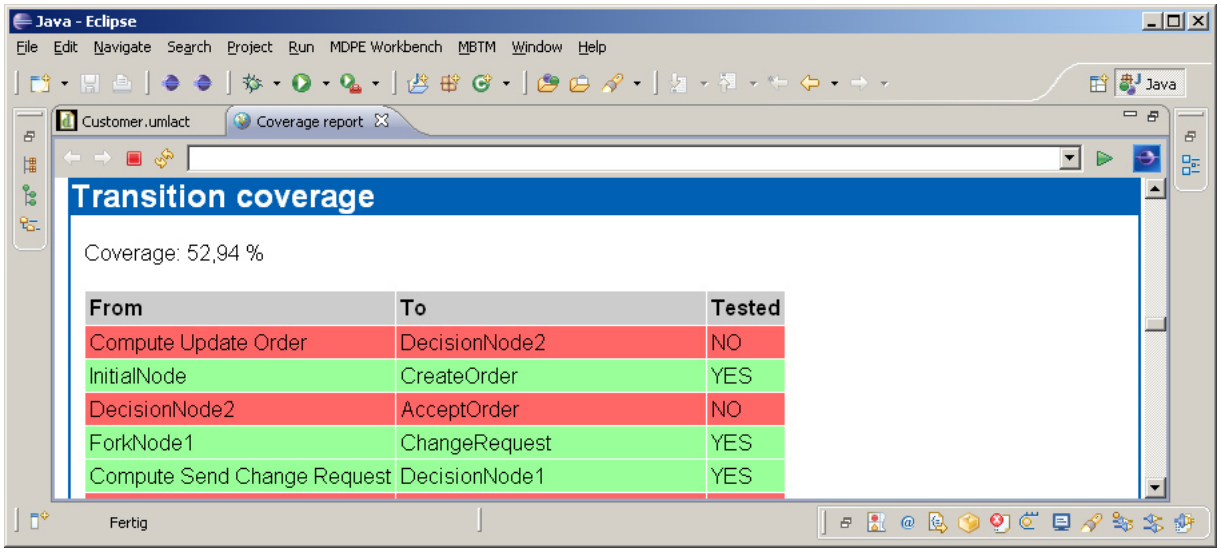


Abbildung 6.22: Generierter Report: Detailansicht der Testüberdeckung (Screenshot)

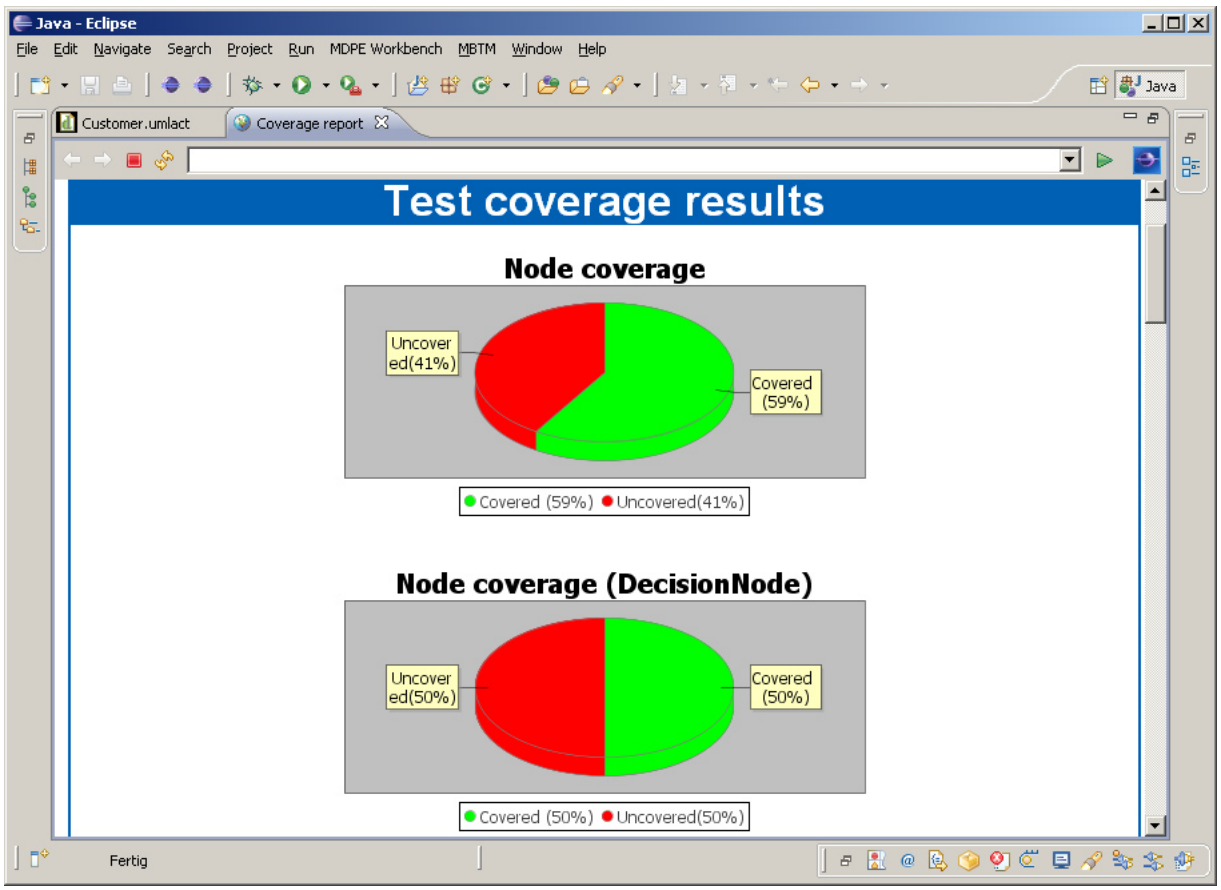


Abbildung 6.23: Generierter Report: Übersicht der Testüberdeckung (Screenshot)

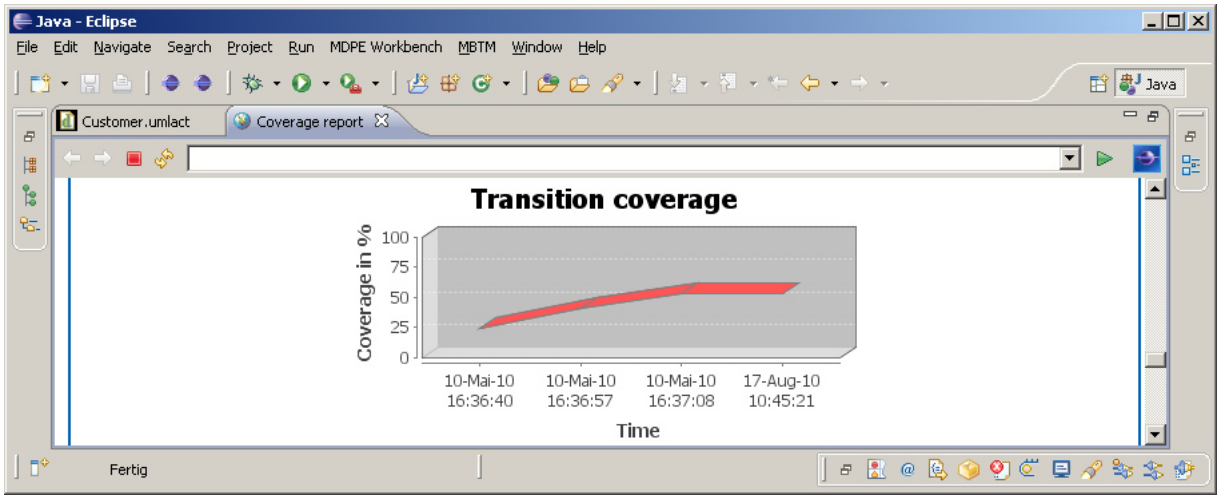


Abbildung 6.24: Generierter Report: Entwicklung der Testüberdeckung (Screenshot)

sondern auch für die agile Verwendung der UML/P, da nur geänderte Modelle und Generatoren erneut verarbeitet bzw. ausgeführt werden müssen. Insgesamt sind dabei die Konzepte dieses Kapitels als Ergänzung der in [Rum04b] dargestellten Generierung zu sehen.

Zwischen den einzelnen in diesem Kapitel angesprochenen Generaten bestehen Abhängigkeiten, die bei der Auswahl der Generatoren berücksichtigt werden müssen. So setzt die Nutzung des Generats aus Sequenzdiagrammen eine existierende Codeinstrumentierung voraus (siehe Abschnitt 6.4). Auch die Generierung aus Modellen der Testspezifikationssprache trifft Annahmen über Schnittstellen der Generate aus Objekt- und Sequenzdiagrammen (siehe Abschnitt 6.5). Mögliche und geforderte Kombinationen von Generatoren können in Form von Feature-Diagrammen [CE00] dokumentiert werden.

Bereits die in [Rum04a] eingeführte Fassung der UML/P war so konzipiert, dass ein System damit nicht nur modelliert, sondern durch die Verwendung von Codeeinbettung und Verhaltensmodellen ebenfalls vollständig implementiert und getestet werden kann. Dies wird in dieser Arbeit durch den konsequenten Bezug auf die Modellebene insbesondere des eingebetteten Codes und der Testfallergebnisse weiter fortgeführt (siehe Abschnitte 3.7, 3.9, 6.5 und 6.6). Insgesamt rückt damit der generierte Code in den Hintergrund und spielt auf Modellebene keine Rolle mehr.

Trotz dieser Mächtigkeit der UML/P ist es doch nach wie vor Aufgabe der Modelle, das Zielsystem auf einer abstrakten Ebene darzustellen. Ein wichtiger Aspekt dabei ist die Abstraktion von plattformspezifischen Implementierungsdetails des Zielsystems, die in den Generatoren gekapselt sind (siehe auch Kapitel 5). Somit bleibt die Betrachtung des aus den Modellen zu generierenden Codes weiterhin für Spezifikation der Generatoren und Ausführung des Zielsystems wichtig. Durch die Trennung zwischen abstrakter Spezifikation in den Modellen und technische Konkretisierung in den Generatoren kann die UML/P für unterschiedlichste Anforderungen, die etwa Websysteme, Stand-Alone Applikationen oder neue Gerätekategorien wie Tablet-PCs und Smartphones an eine Software stellen, eingesetzt werden. Auf diese Weise bleiben Modelle flexibel und anpassbar auch an zukünftige Anforderungen von Projekten, Plattformen und Technologien.

Vor diesem Hintergrund sind Entwurf und Implementierung bzw. Auswahl und Anpassung der Generatoren wichtige Aspekte bei der Nutzung der UML/P im Rahmen der Softwareentwicklung. Wie eine gut gewählte Architektur der Generate die Konzeption weiterer Generate erleichtern kann, zeigt das Beispiel der in diesem Kapitel diskutierten Testfallgenerierung, die auf die Infrastruktur der Codeinstrumentierung und der Systemarchitektur aufbaut. Auch die Nutzung von Frameworks wie bei der Qualitätssicherung von modellierten Testfällen in Abschnitt 6.6 kann den Implementierungsaufwand der Generatoren verringern. Trotz der vorgestellten Konzepte bleibt die Entwicklung von Generatoren als Bindeglied zwischen Modellen und Zielsystem eine herausfordernde Aufgabe, die aufgrund der bereits erwähnten projektspezifischen Anforderungen nicht allumfassend gelöst werden kann. Insgesamt entstehen damit bei Entwurf und Nutzung der UML/P unterschiedlichste Aufgabenfelder von Sprachentwurf und Modellierung bis hin zur Generator- und Werkzeugentwicklung, die zusammen mit der Architektur des UML/P-Frameworks im Kapitel 7 diskutiert werden.

Kapitel 7

Das UML/P-Framework

Im Rahmen dieser Arbeit wurde mit dem UML/P-Framework eine umfangreiche Werkzeuginfrastruktur entwickelt, die es erlaubt, die in den bisherigen Kapiteln vorgestellten Konzepte in Softwareentwicklungsprojekten einzusetzen. Das Framework basiert auf dem parallel entstandenen MontiCore-Framework, das Entwurf und Entwicklung von textuellen Sprachen und entsprechender Werkzeuge für deren Nutzung unterstützt [GKR+06, GKR+08, Kra10]. Bei der Entwicklung der vorliegenden textuellen Fassung der UML/P und des Frameworks wurden folgende Ziele verfolgt:

1. **Agilität**: Die Konzentration auf die Implementierung des Zielsystems, das während der Entwicklung jederzeit ausführbar und gleichzeitig leicht an geänderte Anforderungen anpassbar ist, sind Kernelemente agiler Vorgehensmodelle in der Softwareentwicklung [BBB+]. Sowohl die UML/P als Sprache als auch deren Werkzeuge sollen sich für die agile Entwicklung von Softwaresystemen einsetzen lassen. Dieses Ziel stand bereits im Fokus beim Entwurf der UML/P in [Rum04a] und [Rum04b]. Die dort behandelten Konzepte lassen sich auf die textuelle Notation übertragen, da bei dieser auf eine größtmögliche Konformität zur graphischen UML/P geachtet wurde (siehe Kapitel 3). Darüber hinaus erlaubt das Framework eine unabhängige Verarbeitung von Modellen und Generatoren, so dass bei Änderungen nur die betroffenen Komponenten vom Framework verarbeitet werden müssen. Dies ermöglicht eine effiziente Aktualisierung und Überprüfung des Systems als Grundvoraussetzung agiler Vorgehensweisen. Entsprechende Konzepte wurden in Kapitel 5 und 6 behandelt.
2. **Kompositionalität**: Die Kompositionalität ist eine Anforderung, die sich auf alle Aspekte der Werkzeuginfrastruktur bezieht:

   - *Sprachdefinition*: konkrete und abstrakte Syntax, Kontextbedingungen
   - *Werkzeuge*: Infrastruktur zur Sprachverarbeitung, Konsistenzprüfung, Editoren
   - *Generatoren*: Templates, Generate

   Schon in MontiCore war die Kompositionalität ein wichtiges Ziel für Sprachdefinition und -verarbeitung [KRV08b, KRV08a, KRV07a]. Die in diesem Rahmen entwickelten Konzepte

zur Komposition von konkreter und abstrakter Syntax, Lexer, Parser sowie Editoren fanden auch für die UML/P Anwendung. Danach ist die UML/P nichts anderes als die Komposition der einzelnen Sprachen und Sprachbestandteile. Dies wurde bereits bei der Aufstellung der Kontextbedingungen in Kapitel 4 berücksichtigt. Indem sich diese Strukturierung ebenfalls im zugehörigen Framework wiederfindet, ist es möglich, die UML/P in Ausschnitten oder in Kombination mit anderen Sprachen zu nutzen und dafür die entsprechenden Teile des Frameworks wieder zu verwenden.

Die Sprache UML/P erfüllt weitgehend die Anforderungen der Kompositionalität [HKR+07]. Die Modelle beschreiben verschiedene Aspekte des Systems und bestimmen in ihrer Gesamtheit dessen Struktur und Verhalten. Der Austausch oder die Hinzunahme eines Modells ist dabei unabhängig von anderen Modellen, solange davon keine Schnittstellen bzw. Referenzen betroffen sind. Wie die bisherigen Beispiele der TripleLogo-Applikation zeigen, genügt demnach ein einzelnes Statechart oder eine OCL-Datei, um zusätzliches Verhalten oder Bedingungen des Systems zu spezifizieren (siehe Kapitel 3). Entsprechend wurden in den Kapiteln 5 und 6 Konzepte vorgestellt, um diese Kompositionalität der Modelle auf die der Generatoren und Generate zu übertragen.

3. **Anpassbarkeit**: Die UML/P ist bei der Modellierung von Softwaresystemen nicht auf bestimmte Domänen oder Plattformen festgelegt. Für eine stärkere Spezialisierung kann eine Anpassung über die in Abschnitt 6.2 diskutierte Profilbildung erfolgen. Das Framework bietet dafür eine Unterstützung für die Implementierung von entsprechenden Kontextbedingungen und Generatoren. Auf diese Weise können auf die Erfordernisse von Projekten, Unternehmen, Domänen oder Technologien angepasste Varianten der UML/P einschließlich automatisierter Konsistenzsicherung und Codegenerierung entstehen.
4. **Flexible Verwendbarkeit**: Unternehmen bis hin zu einzelnen Projekten besitzen im Allgemeinen eine ganz eigene Werkzeuglandschaft. Das UML/P-Framework lässt sich deshalb flexibel in solch eine bestehende Infrastruktur integrieren. Die Verwendung von textuellen Sprachen und Generatoren auf Templatebasis bietet hier bereits den Vorteil, dass sie sich ohne weiteres in Versionsmanagementsysteme wie CVS oder SVN einbinden und innerhalb beliebiger Entwicklungsumgebungen oder Editoren bearbeiten lassen (siehe auch Abschnitt 2.3 und [GKR+07]). Weitere Anwendungsszenarien sind die Integration als Plugin in Entwicklungsumgebungen wie Eclipse oder die Verwendung innerhalb von Build-Systemen wie Ant oder Make [Hol05, Mec04], die oft auch in Form von sogenannten "Nightly-builds" die tägliche automatisierte Qualitätssicherung des vollständigen Systems übernehmen.

Abbildung 7.1 gibt einen abstrakten Überblick der Gesamtstruktur des Frameworks und der beteiligten Komponenten. Gleichzeitig werden dabei die unterschiedlichen Rollen aufgeführt, die sich aus Entwicklung, Anpassung und Nutzung der UML/P und des Frameworks ergeben:

- **Modellierer** (engl. "Modeler"): Der Modellierer setzt die Sprachen der UML/P zur Spezifikation und Entwicklung von Softwaresystemen ein. Die für Verarbeitung, Ausführung

und Testen der Modelle notwendigen Aufrufe des Frameworks bzw. des Generats können soweit gekapselt und automatisiert werden, dass diese im Hintergrund stattfinden. Für den Modellierer erfolgt dadurch Entwurf, Implementierung, Qualitätssicherung und Ausführung des Softwaresystems allein auf Ebene der Modelle. Demnach sind für diese Anwendung der UML/P nur Kenntnisse über Syntax und Semantik der Sprachen, nicht aber über Aufbau und Arbeitsweise des Frameworks oder des Generats erforderlich.

- **Programmierer** (engl. "Programmer"): Gemeinsam mit der UML/P kann für die Umsetzung eines Softwaresystems weiterhin auch herkömmliche Programmierung zum Einsatz kommen. Dabei handelt es sich einerseits um Code wie Java/P, der sich in Form von Einbettung oder separaten Dateien auf die Modelle bezieht und dadurch ebenfalls vom UML/P-Framework verarbeitet werden muss. In diesem Fall sind wie beim Modellierer entsprechende Kenntnisse der UML/P-Sprachen für die Implementierung erforderlich. Gleichzeitig können aber auch manuell implementierte Frameworks oder Bibliotheken verwendet werden, die als Laufzeitumgebung[1] das generierte System ergänzen (vgl. Kapitel 6). Diese Laufzeitumgebung ist unabhängig von der UML/P implementier- und einsetzbar.
- **Sprachentwickler** (engl. "Language developer"): Der Sprachentwickler entwirft die Grammatiken, auf denen konkrete und abstrakte Syntax der UML/P basieren, sowie die Semantik der Sprache. Während die Grammatiken bzw. die daraus erzeugte Infrastruktur zur Sprachverarbeitung fester Bestandteil des UML/P-Frameworks ist, kann die Semantik in Form von Profilen an die Erfordernisse von Domänen, Unternehmen oder Projekten angepasst werden.
- **Werkzeugentwickler** (engl. "Tool developer"): Die Aufgabe des Werkzeugentwicklers ist die Erstellung der grundlegenden Infrastruktur. Im vorliegenden Fall ist dies die Entwicklung der Basis-Frameworks von MontiCore und der UML/P.
- **Werkzeugcustomizer** (engl. "Tool customizer"): Im Gegensatz zum Werkzeugentwickler entwirft der Werkzeugcustomizer projektspezifische Infrastruktur in Form von angepassten Profilen und Generatoren, wobei auch neue Werkzeuge entstehen können. Eine solche spezifische Infrastruktur muss nicht notwendigerweise für jedes Projekt neu entwickelt werden. So werden mit dem UML/P-Framework bereits die in Kapitel 4 und 6 diskutierten Kontextbedingungen und Generatoren mit ausgeliefert. Darüber hinaus sind in Zukunft Bibliotheken von vorgefertigten Profilen und Generatoren für spezielle Domänen, Plattformen oder Technologien denkbar, aus denen entsprechend den Erfordernissen eines Projekts passende Infrastruktur ausgewählt werden kann. Dennoch handelt es sich hier um einen Erweiterungspunkt für Nutzer des Frameworks, der die UML/P auch an zukünftige Technologien und Anforderungen anpassbar macht.

Trotz der getrennten Aufführung kann ein Entwickler im Laufe eines Projekts mehrere Rollen im Wechsel oder parallel einnehmen. Dabei ist nicht notwendigerweise ein bestimmtes

[1] engl. "Runtime Environment"

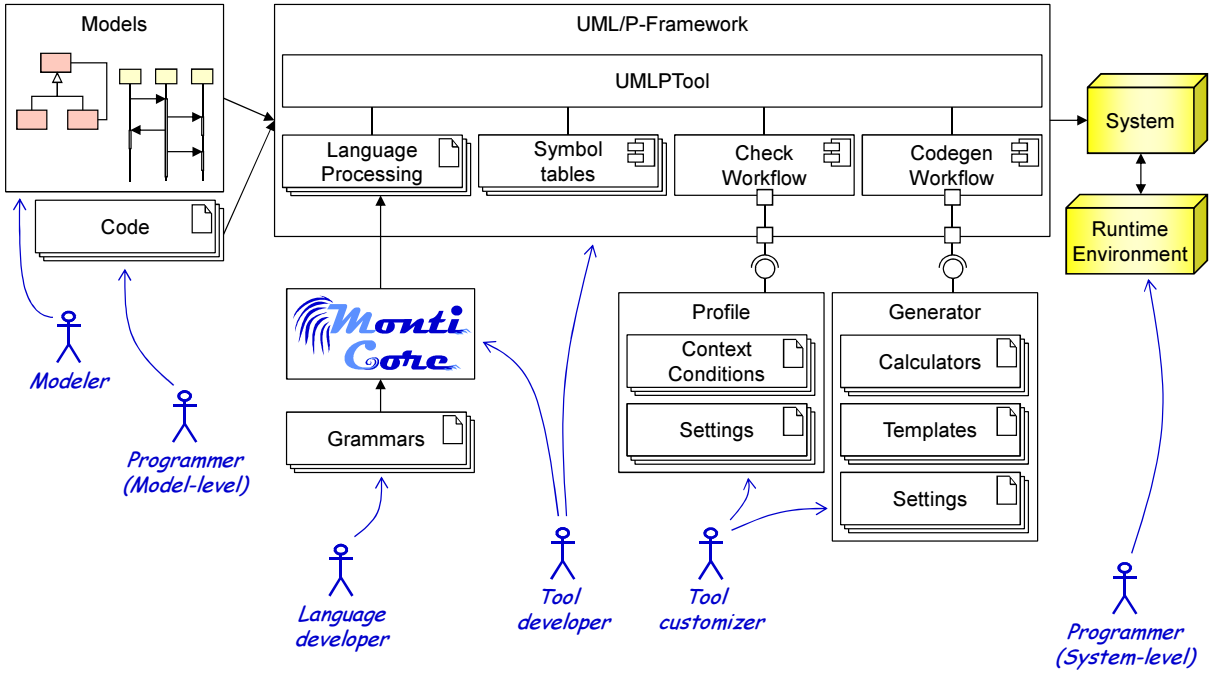


Abbildung 7.1: Bestandteile und Rollen im UML/P-Framework

Vorgehensmodell einzuhalten. Selbst ein agiles, Code-zentriertes Vorgehen wie bei Extreme Programming (XP, [BA04]) ist möglich. Bei Anwendung der UML/P setzen sich in diesem Fall die Quellen des Softwareprodukts aus den Artefakten der Modellebene, den Generatoren und dem Laufzeitsystem zusammen.

Allgemein bietet es sich aufgrund der Abstraktion an, frühe Versionen des Zielsystems im Wesentlichen mit Modellen zu entwerfen. Im Projektverlauf kann das System sukzessiv durch weitere Modelle und Code verfeinert, sowie durch die Entwicklung entsprechender Generatoren oder Laufzeitumgebungen mit plattformspezifischen Aspekten ergänzt werden. Gleichzeitig bietet die Profilbildung die Möglichkeit, eine effizientere Modellierung und gezieltere Codegenerierung durch die Definition von Stereotypen und Kontextbedingungen zu unterstützen.

Im Folgenden wird der Aufbau des UML/P-Frameworks und dessen Schnittstellen beschrieben. Einen Überblick der Projektstruktur gibt Abschnitt 7.1. Neben der bereits erwähnten Profilbildung bietet diese Struktur weitere Flexibilität beim Einsatz der UML/P, indem Teile des Frameworks ausgetauscht und mit neuen Sprachen kombiniert werden können. Der folgende Abschnitt 7.2 befasst sich mit der Architektur der Symboltabellen. Diese Infrastruktur stellt Informationen über Modelle und Modellelemente zur Verfügung, die aus der abstrakten Syntax nur indirekt zu ermitteln sind. Hilfreich ist dies insbesondere für die Implementierung von Kontextbedingungen. Für diese und die Implementierung von Codegeneratoren bietet das UML/P-Framework Schnittstellen an, die die Abschnitte 7.3 und 7.4 beschreiben. Darüber hinaus kann die Codegenerierung optional durch das in Abschnitt 7.5 behandelte Refactoringframework

ergänzt werden. Abschließend diskutiert Abschnitt 7.6 unterschiedliche Nutzungsmöglichkeiten des Frameworks.

7.1 Projektstruktur

Das UML/P-Framework ist in eine Reihe von Komponenten aufgeteilt, um eine Wiederverwendbarkeit der einzelnen Sprachen auch in unterschiedlichen Konstellationen und Kontexten zu ermöglichen. Diese stehen als Java Archive[2] zur Verfügung, die je nach Einsatzzweck zusammengestellt werden können.

Die Aufteilung beginnt bei der Sprachdefinition, indem die von MontiCore zur Verfügung gestellten Konzepte der Grammatikvererbung und -einbettung genutzt werden [KRV08a, KRV08b]. Wie Abbildung 7.2 zeigt, sind die sieben in Kapitel 3 beschriebenen Sprachen der UML/P jeweils als eigene Grammatiken umgesetzt (vgl. Anhang C). Diese teilen sich mit `Literals` und `Types` eine gemeinsame Syntax für Literale und Typen. Literale definieren Werte für Basistypen wie Zeichenketten (Strings) und Zahlen, während die Basistypen selbst zusammen mit dem Aufbau von Bezeichnern für komplexe Typen in der Grammatik `Types` enthalten sind. Aufgrund der Java-orientierten Syntax der UML/P sind diese Elemente konform zu Java 6.0, so dass sie ebenfalls für die Java-Grammatik wiederverwendet werden konnten. Diese definiert die Syntax von Java/P, die in der in dieser Arbeit verwendeten Fassung mit der von Java 6.0 identisch ist. Die speziellen Eigenschaften von Java/P, Elemente der Modellebene referenzieren und verwenden zu können, wirkt sich auf die konkrete Syntax nicht aus und wird allein auf Basis der Symboltabelle realisiert (siehe Abschnitt 7.2). Weitere gemeinsame Elemente wie Stereotypen oder Modifikatoren, die in mehreren der UML/P-Sprachen verwendet werden, wurden darüber hinaus in eine eigene Grammatik `Common` ausgelagert. Auf diese Weise wird mit Hilfe der Sprachvererbung bereits auf Grammatikebene Redundanz vermieden und die Wiederverwendung maximiert.

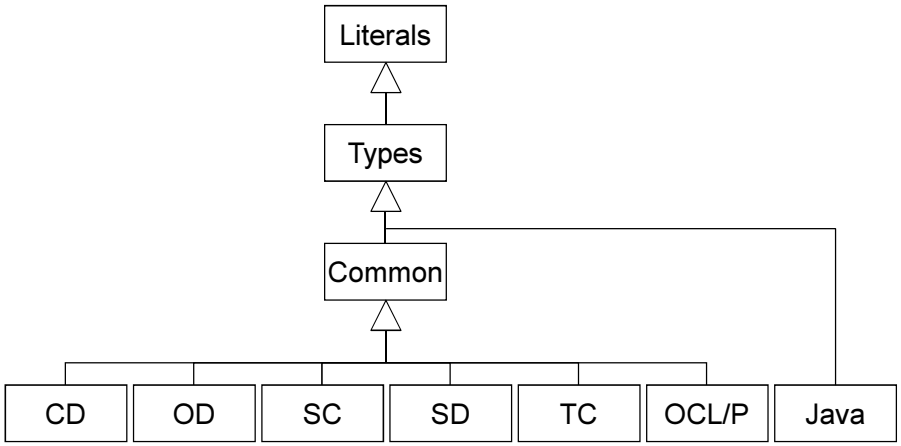


Abbildung 7.2: Sprachvererbung der Grammatiken innerhalb der UML/P

[2] nach der Dateiendung werden diese häufig auch als JAR-Dateien bezeichnet

Eine weitere Form der Wiederverwendung auf Grammatikebene findet durch Einbettung statt. So können Java- und OCL/P-Ausdrücke in Klassen-, Objekt- und Sequenzdiagrammen, sowie in Statecharts und der Testspezifikationssprache verwendet werden. Diese Einbettung wurde in den Grammatiken nicht fest vorgegeben, sondern separat spezifiziert und ist damit austauschbar (vgl. Abschnitt 3.9). Dies und die Aufteilung der Sprachen ermöglicht die Entstehung neuer Formen der UML/P, die nur eine Submenge der Sprachen verwendet oder diese mit anderen Sprachen kombiniert. Hinzu kommen verschiedene Einsatzszenarien des Frameworks, die sich je nach der in einem Projekt oder Unternehmen verwendeten Infrastruktur bzw. Werkzeuge unterscheiden. So können etwa Plugins für Entwicklungsumgebungen wie Eclipse zur Verfügung gestellt werden, die eine integrierte Nutzung des Frameworks erlauben. Allerdings ist so eine Integration im Allgemeinen werkzeugspezifisch und führt darüber hinaus zu Abhängigkeiten des Frameworks zu dem jeweiligen Werkzeug. Deshalb ist es sinnvoll, die Kernfunktionalität des Frameworks und die Werkzeugintegration als separate Komponenten umzusetzen. Um diese zu unterscheiden, wird in dieser Arbeit folgende Namenskonvention verwendet, wobei die in Klammern stehenden Abkürzungen jeweils an den Namen der Komponente angehängt werden:

- **Front-End (FE)**: Kernfunktionalität des Frameworks. Bei Sprachkomponenten handelt es sich dabei insbesondere um Infrastruktur zur Verarbeitung und Analyse der Modelle sowie Definition und Ausführung von Generatoren.
- **Plugin (Plugin)**: Integration der Funktionalität des Front-Ends in ein Werkzeug, sofern dafür spezifischer Code notwendig ist. Im Folgenden stehen diese Komponenten für Editoren innerhalb der Entwicklungsumgebung Eclipse. Zukünftig sind jedoch auch Plugins für andere Werkzeuge denkbar.
- **Runtime Environment (RE)**: Laufzeitumgebung, die für die Ausführung des aus den Modellen generierten Codes erforderlich ist (vgl. Kapitel 6). Diese wird durch den jeweils verwendeten Generator bestimmt und muss gemeinsam mit diesem zur Verfügung stehen. Im Gegensatz zum Generator ist die Laufzeitumgebung notwendiger Bestandteil des Zielsystems, so dass es sinnvoll ist, diese als separate Komponente umzusetzen.
- **Online Service (OSTP)**: Komponente für die Verarbeitung der Modelle durch den Online Service. Bei dieser Komponente handelt es sich um die Integration des UML/P-Frameworks in die "Online Software Transformation Platform" (OSTP), die es ermöglicht, den Generierungsprozess auf einen zentralen Server auszulagern [Her06].
- **Back-End (BE)**: Funktionalität, die speziell für die Verarbeitung der Modelle auf dem OSTP-Server zur Verfügung stehen soll. Auf diese Weise ist es möglich, Generatoren zur Nutzung anzubieten, deren Umsetzung jedoch nicht offen zu legen.

Neben dieser Unterscheidung nach Einsatzszenarien wurde das UML/P-Framework entsprechend den Grammatiken zusätzlich nach Sprachen und Einbettung unterteilt. Die entsprechenden Komponenten sind in Tabelle 7.3 aufgeführt, wobei die Anteile der `Common`-Grammatik in der Komponente `umlpCoreFE` abgelegt sind.

Kategorie	*Beschreibung*	*Komponenten*
Sprache	Infrastruktur, die die Verarbeitung der Sprache selbst betrifft, ohne Einbettung oder Referenzen zu Modellen anderer Sprachen zu behandeln. Dazu zählen konkrete und abstrakte Syntax (Grammatik), Lexer, Parser, Pretty-Printer, Infrastruktur zur Auflösung von sprachinternen Elementen (siehe Abschnitt 7.2), Kontextbedingungen und Generatoren.	`umlpCoreFE`, `cdFE`, `odFE`, `scFE`, `sdFE`, `tcFE`, `oclFE`, `javaFE`
Einbettung	Festlegung, welche anderen Sprachen in die Hauptsprache eingebettet werden sollen. In der vorliegenden Arbeit ist dies Java und OCL/P, wobei letztere als obligatorisch angesehen und deshalb im Namen der Komponente nicht extra aufgeführt wird. Dennoch ist auch die OCL/P austauschbar. In diesen Komponenten werden somit Lexer und Parser der einzelnen Sprachen kombiniert sowie entsprechende Infrastruktur zur Verarbeitung und Überprüfung vollständiger Modelle bereitgestellt.	`cdJavaFE`, `odJavaFE`, `scJavaFE`, `sdJavaFE`, `tcJavaFE`
Plugin (Sprache)	Plugin für Eclipse, das für die jeweilige Sprachkomponente einen Editor mit Komfortfunktionen wie Autovervollständigung, Syntaxhervorhebung, Modellübersicht (Outline) oder dem Ein- und Ausblenden von Details (Folding) zur Verfügung stellt.	`umlpCorePlugin`, `cdPlugin`, `odPlugin`, `scPlugin`, `sdPlugin`, `tcPlugin`, `oclPlugin`, `javaPlugin`
Plugin (Einbettung)	Editor für Eclipse, der aufbauend auf dem Plugin der Hauptsprache zusätzlich die Spracheinbettung berücksichtigt. Dieser entspricht also der Zusammensetzung der jeweiligen Einbettungskomponente.	`cdJavaPlugin`, `odJavaPlugin`, `scJavaPlugin`, `sdJavaPlugin`, `tcJavaPlugin`

Tabelle 7.3: Übersicht der Sprachkomponenten der UML/P

In den Komponenten der einzelnen Sprachen werden Eigenschaften und Infrastruktur der Sprache selbst realisiert. Beziehungen untereinander sind abgesehen von der Einbettung an dieser Stelle noch nicht festgelegt. Diese werden erst in der Komponente `umlpFE` umgesetzt, die die einzelnen Sprachen zu einer Sprachfamilie zusammenfasst. Somit sind auch erst hier die sprachübergreifenden Inter-Modell-Bedingungen überprüfbar. Auf diese Weise ist es möglich, jede Sprache der UML/P auch separat zu verwenden und Beziehungen zu anderen Sprachen neu zu definieren (siehe auch Abschnitt 7.2).

Die für die Entwicklungsumgebung Eclipse realisierten Editoren sind in den Plugin-Komponenten gekapselt. Diese sind für die Nutzung der UML/P nicht erforderlich, bieten jedoch mehr Komfort bei der Modellierung als gewöhnliche Texteditoren. Darüber hinaus empfiehlt es sich, Laufzeitumgebungen der Generate ebenfalls als eigene Komponenten zu verwalten. Da diese jedoch von der jeweiligen Generatorumsetzung abhängig und im Allgemeinen erst zur Kompilierung bzw. Ausführung des generierten Systems notwendig sind, werden diese im Folgenden nicht weiter berücksichtigt. Die in Kapitel 6 diskutierten Ansätze sind zurzeit zwar als Generatoren im UML/P-Framework integriert, wobei die Laufzeitumgebung als Komponente `cdRE` zur Verfügung steht. Zukünftig sind jedoch eher Bibliotheken von Generatoren und entsprechender Laufzeitumgebungen sinnvoll, aus denen je nach Einsatzzweck und Zielplattform eines Projektes ausgewählt werden kann.

Die Abhängigkeiten zwischen den einzelnen Komponenten des UML/P-Frameworks sind in Abbildung 7.4, 7.5 und 7.6 dargestellt, wobei die Plugin- und Service-Komponenten nur bei entsprechender Nutzung erforderlich sind. Zusätzlich zur Tabelle 7.3 sind dabei folgende weitere Komponenten des UML/P-, MontiCore- und OSTP-Frameworks aufgeführt:

1. **UML/P-Framework**:
   - `umlpFE`: Front-End für die Nutzung der UML/P als Sprachfamilie.
   - `umlpOSTP`: Integration der UML/P als OSTP-Service.
   - `umlpBE`: Über den OSTP-Service bereitgestellte Generatoren und Infrastruktur.
2. **MontiCore-Framework**:
   - `ETS`[3]: Framework zur Implementierung von Symboltabellen.
   - `MontiCoreFreeMarker`: Integration der FreeMarker-Templateengine in MontiCore.
   - `MontiCoreGLI`[4]: Infrastruktur für Generator- und Sprachentwicklung.
   - `MontiCoreGLIPlugin`: Integration von `MontiCoreGLI` in Eclipse.
   - `MonitCoreTE_RE`[5]: Für die Plugins verwendete Schnittstelle zu Eclipse.
   - `MontiCoreFE`: Front-End für die Sprachentwicklung mit MontiCore.
   - `MontiCore`: Generator für Lexer, Parser und abstrakter Syntax aus Grammatiken.
   - `MontiCorePlugin`: Integration von MontiCore in Eclipse.
   - `MontiCoreRE`: Laufzeitumgebung für die aus Grammatiken generierte Infrastruktur.
   - `MontiCoreOSTP`: Serverseitige Integration von MontiCore und OSTP.
3. **OSTP-Framework**:
   - `ostp.client.common`: Clientseitige-Schnittstelle zum OSTP-Service.

[3]ETS: Extensible Type System

[4]GLI: Generation and Language Infrastructure

[5]TE_RE: Text Editor Runtime Environment

- `ostp.client.eclipse`: Integration des OSTP-Clients in Eclipse.
- `ostp.pluginapi`: Infrastruktur für die serverseitige Bereitstellung von Services.

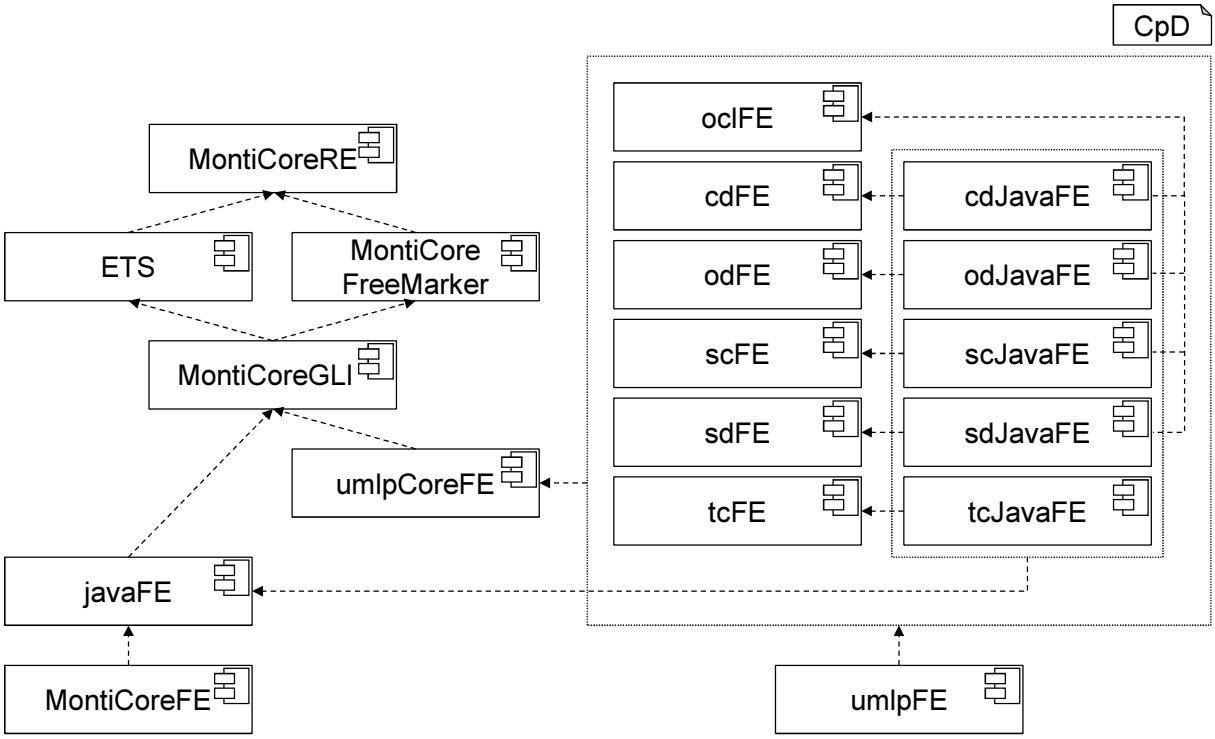


Abbildung 7.4: Kernkomponenten und Abhängigkeiten des UML/P-Frameworks

Seit der Veröffentlichung von MontiCore 1.0 in [GKR+06] wurde das Framework kontinuierlich weiterentwickelt. Ursprünglich im Rahmen dieser Arbeit für die UML/P entworfen, ist das Generierungs- und Refactoringframework mittlerweile als eigene Komponente `MontiCoreGLI` in MontiCore integriert. Dadurch steht der in Abschnitt 5.2 diskutierte Generatoraufbau allen mit MontiCore entwickelten Sprachen zur Verfügung. Gleichzeitig wurde der Generator, der in MontiCore die für die Sprachverarbeitung notwendige Infrastruktur aus den Grammatiken erstellt, auf diesen Ansatz umgestellt.

Neben dem Generierungs- und Refactoringframework, die im Detail in den Abschnitten 7.4 und 7.5 beschrieben werden, enthält `MontiCoreGLI` außerdem die Grammatiken `Literals` und `Types` sowie entsprechende Infrastruktur für deren Nutzung. Beide Grammatiken stellen selbst keine eigenständigen Sprachen dar, können aber als Grundlage für die Entwicklung neuer Sprachen dienen. Insgesamt bietet `MontiCoreGLI` damit eine allgemeine Infrastruktur für Sprachdefinition und Verarbeitung.

Die Nutzung der FreeMarker-Templateengine wurde ebenfalls als eigenständige Komponente realisiert, um die Abhängigkeiten zu FreeMarker zu kapseln und die Templateengine ggf. leicht austauschen zu können.

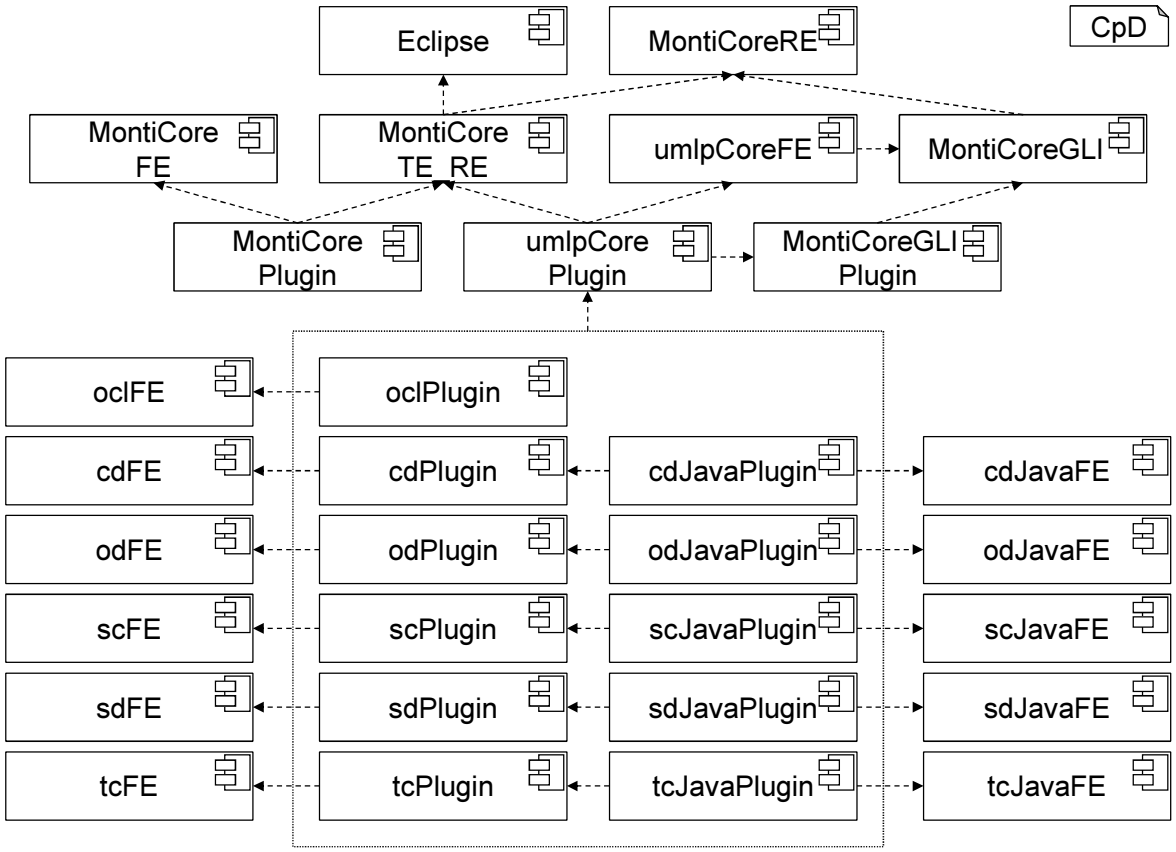


Abbildung 7.5: Pluginkomponenten und Abhängigkeiten des UML/P-Frameworks (zusätzlich gelten die Abhängigkeiten aus Abbildung 7.4)

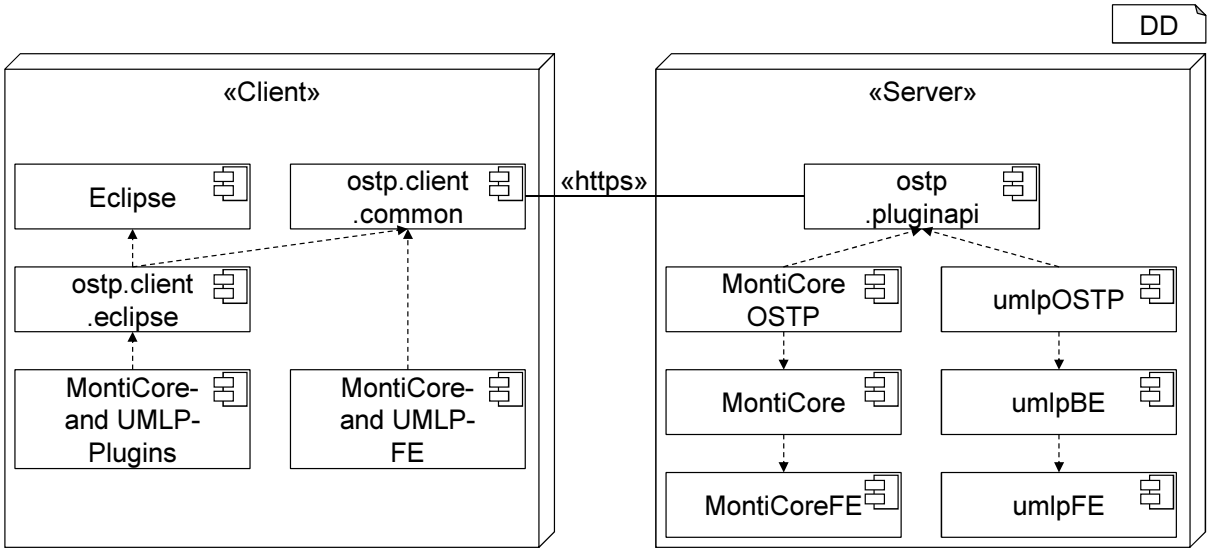


Abbildung 7.6: Komponenten bei Nutzung des UML/P-Frameworks als Service

Die übrigen Komponenten des MontiCore-Frameworks wurden im Vorfeld oder parallel zum UML/P-Framework entwickelt [GKRS06, GKR+06, GKR+08]. Die `ETS`-Komponente ist im Rahmen von [Völ11] entstanden und wird in Bezug auf die Anwendung in der UML/P in Abschnitt 7.2 behandelt. Alle weiteren Komponenten von MontiCore fasst [Kra10] zusammen. Die Komponenten des OSTP-Frameworks basieren auf [Her06].

Abbildung 7.7 zeigt, wie die einzelnen Komponenten auf Architekturebene des UML/P-Frameworks integriert werden. Diese Integration der Infrastruktur erfolgt dabei auf drei Ebenen:

1. **Sprachverarbeitung**: Das `DSLTool` der `MontiCoreRE`-Komponente ist die Hauptschnittstelle von MontiCore für die Verarbeitung von Sprachinstanzen [Kra10]. Aus Sicht der UML/P sind dies die Modelle, die über das `DSLTool` eingelesen und in die abstrakte Syntax überführt werden können. Diese Prozesse sind als sogenannte *Workflows* gekapselt, über die die Sprachverarbeitung gesteuert und erweitert werden kann. Die Verwaltung von eingelesenen Modellen erfolgt in Form von `DSLRoot`-Objekten, die jeweils die Instanz der abstrakten Syntax (AST) sowie weitere Informationen für die Verarbeitung eines Modells enthalten.
2. **Sprachanalyse**: Aufbauend auf dem `DSLTool` ergänzt das `ETSTool` der `ETS`-Komponente die Sprachverarbeitung von MontiCore um Infrastruktur zur Sprachanalyse einschließlich deren sprachübergreifender Integration im Hinblick auf eingebettete Sprachen und Sprachfamilien [Völ11]. Die Infrastruktur beinhaltet im Wesentlichen Aufbau und Auflösungsmechanismen der Symboltabellen sowie Konfiguration und Überprüfung der Kontextbedingungen, die für jede Sprache in einer Subklasse von `ILanguage` gekapselt ist (siehe Abschnitte 7.2 und 7.3). Dieser fügt eine `ModelingLanguage` weitere Informationen wie Dateiendungen hinzu, die für die Verwaltung einer Sprache als eigenständige Einheiten nötig sind. Mehrere solcher Sprachen wie im Falle der UML/P können schließlich als Sprachfamilie innerhalb von `LanguageFamily` zusammengefasst werden.
3. **Codegenerierung**: Das `GenerationTool` aus `MontiCoreGLI` kapselt schließlich die Infrastruktur für die Codegenerierung (siehe Abschnitte 7.4 und 7.5). Diese beinhaltet im Wesentlichen die in der Klasse `GenerationLanguage` realisierte Verwaltung der auf den einzelnen Modellen auszuführenden Generatoren und Refactorings. Die Generatoren werden als Parameter beim Aufruf des Tools übergeben, in der Klasse `GeneratorSetup` aufbereitet und schließlich über den generischen `CodegenWorkflow` im Rahmen der Modellverarbeitung ausgeführt. Ebenfalls über Parameter wird die Ausführung der Refactorings bestimmt, deren Umsetzung auf dem Visitor-Muster basiert (siehe Abschnitt 5.2).

Durch diese Architektur beinhaltet jede Sprache der UML/P die für Verarbeitung, Analyse und Generierung notwendige Infrastruktur, die im `UMLPTool` der `umlpFE`-Komponente vollständig integriert wird. Die Integration der Sprachen sowie deren Eigenschaften findet dabei wiederum auf zwei Ebenen statt, die sich aus Strukturierung und Komposition der Grammatiken ergeben:

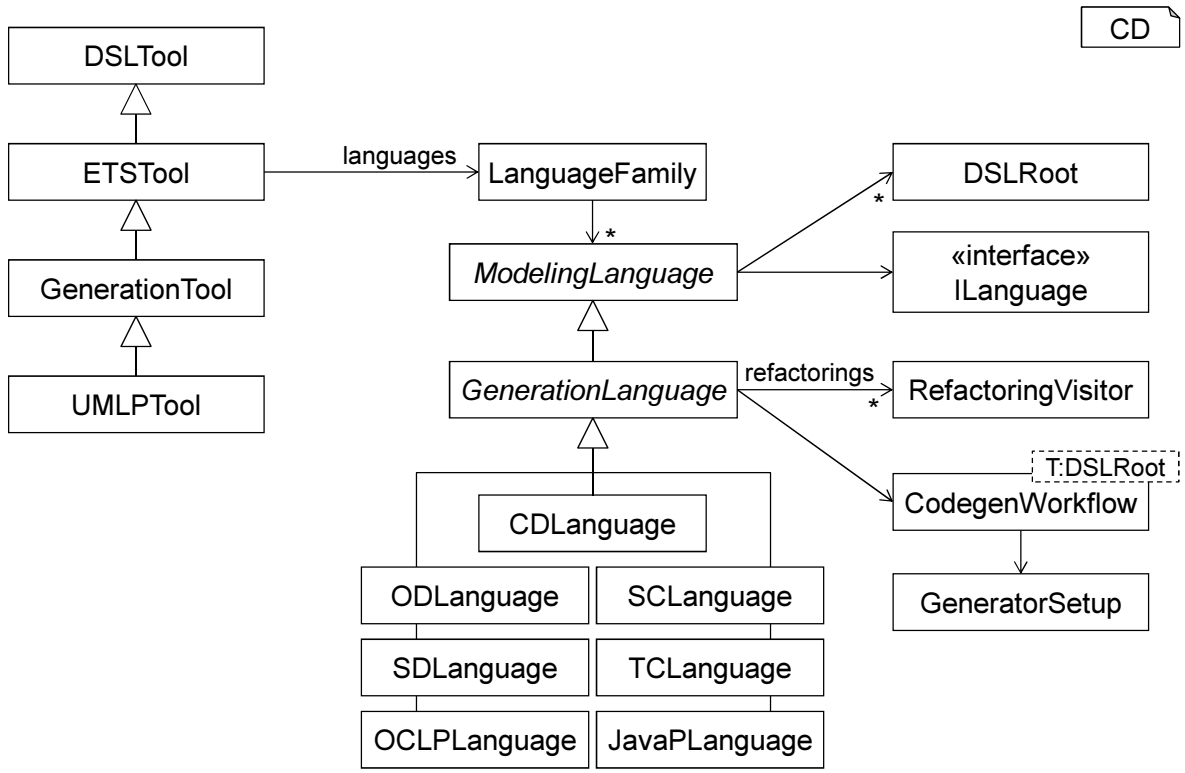


Abbildung 7.7: Architektur des UML/P-Frameworks

1. **Komposition durch Einbettung**: Sprachen oder Sprachfragmente können in andere Sprachen integriert werden. Auf Ebene der Grammatiken und der Sprachverarbeitung erfolgt dies in MontiCore durch das Konzept der Spracheinbettung. So können in Klassendiagrammen Ausdrücke genutzt werden, die auf Regeln der Java- und OCL/P-Grammatiken basieren. Darüber hinaus bringt jede Sprache eigene Kontextbedingungen und Symboltabellen für die Sprachanalyse mit sich. Diese werden in den konkreten Implementierungen von `GenerationLanguage` in Abbildung 7.7 integriert und gegebenenfalls um zusätzliche Analysen und Auflösungsmechanismen ergänzt, die etwa dazu führen, dass die Java-Syntax als Java/P interpretiert wird (siehe Abschnitt 7.2).
2. **Komposition durch gemeinsame Verwendung**: Neben der Einbettung werden die einzelnen Sprachen der UML/P zu einer Sprachfamilie zusammengefasst. Diese Integration findet nicht auf Ebene der abstrakten Syntax statt, sondern erfolgt über Referenzen zwischen Modellen sowie deren entsprechende Interpretation. Umgesetzt ist dies durch die Vereinigung der einzelnen Instanzen von `GenerationLanguage` in einer `LanguageFamily`, die über das `UMLPTool` genutzt werden kann. Zusätzlich für die gemeinsame Interpretation der Modelle notwendiger Code wird hier ebenfalls hinzugefügt (siehe Abschnitt 7.2).

Die Komposition der Sprachen erfolgt demnach auf Basis der integrierten Infrastruktur. Diese ermöglicht ebenfalls eine gemeinsame Nutzung der Generatoren jeder Sprache. Entsprechende

Konzepte für deren Integration auf Template- oder Generatebene wurden in Abschnitt 5.5 und Kapitel 6 behandelt.

Abbildung 7.8 stellt die beiden Ebenen der Sprachkomposition für die UML/P dar, wie sie im `UMLPTool` umgesetzt ist. Diese kann jederzeit durch die Schnittstellen des Tools ergänzt oder modifiziert werden. So sind Anpassungen der Symboltabellen und Kontextbedingungen, die Registrierung von Refactorings bis hin zum Hinzufügen weiterer Sprachen möglich. Gleichzeitig erlaubt es die vorgestellte Aufteilung des Frameworks, Komponenten flexibel zusammen zu stellen, einzelne Sprachen zu verwenden oder diese neu zu kombinieren. So könnte etwa eine Variante der Klassendiagramme entstehen, die die Einbettung von C++ erlaubt. In diesem Fall kann die `cdFE`-Komponente wieder verwendet werden, so dass nur die Implementierung einer entsprechenden `cdCppFE`-Komponente notwendig ist.

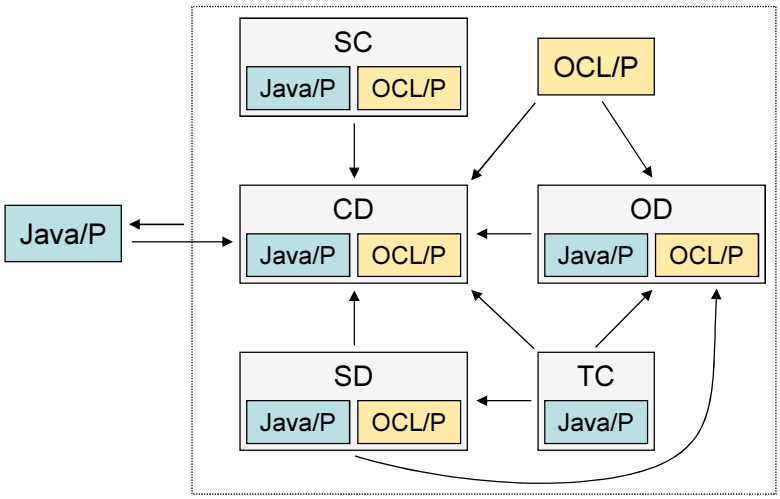


Abbildung 7.8: Sprachkomposition durch Einbettung und Referenzierung in der UML/P

Andere Szenarien sind bereits mit der jetzigen Infrastruktur des UML/P-Frameworks umsetzbar. So kann die Testspezifikationssprache gemeinsam mit Sequenz- und Objektdiagrammen auch für Tests von Java/P-Klassen eingesetzt werden. Dazu müssen nur entsprechende Codegeneratoren hinzugefügt werden, die die Instrumentierung von Java/P-Klassen ermöglichen (vgl. Abschnitt 6.4).

Insgesamt erlaubt die hier vorgestellte Infrastruktur sowie deren Aufteilung nach Sprachen, Einbettung und Einsatzzweck, die UML/P gezielt an Anforderungen von Domänen oder Unternehmen anzupassen (Customizing). Dazu kann aus einem Portfolio an Komponenten ausgewählt oder das Framework gezielt um weitere Komponenten erweitert werden. Auf diese Weise wird eine Flexibilität erreicht, die von der Integration des UML/P-Frameworks in unterschiedlichste Werkzeuge (siehe auch Abschnitt 7.6) bis hin zur Entstehung neuer Formen der UML/P reicht.

7.2 Symboltabellen

Die Symboltabelle ist eine Datenstruktur, die parallel zur abstrakten Syntax aufgebaut wird. Sie enthält Zusatzinformationen für das Identifizieren (Auflösen) von Bezeichnern (Namen) in der abstrakten Syntax. So kann etwa für Objekte und Attribute in Objektdiagrammen jeweils ein Typ angegeben werden. Innerhalb der konkreten und abstrakten Syntax ist dies nur ein Name in Form eines Strings (vgl. Quellcode C.6 im Anhang C.5). Innerhalb der Sprache hat dieser Name jedoch die Bedeutung einer Referenz, die in diesem Fall z.B. auf eine Klasse eines Klassendiagramms oder auf eine Java-Klasse verweisen kann. Das referenzierte Element sowie dessen Eigenschaften zu ermitteln, ist Aufgabe der Symboltabelle. Der Vorgang wird auch als Namensauflösung bezeichnet.

Die Symboltabelle bildet eine wichtige Grundlage für die Überprüfung von Kontextbedingungen (siehe auch Abschnitt 7.3). Zum Beispiel erweitert die Klasse `Turtle` im Klassendiagramm der TripleLogo-Applikation die Klasse `Animal` (siehe Quellcode 3.4). Damit diese Referenz im Kontext einer Vererbungsbeziehung gültig ist, muss es sich bei `Animal` um eine nicht finale Klasse handeln (vgl. CD-1-04 in Kapitel 4), was hier leicht anhand der abstrakten Syntax bei der Verarbeitung des Klassendiagramms überprüfbar wäre. In anderen Fällen könnte es sich jedoch bei `Animal` um eine importierte Klasse handeln, deren Überprüfung das Einlesen weiterer Modelle anhand der Import-Anweisungen notwendig machen würde. Ist darüber hinaus der Typ `Animal` nicht als einzelner Import aufgeführt, müssten dabei alle `*`-Importe überprüft werden.

Diese komplexen Überlegungen für die Auflösung von Namen werden in der Symboltabelle gekapselt und gleichzeitig optimiert. Hierzu wird einmalig für jede Sprache festgelegt, welche Elemente außerhalb der Modelle zugreifbar sind. Für diese werden zugeschnittene Einträge in der Symboltabelle abgelegt, die nur die für eine externe Verwendung notwendigen Informationen enthalten. Dadurch entsteht eine Struktur, die das Auflösen von Namen wie `Animal` und die Ermittlung deren Eigenschaften deutlich erleichtert. Gleichzeitig wird eine Schnittstelle geschaffen, um Informationen über Elemente zwischen Modellen auch unterschiedlicher Sprachen auszutauschen.

Die folgenden Konzepte für Architektur und Implementierung von Symboltabellen sind im Rahmen von [Völ11] entstanden. Gleichzeitig wurde ein Framework entwickelt, das die Implementierung von Symboltabellen für auf MontiCore basierenden Sprachen unterstützt. Dieses steht über die `ETS`-Komponente als Teil des MontiCore-Frameworks zur Verfügung (siehe Abschnitt 7.1). Konzepte und Infrastruktur sind dabei parallel zu der vorliegenden Arbeit entstanden, so dass die UML/P sowohl als erster großer Anwendungsfall als auch als Architekturtreiber gedient hat. Im Folgenden wird ein kurzer Überblick der wichtigsten Aspekte im Bezug auf die UML/P-Symboltabelle gegeben. Vertiefende Details und Hintergründe zu den Symboltabellen in MontiCore finden sich in [Völ11].

Abbildung 7.9 zeigt die Hauptbestandteile für die Nutzung der Symboltabellen, wobei für Interfaces und abstrakte Klassen in den Sprachkomponenten des UML/P-Frameworks jeweils auf

die Sprache zugeschnittene Implementierungen existieren. Insgesamt basieren die dargestellten Bestandteile auf den folgenden Konzepten:

- **Symboltabelleneintrag** (engl. "Symboltable-Entry"): Referenzierbare Elemente einer Sprache werden in der Symboltabelle als Instanzen von `STEntry` eingetragen, wobei für jede *Elementart* einer Sprache (engl. "Kind") eine eigene Subklasse von `STEntry` zur Verfügung steht. Diese bieten Zugriff auf die spezifischen Eigenschaften der jeweiligen Elemente, die für eine Überprüfung der korrekten Verwendung einer Referenz erforderlich sind. Dazu werden die Symboltabelleneinträge mindestens anhand des Namens und der Art des zugehörigen Elements unterschieden. Letzteres ist in den Subklassen als statisches Attribut hinterlegt und zusätzlich über die Methode `getKind()` abrufbar. Darüber hinaus können sich Eigenschaften auch durch eine gegenseitige Referenzierung der Symboltabelleneinträge ergeben. In Klassendiagrammen und Java ist etwa der Eintrag für einen Typ mit den Einträgen der darin enthaltenen Methoden verknüpft, die wiederum auf den definierenden Typ sowie auf Einträge für Rückgabe- und Parametertypen verweisen. Auf diese Weise kann zum Beispiel überprüft werden, ob der Aufruf einer Methode in einem Kontext einer Klasse oder einer Variablen erlaubt ist. In diesem Fall verborgen bleibt die Implementierung der Methode, da diese für eine Verwendung keine Rolle spielt. Somit abstrahieren die Symboltabelleneinträge von den Interna der Modellelemente und stellen nur die für eine referenzielle Nutzung relevanten Informationen dar.
- **Gültigkeitsbereich** (engl. "Scope"): Jeder in einem Modell definierte Name hat einen bestimmten Bereich, in dem dieser gültig (sichtbar) ist. Daraus werden die Namensräume abgeleitet.
- **Namensraum** (engl. "Namespace"): Namensräume sind abstrakte Container oder Umgebungen für Namen und deren Abbildung auf Symboltabelleneinträge innerhalb einer Sprache. Sie bestimmen die Menge der Einträge, aus denen bei der Auflösung einer Referenz ausgewählt werden kann. Die Mengen können hierarchisch geschachtelt sein, wobei Einträge übergeordneter Namensräume von darunter liegenden importiert werden können und dadurch ebenfalls referenzierbar sind. Dies wird auch als *Sichtbarkeit* eines Eintrags bezeichnet. Auf diese Weise kann etwa in Methoden ebenfalls auf Attribute und Methoden der zugehörigen Klassen zugegriffen werden. In der Methode definierte Variablen sind jedoch außerhalb dieser nicht sichtbar.
- **Symboltabelle** (engl. "Symboltable"): Symboltabelleneinträge werden innerhalb der Namensräume in Instanzen vom Typ `SymbolTable` verwaltet. Dabei werden Einträge für den Im- und Export voneinander unterschieden. Weitere Unterscheidungen entstehen durch unterschiedliche Sichtbarkeiten von Einträgen, wie sie in der UML/P über Modifikatoren für einige Elemente spezifiziert werden können (siehe Kapitel 3). So kann etwa in einem Klassendiagramm innerhalb einer Methode auf alle Einträge der zugehörigen Klasse zugegriffen werden, nicht aber auf private Einträge von Superklassen.

- **Deserialisierer** (engl. "Deserializer"): Für eine effiziente Verwendung können Symboltabelleneinträge als Dateien abgespeichert (*serialisiert*) und bei Bedarf wieder eingelesen (*deserialisiert*) werden, ohne dass das zugehörige Modell erneut verarbeitet werden muss. Während die Serialisierung über die Symboltabelleneinträge selbst erfolgt, sind die entsprechenden Deserialisierer als Subklassen von `STEntryDeserializer` implementiert.

  Insgesamt wird die Symboltabellenstruktur für ein vorliegendes Modell bedarfsorientiert aufgebaut. Insbesondere Referenzen auf andere Modelle werden während des Aufbaus nur auf Existenz und Eindeutigkeit überprüft. Deren Eigenschaften werden hingegen erst auf Anfrage über die Deserialisierer geladen und die Einträge entsprechend vervollständigt.

- **Namensauflöser** (engl. "Resolver"): Die Ermittlung von Symboltabelleneinträgen erfolgt über eine Instanz der Klasse `Resolver`, wobei im Allgemeinen folgende Angaben erforderlich sind:

  1. Name des Symboltabelleneintrags
  2. Erwartete Elementart
  3. Namensraum als Ausgangspunkt der Suche

  In einigen Fällen müssen für die Auflösung weitere Parameter angegeben werden, um einen Symboltabelleneintrag eindeutig zu bestimmen. Dies ist in der UML/P und Java bei Methoden der Fall, die zusätzlich zum Methodennamen die Angabe der Parametertypen erfordern. Die Auflösung ist somit abhängig von der Elementart und wird jeweils in eigenen Subklassen von `IResolverClient` realisiert, an die der `Resolver` Anfragen weiterleitet.

Die Klasse `SymbolTableInterface` ermittelt diese Bestandteile der Symboltabelle für ein konkretes Modell und bietet damit einen einfachen Zugriff für deren Nutzung. Darüber hinaus stehen verschiedene Methoden für die Auflösung von Symboltabelleneinträgen zur Verfügung. So können Einträge ausgehend von einem Element der abstrakten Syntax ermittelt oder gleichzeitig deren Eigenschaften geladen werden.

Um eine Symboltabelle für ein neues oder geändertes Modell aufzubauen, sind verschiedene Schritte erforderlich. Die Wichtigsten sind im Folgenden gemeinsam mit der dafür von den Sprachen bereitzustellenden Infrastruktur zusammengefasst:

1. **Erstellung von Namensräumen**: Im `ETS`-Framework werden Namensräume anhand der abstrakten Syntax einer Sprache berechnet. Dazu werden in jeder Sprache AST-Klassen bestimmt, die einen neuen Namensraum aufspannen.
2. **Erstellung von Symboltabelleneinträgen**: Für die Erstellung von Einträgen und deren Zuordnung zu einem Namensraum verwendet das `ETS`-Framework Visitoren [GHJV95]. Diese sind in jeder Sprache als Subklassen von `ConcreteASTAndNameSpaceVisitor` realisiert, die vom `ETS`-Framework auf der abstrakten Syntax der Modelle ausgeführt werden.

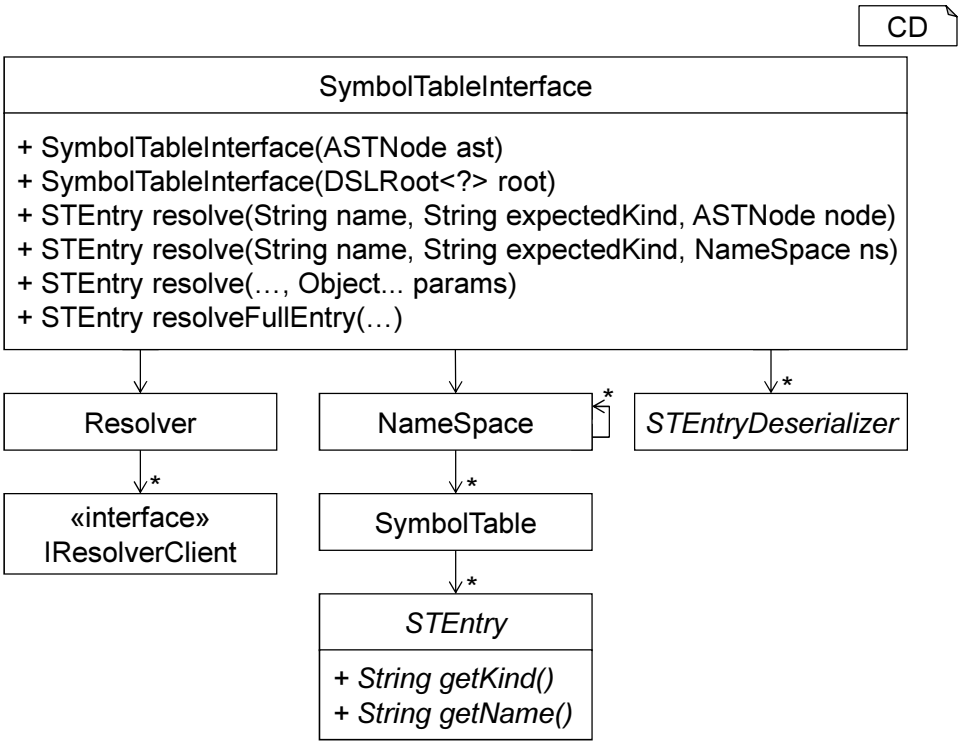


Abbildung 7.9: Schnittstelle zur Symboltabelle

3. **Import von Symboltabellen**: Welche Einträge eines Namensraums ebenfalls in dessen untergeordneten Namensräumen sichtbar sind, wird in den Sprachen als Subklassen von `IInheritedEntriesCalculatorClient` spezifiziert.
4. **Qualifizierung von Symboltabelleneinträgen**: Enthält ein Symboltabelleneintrag Referenzen auf andere Einträge, werden diese in einem ersten Schritt als neue Einträge angelegt und erst bei Bedarf vollständig geladen. Um dennoch die Konsistenz der Symboltabelle sicher zu stellen, wird während des Aufbaus überprüft, ob sich für jede Referenz ein entsprechendes Modellelement eindeutig ermitteln lässt. Dies wird als Qualifizierung bezeichnet und ist in den Sprachen als Subklassen von `IQualifierClient` realisiert.

Wie bereits bei der Projektstruktur in Abschnitt 7.1 beschrieben, sind die Sprachen im UML/P-Framework jeweils in Komponenten für Hauptsprache und Einbettung aufgeteilt (siehe Tabelle 7.3). Auf Klassenebene findet sich diese Aufteilung als Kompositum Entwurfsmuster[6] [GHJV95] wieder. Dies sowie die Zuordnung der sprachspezifischen Infrastruktur sind in Abbildung 7.10 dargestellt.

Für jede Sprache der UML/P ist in der jeweiligen Sprachkomponente eine `LanguageComponent` angelegt. Diese enthält die für die Sprache spezifische Infrastruktur für Aufbau und Nutzung der Symboltabelle sowie die zugehörigen Kontextbedingungen (siehe auch Abschnitt 7.3). Ein-

[6] engl.: “Composite Pattern”

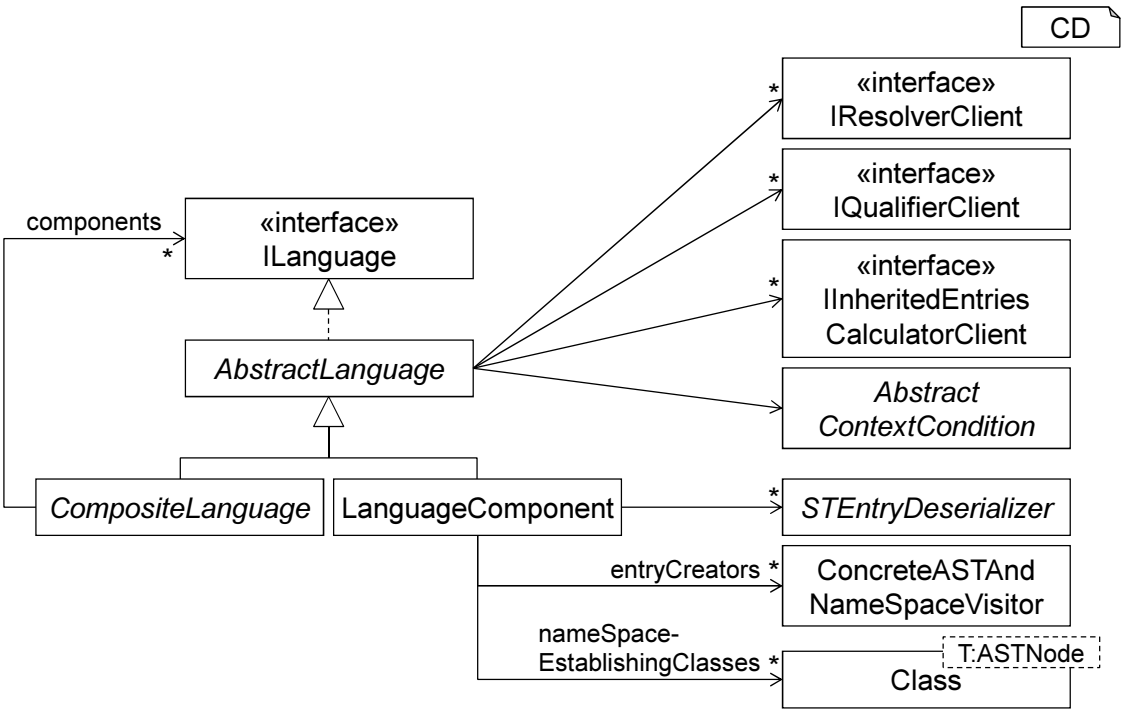


Abbildung 7.10: Spracharchitektur und Zuordnung der sprachspezifischen Infrastruktur

gebettete Sprachanteile werden an dieser Stelle noch nicht berücksichtigt. Dies erfolgt erst in den Einbettungskomponenten, indem die entsprechenden `LanguageComponent`s zu einer `CompositeLanguage` kombiniert werden. Abbildung 7.11 zeigt dies am Beispiel der Klassendiagramme, die sich in der vorliegenden Fassung der UML/P aus Java, OCL/P und der Klassendiagrammsprache selbst zusammensetzt. Diese Sprachkombination wird in der `CDLanguage` zusätzlich mit der Infrastruktur für Sprachverwaltung und Codegenerierung vereint (siehe Abschnitt 7.1). Auf diese Weise kann die `LanguageComponent` von Java und OCL/P jeweils für die Einbettung in den anderen Sprachen der UML/P wieder verwendet werden.

Durch das Hinzufügen zu einer `CompositeLanguage` werden die Bestandteile der einzelnen Sprachkomponenten vereinigt. Diese Vereinigung genügt für einen vollständigen Symboltabellenaufbau, solange in der komponierten Sprache keine Elemente sprachübergreifend referenziert werden sollen. Dies ist z.B. in Klassendiagrammen der Fall, da in den eingebetteten Java- und OCL-Ausdrücken im Allgemeinen auf Typen, Attribute und Methoden des übergeordneten Modells Bezug genommen wird. Andere sprachübergreifende Bezüge entstehen in der UML/P zwischen Modellen und ergeben sich somit erst auf Ebene der Sprachfamilie (siehe auch Abbildung 6.5). So können in allen Modellarten Typen referenziert werden, die entweder in Java oder einem Klassendiagramm spezifiziert sind. Darüber hinaus werden Testfälle in der Testspezifikationssprache durch Referenzen auf Objekt- und Sequenzdiagramme oder Statecharts für eine bestimmte Methode in einem Klassendiagramm modelliert.

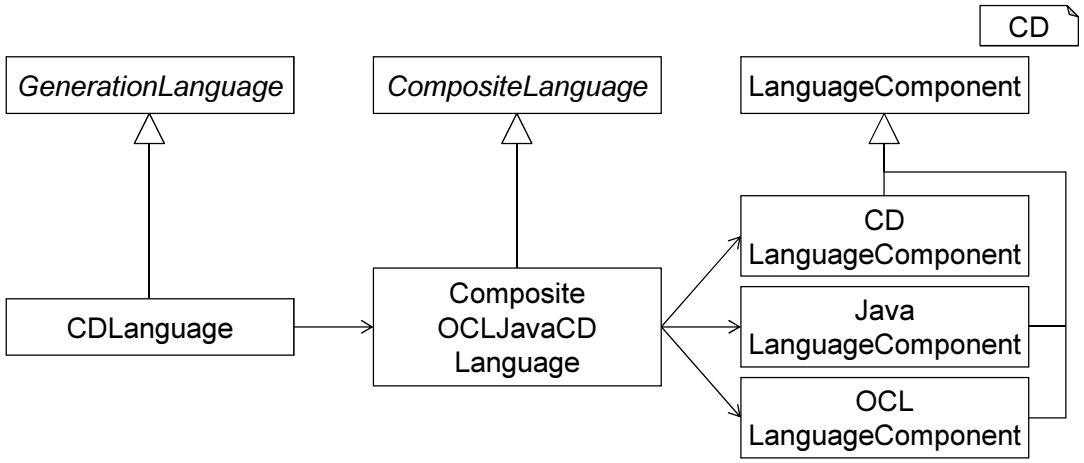


Abbildung 7.11: Komposition der Sprachen am Beispiel der Klassendiagramme

Um diese sprachübergreifenden Referenzen auflösen und Symboltabellen unterschiedlicher Sprachen vereinen zu können, wird für die Symboltabelleneinträge das Adapter Entwurfsmuster [GHJV95] verwendet. Abbildung 7.12 zeigt dies am Beispiel von Typen, die in Klassendiagrammen spezifiziert sind, aber in Java-Ausdrücken aufgelöst werden sollen. Da beide Sprachen auch unabhängig voneinander verwendet werden können und die Spezifikation von Typen erlauben, existiert in den Sprachkomponenten mit `TypeEntry` für Java und `CDTypeEntry` für Klassendiagramme jeweils ein eigener Symboltabelleneintrag. Der Aufbau dieser Einträge ist zwar recht ähnlich. Im Gegensatz zu Java kann ein `CDTypeEntry` jedoch neben Attribut- und Methodeneinträgen unter anderem auch Einträge für Assoziationen enthalten. Um nun einen `CDTypeEntry` in die Symboltabellenstruktur eines Java-Ausdrucks integrieren zu können, wird der `CDTypeToJavaAdapter` als Vermittler verwendet. Dieser übersetzt die Eigenschaften des `CDTypeEntry` konform zur Java-Symboltabelle. So kann der Typname, wie in Abbildung 7.12 gezeigt, direkt übernommen werden. Assoziationen werden hingegen als Java-Attribute interpretiert.

Ein Adapter kapselt somit die Information, wie ein Element einer Sprache als Element einer anderen Sprache interpretiert werden kann. Für eine vollständige Integration können mehrere solcher Adapter erforderlich sein. Insgesamt werden z.B. für die Integration von Java in Klassendiagrammen Adapter für Typen, Attribute, Methoden und Assoziationen benötigt. Diese werden in der UML/P für jede Sprachkombination jeweils über eine Factory zur Verfügung gestellt. Für die Auflösung solcher adaptierten Einträge ist darüber hinaus weitere Infrastruktur wie Namensauflöser oder Qualifizierer notwendig.

Für die Integration von Sprachen als Sprachfamilie wird dasselbe Verfahren angewendet wie beim obigen Beispiel der Spracheinbettung. Insgesamt können bei der Komposition der einzelnen Sprachen in der vorliegenden Fassung der UML/P folgende Adaptierungen unterschieden werden:

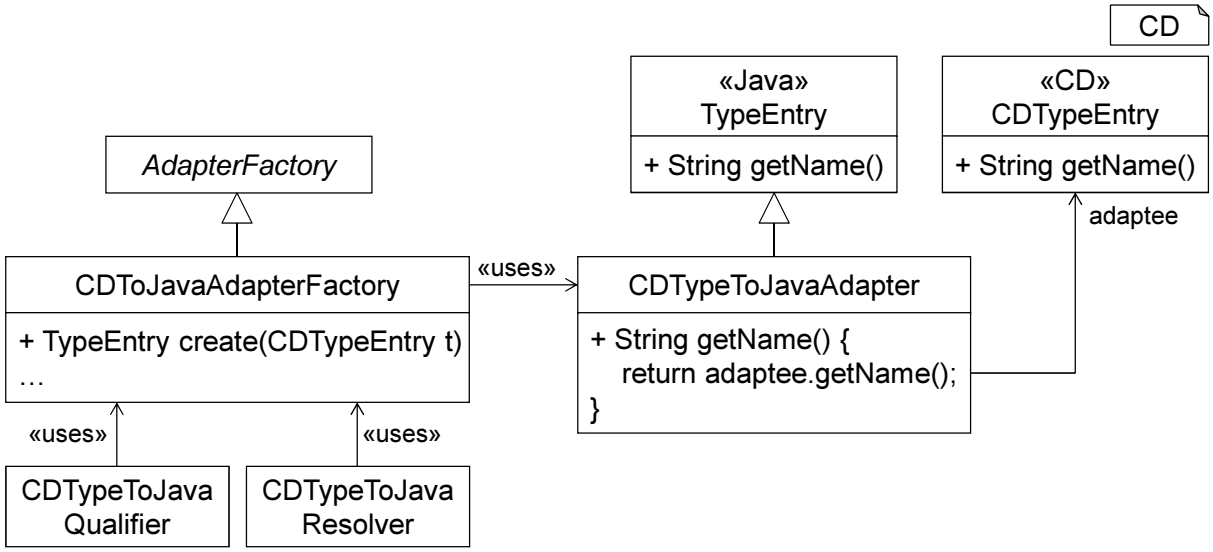


Abbildung 7.12: Adaptierung von Symboltabelleneinträgen am Beispiel von Typen

1. **Typadaptierung**: für die Auflösung von Typreferenzen und deren Schnittstellen. Zu den Schnittstellen gehören im Allgemeinen Attribute, Konstruktoren und Methoden. In Klassendiagrammen zählen dazu außerdem noch die Assoziationen, die bei der Adaptierung in Sprachen wie Java als Attribute interpretiert werden können.
2. **Objektadaptierung**: für die Auflösung von Objektreferenzen. Objekt- und Sequenzdiagramme bieten eigene Symboltabelleneinträge für Objekte, die entsprechend für die Adaptierung verwendet werden. In anderen Sprachen werden Objektreferenzen hingegen wie Referenzen auf Attribute oder Variablen genutzt und interpretiert.
3. **Modelladaptierung**: für die Auflösung von Referenzen auf Modelle. Dies wird zurzeit nur für die Spezifikation von Testfällen benötigt und stellt somit einen Sonderfall dar. Die Nutzung dieser Adaptierung beschränkt sich auf die Existenzprüfung der referenzierten Objekt- und Sequenzdiagramme. Die darin spezifizierten Objekte werden hingegen über die Objektadaptierung integriert.

Die für jede Sprachkombination notwendigen Adaptierungen lassen sich aus Abbildung 7.8 ablesen. Unabhängig davon, ob die Komposition durch Einbettung oder Referenzierung erfolgt, werden die Symboltabellen der äußeren bzw. referenzierten Sprache durch Typadaptierung in die jeweils andere Sprache integriert. Für die Integration von Objekt- oder Sequenzdiagrammen ist darüber hinaus zusätzlich eine Objektadaptierung, sowie innerhalb der Testspezifikationssprache eine Modelladaptierung erforderlich.

In der vorliegenden Fassung der UML/P können Typen nur in Klassendiagrammen und Java definiert werden. Dennoch enthalten auch die anderen Sprachen eigene Symboltabelleneinträge für Typen, um die Symboltabellen der einzelnen Sprachen unabhängig voneinander aufbauen und integrieren zu können. Auf diese Weise können z.B. die Symboltabellen von Objekt- und

Sequenzdiagrammen aufgebaut werden, ohne nach Herkunft eines Typeintrags zu unterscheiden. Gleichzeitig sind dadurch die möglichen Sprachen für Typspezfikationen nicht festgelegt, so dass eine Verwendung der einzelnen Modellarten auch unabhängig von Klassendiagrammen oder Java möglich ist. Dabei beschreiben die zusätzlichen Einträge die Informationen, die eine Typspezifikation liefern muss, um als solche innerhalb einer Modellart akzeptiert zu werden. Objekt- und Modelladaptierungen wurden auf die gleiche Weise umgesetzt. Die Verbindung zwischen den Sprachen erfolgt somit ausschließlich mit dem beschriebenen Adapteransatz, so dass die Abhängigkeiten in den Adaptern gekapselt und damit modifizierbar sind. Dies sichert die Eigenständigkeit der Sprachkomponenten und erlaubt eine größtmögliche Flexibilität bei der Verwendung der einzelnen Sprachen auch in anderen als den in dieser Arbeit beschriebenen Kontexten.

Bezüglich der in Abschnitt 7.1 beschriebenen Projektstruktur enthalten somit die Sprachkomponenten sämtliche Infrastruktur, die für den Aufbau der Symboltabelle einer Sprache notwendig ist. Abhängigkeiten zu anderen Sprachen bestehen an dieser Stelle nicht. In den Einbettungskomponenten bzw. in der Komponente `umlpFE`, die die Sprachen zu einer Familie zusammenfasst, kommen hingegen nur noch die für die Integration der jeweiligen Symboltabellen erforderlichen Elemente hinzu. Diese umfassen im Wesentlichen Adaptoren sowie weitere Infrastruktur für Namensauflösung und Qualifizierung der adaptierten Einträge. Darüber hinaus wird in zusätzlichen `IInheritedEntriesCalculatorClient`s bestimmt, welche Symboltabellen einer anderen Sprache importiert werden. So ist für die Modellierung von Verhalten in Statecharts oder Sequenzdiagrammen, sowie die Darstellung des Zustands von Objekten in Objektdiagrammen teilweise die Auflösung von privaten Elementen der jeweiligen Typen für eine vollständige Darstellung nötig. Andere Typreferenzen erlauben hingegen nur einen eingeschränkten Zugriff.

Die Flexibilität der Sprachkomposition durch Adaptierung von Symboltabelleneinträgen und zusätzlichen Auflösungsmechanismen zeigt sich auch an der Sprache Java/P. Diese unterscheidet sich von der Sprache Java nur durch die Möglichkeit, Elemente der Modellebene verwenden zu können. Diese Eigenschaft wurde allein mit Hilfe der beschriebenen Integrationsmechanismen der Symboltabellen umgesetzt. Aus diesem Grund können Java und Java/P an vielen Stellen, wenn es nur um die Syntax geht, als gleichbedeutend angesehen werden, nicht jedoch, wenn ebenfalls die Eigenschaften relevant sind.

7.3 Framework für Kontextbedingungen

Kontextbedingungen werden in dieser Arbeit nicht als fester und unveränderlicher Bestandteil der UML/P angesehen, sondern sind Teil der flexiblen Konfiguration und Einsatzmöglichkeiten der Sprache und des Frameworks (vgl. auch Abbildung 7.1). Die in Kapitel 4 aufgestellten Kontextbedingungen stellen somit nur einen Basissatz dar, der zwar gemeinsam mit dem UML/P-Framework ausgeliefert wird, aber aus verschiedenen Gründen angepasst und erweitert werden kann:

- **Semantische Variationspunkte**: Aufgrund der Abstraktion ist die Interpretation von Modellen nicht immer eindeutig (siehe auch Abschnitt 6.1). Solche semantischen Variationspunkte können für einzelne Modelle durch Stereotypen oder als Konvention für alle Modelle eines Projekts konkretisiert werden. Neben der Berücksichtigung bei der Umsetzung der Generatoren können entsprechend angepasste Kontextbedingungen eine solche Festlegung bereits auf Modellebene unterstützen.
- **Profile**: Mit Hilfe von Profilen ist es möglich, die UML/P auf bestimmte Anwendungsfälle oder Domänen anzupassen (siehe auch Abschnitt 6.2). Die automatisierte Verarbeitung entsprechender Modelle durch das UML/P-Framework kann wie bei den semantischen Variationspunkten durch angepasste Kontextbedingungen und Codegeneratoren umgesetzt werden.
- **Entwicklungsphasen**: Modelle in einer frühen Phase der Softwareentwicklung werden im Allgemeinen auf einem hohen Abstraktionslevel entworfen und erst im weiteren Verlauf verfeinert, bis sie sich schließlich zur Codegenerierung eignen. Aus diesen Phasen können sich unterschiedliche Anforderungen an die Modellinhalte ergeben und mit entsprechend angepassten Kontextbedingungen überprüft werden.
- **Codegeneratoren**: Anforderungen an Modelle können sich auch aus der Umsetzung von Codegeneratoren oder der Zielplattform ergeben. So kann etwa Mehrfachvererbung in Klassendiagrammen verboten sein, wenn eine Zielsprache der Generierung wie Java dieses Konzept nicht unterstützt und der Codegenerator keine semantisch äquivalente Übersetzung bietet.
- **Sprachvariationen**: Das im Rahmen dieser Arbeit entstandene Framework der UML/P wurde so entworfen, dass sich die einzelnen Sprachen auch unabhängig voneinander einsetzen und durch neue Sprachen ergänzen lassen. Solche Änderungen der Sprachzusammensetzung oder -verwendung können sich ebenfalls auf die Kontextbedingungen auswirken.

Vor diesem Hintergrund wurde im Rahmen von [Völ11] und der vorliegenden Arbeit die in Abbildung 7.13 dargestellte Architektur für Kontextbedingungen entworfen. Das zugrunde liegende Kompositum-Entwurfsmuster [GHJV95] erlaubt es, einzelne Bedingungen zu immer komplexeren Bedingungen zusammen zu fassen. Auf diese Weise können die in Kapitel 4 spezifizierten Kontextbedingungen in der Implementierung unterteilt oder anhand von Modellelementen oder Kategorien wie Typkompatibilität gruppiert werden. Gleichzeitig ist jederzeit eine Erweiterung durch zusätzliche Bedingungen möglich.

Die als Attribut `checkedProperty` hinterlegte Kennung dient der Identifizierung einer Kontextbedingung innerhalb des Frameworks. Darüber hinaus kann eine Bedingung aktiviert bzw. deaktiviert und ein Schweregrad zugeordnet werden (vgl. auch Kapitel 4), wobei sich eine Änderung dieser Eigenschaften jeweils auf alle Kontextbedingungen innerhalb eines Kompositums auswirkt. Dies erleichtert somit Auswahl und Konfiguration der Kontextbedingungen. Die Implementierung der Bedingungen findet schließlich in Subklassen von `ContextCondition` statt.

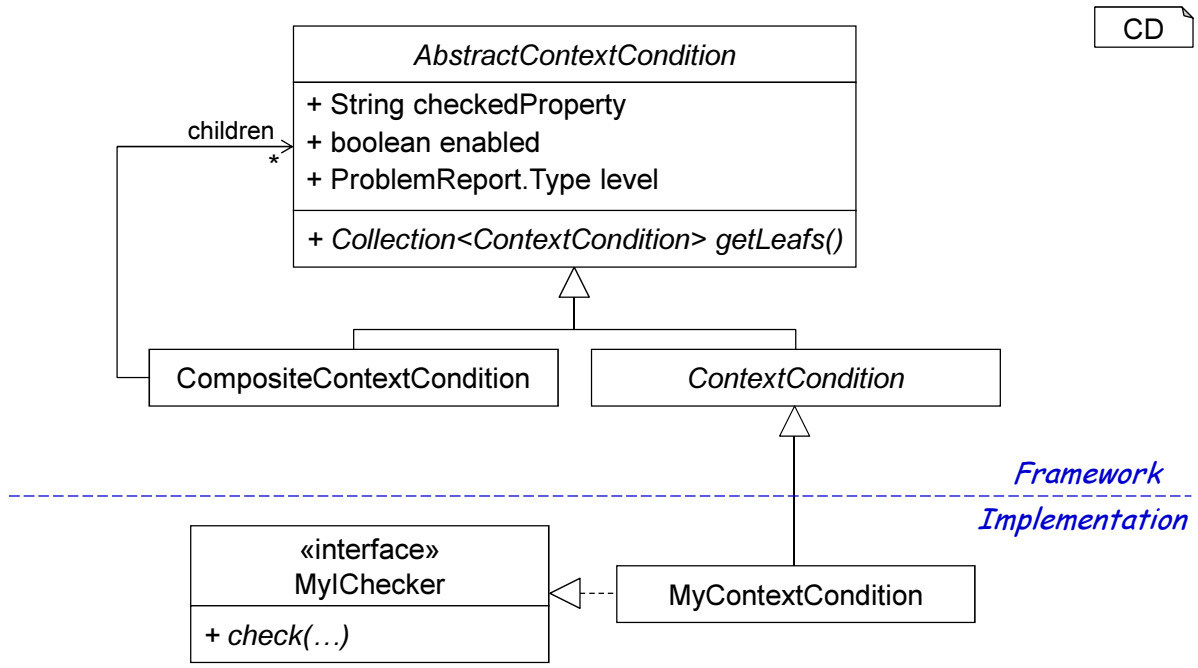


Abbildung 7.13: Architektur der Kontextbedingungen

Innerhalb des Frameworks werden die Kontextbedingungen einer Sprache (`LanguageComponent`) oder Sprachkomposition (`CompositeLanguage` bzw. `LanguageFamily`) zugeordnet (vgl. Abbildung 7.10). Aus diesen kann für die Modellverarbeitung gezielt ausgewählt sowie der Schweregrad angepasst werden. Zurzeit ist hierfür die Methode `setCocoConfiguration` im `ETSTool` vorgesehen, der eine Map der zu aktivierenden Kontextbedingungen bestehend aus den jeweiligen Kennungen und dem gewünschten Schweregrad übergeben wird. In zukünftigen Versionen des Frameworks könnte diese Konfiguration der Kontextanalyse auch mit Hilfe angepasster Konfigurationsdateien erfolgen. Darüber hinaus bietet das `ETSTool` Zugriff auf die zugeordneten Sprachen, so dass auch das Hinzufügen von Bedingungen möglich ist (vgl. Abbildung 7.7).

Quellcode 7.14 zeigt ein Beispiel für die Implementierung einer Kontextbedingung. Diese überprüft für jeden Symboltabelleneintrag einer Klasse innerhalb eines Klassendiagramms, dass keine finalen Klassen erweitert wurden (vgl. CD-1-04 in Abschnitt 4.2.2). Da die Symboltabelle aus Gründen der Effizienz Referenzen auf Elemente außerhalb des zu prüfenden Modells erst bei Bedarf vollständig auflöst, werden die Einträge für die Superklassen in den Zeilen 11-13 zuerst nachgeladen und schließlich in den Zeilen 14-17 überprüft. Im Konstruktor wird das Attribut `checkedProperty` gesetzt, die in der UML/P für jede Sprache in einer Klasse als Konstanten gesammelt werden.

Ebenfalls an diesem Beispiel zeigt die Abbildung 7.15 die für die Ausführung der Kontextbedingungen verwendete Infrastruktur. Deren zentrale Klasse ist der `CheckWorkflow` (vgl. auch Abbildung 7.1), der die Kontextbedingungen anhand der Konfiguration aktiviert und an die `CheckWorkflowClients` weiterreicht. Bei letzteren handelt es sich um Visitoren, die auf der ab-

```java
public class ClassesCanOnlyExtendNonFinalClassesChecker
extends ContextCondition implements ICDClassChecker {

  public ClassesCanOnlyExtendNonFinalClassesChecker() {
    super(CDContextConditionConstants.ClassesExtendNonFinalClasses_Property);
  }

  @Override
  public void check(ASTCDClass cdClass, CDTypeEntry typeEntry) {
    for (CDTypeEntry superType : typeEntry.getSuperClasses()) {
      if (superType.getEntryState() != STEntryState.FULL) {
        superType.toFullVersion(loader, deserializers);
      }
      if (superType.isFinal()) {
        addReport("Class " + cdClass.getName() + " extends the final class "
                  + superType, cdClass.get_SourcePositionStart());
      }
    }
  }
}
```

Java

Quellcode 7.14: Java-Implementierung einer Kontextbedingung

strakten Syntax der zu prüfenden Modelle ausgeführt werden. Die `CheckWorkflowClient`s sind somit spezifisch für eine Sprache und stehen in den jeweiligen Sprachkomponenten zur Verfügung. Ihre Aufgabe ist es, die Daten der abstrakten Syntax für die Analyse der Kontextbedingungen aufzubereiten und die entsprechenden Bedingungen aufzurufen. Dieser Aufruf erfolgt über verschiedene Interfaces, durch deren Implementierung die Kontextbedingungen Zugriff auf den jeweils benötigten Datensatz erhalten. Die Aufbereitung wurde aus Effizienzgründen zentralisiert, so dass dieselben Daten mehrmals verwendet werden können, ohne sie in jeder Kontextbedingung erneut zu berechnen. Gleichzeitig erleichtert dies die Implementierung zusätzlicher Kontextbedingungen. Sollten dafür spezifische Vorberechnungen notwendig sein, können diese als weitere `CheckWorkflowClient`s den Sprachen hinzugefügt werden. Ansonsten genügt es, ein passendes Interface zu implementieren und die Kontextbedingung der entsprechenden Sprache über die Schnittstellen von `AbstractLanguage` oder `LanguageFamily` hinzuzufügen, um die Analyse um zusätzliche Bedingungen zu erweitern.

Durch den in Abschnitt 7.2 beschriebenen Adapteransatz für die Integration von Symboltabellen unterschiedlicher Sprachen ist es möglich, auch die Überprüfung sprachübergreifender Kontextbedingungen unabhängig von der konkreten Sprachkomposition zu implementieren. Dafür stehen in den Sprachkomponenten teilweise zusätzliche Subklassen von `STEntry` für Symboltabelleneinträge zur Verfügung, die sich nicht aus den Elementen der Sprache selbst ergeben. Ein Beispiel dafür sind Einträge für Typen, die auch in Sprachen wie Objekt- oder Sequenzdiagrammen existieren, obwohl eine Typdefinition in diesen Sprachen nicht möglich ist. Der damit verbundene komplexere Symboltabellenaufbau rentiert sich in sofern, als dass die als `ContextCondition` implementierten Überprüfungen auch für andere Sprachkompositionen wieder verwendet werden können. So überprüft das Beispiel in Quellcode 7.14 gleichzeitig auch

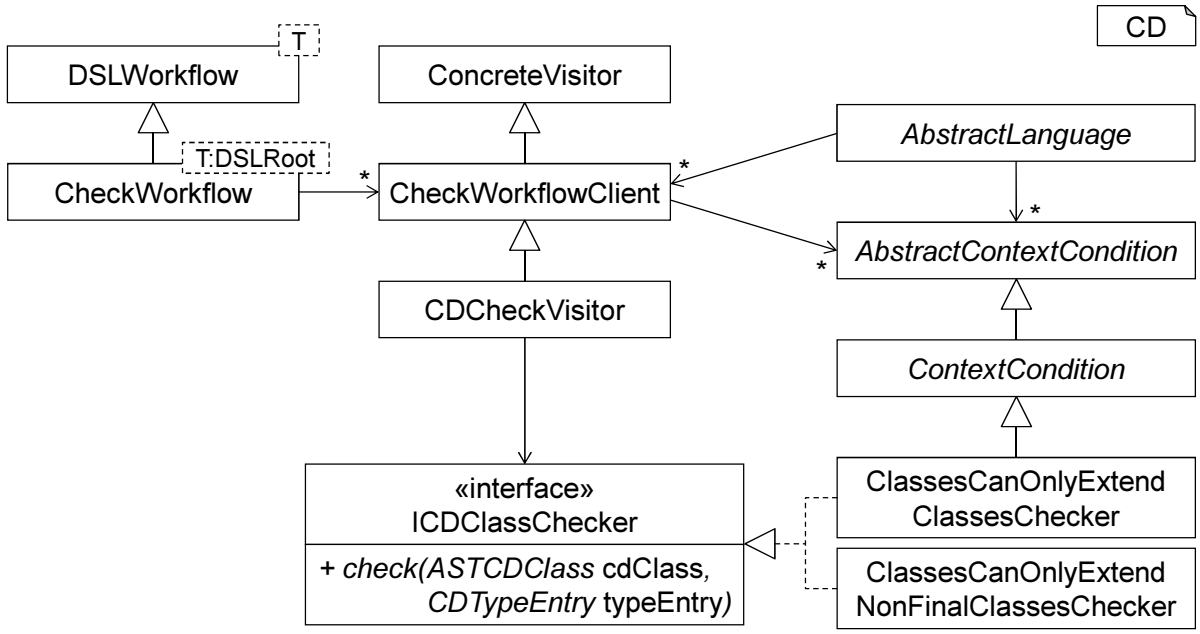


Abbildung 7.15: Einbindung der Kontextbedingungen im UML/P-Framework

Typeinträge von Java-Klassen, wenn diese über einen entsprechenden Adapter in die Symboltabelle von Klassendiagrammen eingebunden sind (vgl. CD-4-01 in Abschnitt 4.5.1).

Insgesamt sind die im Kapitel 4 aufgeführten sprachübergreifenden Intra- und Inter-Modell-Bedingungen erst dann überprüfbar, wenn über eine entsprechende Adapterinfrastruktur die notwendigen Informationen der eingebetteten bzw. referenzierten Sprache zur Verfügung stehen. Darüber hinaus beziehen sich einige Kontextbedingungen wie etwa CD-3-08 in Abschnitt 4.4.2 auf die an einer bestimmten Stelle im Modell referenzierbaren Elemente. Dies wird durch den Aufbau von Namensräumen und einem entsprechenden Import von Symboltabelleneinträgen realisiert (siehe Abschnitt 7.2), wobei gleichzeitig die Eindeutigkeit der Referenzen überprüft wird. Die Erfüllung dieser Kontextbedingungen ergibt sich somit bereits aus dem Aufbau der Symboltabelle. Trotzdem können auch diese Bedingungen durch die Implementierung zusätzlicher Kontextbedingungen eingeschränkt werden. Um die Referenzierung weiterer Elemente zu ermöglichen, ist hingegen in diesen Fällen eine entsprechende Erweiterung der Symboltabelle etwa um zusätzliche Adapter erforderlich.

Eine solche Anpassung der Symboltabelle ist jedoch nur für die Entwicklung von Sprachvariationen der UML/P notwendig. Für andere Modifikationen der Kontextanalyse bezüglich semantischer Variationspunkte, Profilen, Entwicklungsphasen oder Codegeneratoren genügt im Allgemeinen die Erweiterung und Konfiguration der Kontextbedingungen mit denen in diesem Abschnitt vorgestellten Konzepten. Das Framework der UML/P bietet damit insgesamt die Möglichkeit, Kontextbedingungen an die spezifischen Anforderungen von Unternehmen, Projekten oder Domänen anzupassen.

7.4 Generierungsframework

Das Ziel bei der Entwicklung des Generierungsframeworks war eine Infrastruktur, die eine Implementierung von Codegeneratoren für die Sprachen der UML/P weitestgehend auf Basis von Templates ermöglicht. Dazu wurde ein Konzept entwickelt, bei dem die Templates auf die abstrakte Syntax der Sprachen als Datenmodell aufsetzen. Templateaufruf und Navigation entlang der abstrakten Syntax erfolgen dabei in einem integrierten Prozess, der von den Templates selbst gesteuert wird. Dadurch bilden die Templates das zentrale Artefakt der Codegenerierung. Dieses Konzept und der sich daraus ergebende Aufbau sowie die Implementierung von Codegeneratoren wurden bereits in Kapitel 5 ausführlich behandelt. Im Folgenden wird nun die Infrastruktur beschrieben, auf der die Verarbeitung der Templates basiert.

Abbildung 7.16 zeigt die zentralen Klassen des Generierungsframeworks. Dessen Aufruf erfolgt über ein `GenerationTool`, das die Basis für sprachspezifische Werkzeuge wie das `UMLPTool` bildet (vgl. Abbildung 7.7). Diesem werden die zu verwendenden Templates als Parameter übergeben (siehe auch Abschnitt 7.6). Die Parameter setzen sich jeweils aus dem Namen eines Templates und dem Namen einer Grammatikregel zusammen, wobei letztere angibt, auf welchem Element der abstrakten Syntax das Template ausgeführt werden soll. Statt der Grammatikregel ist alternativ auch die Angabe des voll-qualifizierten Namens der AST-Klasse möglich. Darüber hinaus kann ein Zielverzeichnis für die generierten Dateien festgelegt werden.

Diese Daten werden in der Klasse `GeneratorSetup` gespeichert und dem generischen `CodegenWorkflow` übergeben, der für jede `GenerationLanguage` zur Verfügung steht (vgl. Abbildung 7.1 und 7.7). Der Workflow durchläuft die vorab vom Parser erstellte abstrakte Syntax der Modelle (AST) mittels eines Visitors, der für jeden AST-Knoten entsprechend dem `GeneratorSetup` die zugehörigen Templates aufruft. Innerhalb des Templates kann auf den jeweiligen AST-Knoten über die Variable `ast` zugegriffen werden.

Als zentrale Schnittstelle für die Templateverarbeitung dient der `TemplateOperator`, der in den Templates über die Variable `op` verwendet wird. Dieser stellt die Verbindung der Codegenerierung zur FreeMarker-Templateengine in der Komponente `MontiCoreFreeMarker` her. Die wichtigsten Zugriffsmethoden des `TemplateOperator`s fasst Tabelle 7.17 zusammen.

FreeMarker und andere Templatesprachen bieten im Allgemeinen nur einfache Kontrollstrukturen und eine recht begrenzte Funktionsbibliothek. Auch wenn der obige Ansatz die Implementierung von Codegeneratoren allein mit Templates ermöglicht, sind komplexere Berechnungen oft komfortabler und effizienter in höheren Programmiersprachen wie Java zu programmieren. Andererseits bieten Templates den Vorteil einer intuitiven und Zielcode-nahen Implementierung. Um diese Vorteile zu vereinen und gleichzeitig die Templates als Ausgangspunkt der Codegenerierung beizubehalten, wurde in dieser Arbeit das Konzept der Kalkulatoren entwickelt (siehe Abschnitt 5.2). Dabei handelt es sich um Java-Klassen, die die in Abbildung 7.18 gezeigte abstrakte Klasse `TemplateCalculator` implementieren. Da eine direkte Instanziierung von Java-Klassen in FreeMarker-Templates nicht möglich ist, erfolgt deren Aufruf über die Methode `callCalculator` im `TemplateOperator`, die als Argument den Namen der Kalkulator-Klasse

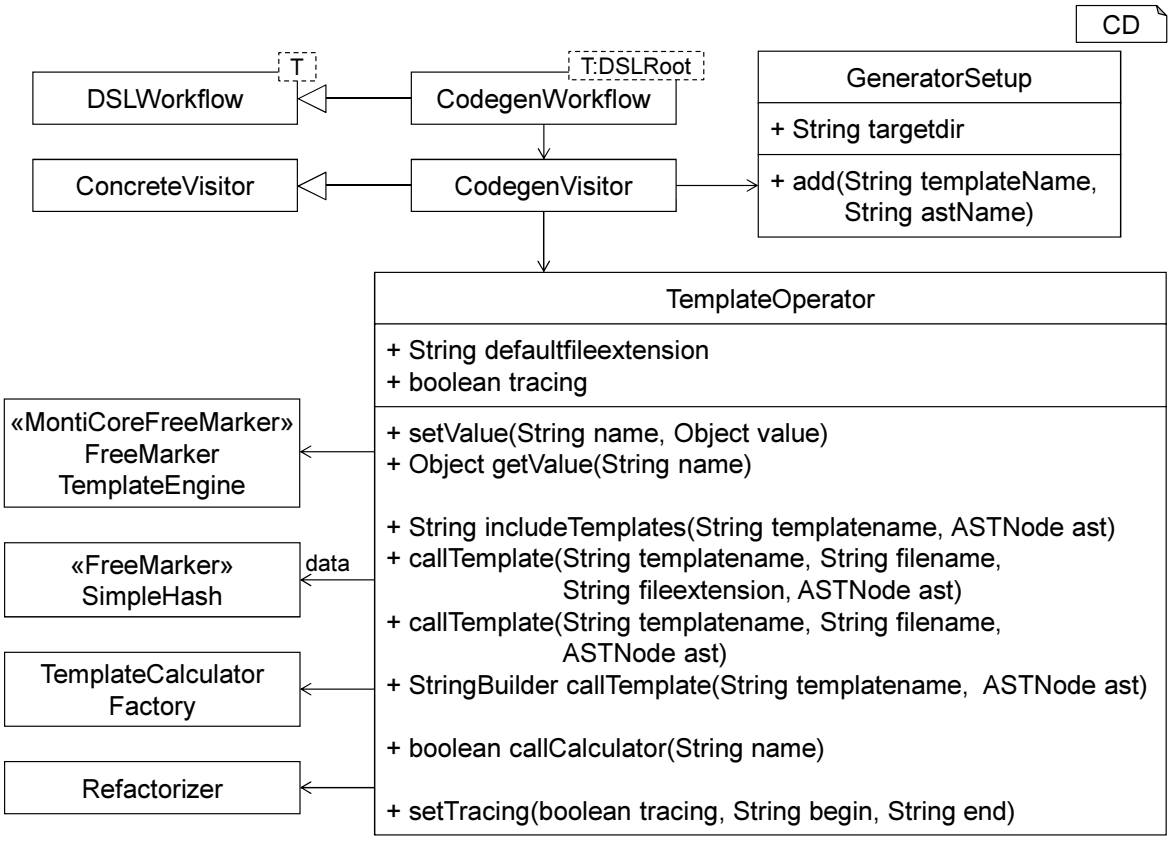


Abbildung 7.16: Architektur des Codegenerierungsframeworks

erwartet. Dadurch wird der Kalkulator über die `TemplateCalculatorFactory` instanziiert und dessen `calc`-Methode mit dem aktuellen AST-Knoten und dem `TemplateOperator` aufgerufen. Somit stehen in den Kalkulatoren dieselben Informationen wie im Template zur Verfügung. Auch das Setzen und Abrufen von Variablen ist über die Methoden des `TemplateOperator`s möglich.

Neben der Nutzung des Funktionsumfangs von Java bieten die Kalkulatoren außerdem den Vorteil, dass die Anzahl der Kontrollstrukturen in den Templates reduziert wird und damit gleichzeitig der Anteil der Zielsprache steigt. Dadurch werden die Templates insgesamt lesbarer und das resultierende Generat lässt sich leichter daraus ableiten, was sich wiederum positiv auf die Wartbarkeit der Generatoren auswirkt.

Das Generierungsframework kann unabhängig von Ausgangs- und Zielsprache für die Codegenerierung eingesetzt werden. Einzige Voraussetzung ist, dass für die Ausgangssprache eine `GenerationLanguage` auf Basis einer MontiCore-Grammatik existiert, die für die Verarbeitung der Sprachfamilie eines `GenerationTool`s hinzugefügt werden kann (siehe Abbildung 7.7). Aufgrund dieser universellen Einsatzmöglichkeiten wurde das Generierungsframework mittlerweile in MontiCore integriert und steht somit allen Sprachen, die mit MontiCore entwickelt wurden, zur Verfügung. Darüber hinaus wird es innerhalb von MontiCore für die Generierung der abstrakten Syntax im Rahmen der Sprachverarbeitung eingesetzt. Innerhalb kürzester Zeit wurde

Methode	*Beschreibung*
`setValue`	Setzen einer Variable, die im aktuellen Template und dessen Subtemplates zugreifbar und überschreibbar ist. Daneben gibt es weitere Methoden, um nicht überschreibbare Konstanten zu definieren (`setConstant`) und bestehende Werte zu erweitern (`addValue`), wobei Einzelwerte automatisch in Listen umgewandelt werden.
`includeTemplates`	Einbettung von Templates, wobei alternativ jeweils auch Listen von Templatenamen und/oder AST-Knoten angegeben werden können, über die automatisch iteriert wird.
`callTemplate`	Aufruf von Templates und Ablage als Datei. Fehlt die Angabe eines Dateinamens, wird das Ergebnis als `StringBuilder` zurückgegeben.
`callCalculator`	Aufruf von Kalkulatoren, die es in den Templates ermöglichen, komplexe Berechnungen als Java-Klassen auszulagern.
`setDefaultFileextension`	Setzen eines Standardwertes für die beim `callTemplate`-Aufruf verwendete Dateiendung. Vorgegeben ist `.java`.
`setTracing`	Bei aktiviertem Tracing werden die Namen der verwendeten Modelle und Templates dem jeweils resultierenden Generat als Kommentar hinzugefügt, wobei die Kommentarbegrenzung festgelegt werden kann. Vorgegeben ist die Einbettung von Java-Kommentaren.

Tabelle 7.17: Schnittstellenübersicht des `TemplateOperator`s (siehe auch Abbildung 7.16)

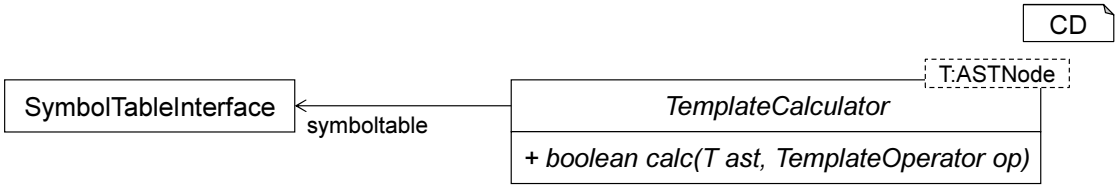


Abbildung 7.18: Architektur der Template-Kalkulatoren

infolgedessen ein zweiter alternativer Generator entwickelt, der eine EMF-kompatible abstrakte Syntax aus den Grammatiken erzeugt. Beides demonstriert die Flexibilität und Anwendbarkeit des Generierungsansatzes.

Das Ergebnis der Templateverarbeitung kann optional durch Refactorings modifiziert werden. Diesen Schritt führt ebenfalls der `TemplateOperator` aus. Die dafür benötigte Infrastruktur wird im Abschnitt 7.5 behandelt.

7.5 Refactoringframework

Das Refactoringframework in Abbildung 7.19 stellt eine optionale Ergänzung der Codegenerierung dar. Diese kann genutzt werden, um im Anschluss an die Templateverarbeitung mit Hilfe von AST-Transformationen das Generat zu modifizieren. Solche abschließenden Refactorings sind etwa dann sinnvoll, wenn Code wie das in den Modellen eingebettete Java, bis auf geringfügige Änderungen für das finale Generat übernommen werden kann. Details hierzu wurden bereits in Abschnitt 5.2 diskutiert.

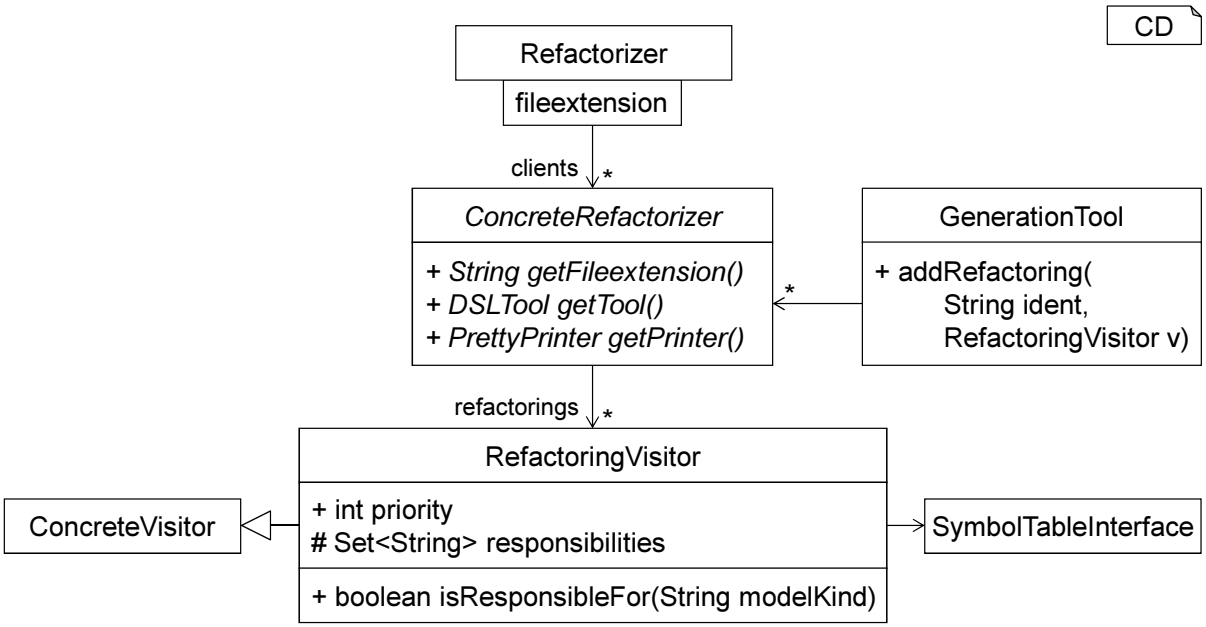


Abbildung 7.19: Architektur des Refactoringframeworks

Der Refactoring-Ansatz beruht auf der Idee, das Ergebnis der Templateverarbeitung mit einem Parser in die abstrakte Syntax zu überführen, diese mittels Visitoren zu transformieren und schließlich über einen Pretty-Printer den finalen Code auszugeben (vgl. Abbildung 5.17 in Abschnitt 5.2). Dies kann etwa genutzt werden, um Attributzugriffe durch Aufrufe entsprechender get- und set-Methoden oder Objektinstanziierungen durch Factory-Aufrufe im generierten Code zu ersetzen. Ausgeführt werden diese Refactorings im `TemplateOperator` über den `Refactorizer` noch bevor das von den Templates erstellte Generat als Datei gespeichert wird.

Für die Verarbeitung eines Generats benötigt der `Refactorizer` ein `DSLTool` zur Erstellung der abstrakten Syntax und einen Pretty-Printer für die abschließende Ausgabe. Beides wird für jeweils eine Zielsprache über `ConcreteRefactorizer` zur Verfügung gestellt, die im `GenerationTool` registriert werden können. Deren Auswahl erfolgt anhand der Dateiendung des Generats.

Auch die als Visitoren implementierten Refactorings werden über das `GenerationTool` hinzugefügt oder alternativ direkt in den Sprachen registriert, um sie gemeinsam mit der Sprach-

komponente zur Verfügung zu stellen (vgl. Abbildung 7.7). Dabei wird für jedes Refactoring ein Name vergeben, über den die Refactorings im Rahmen der Codegenerierung zu steuern sind. Wie bei den Generatoren erfolgt diese Steuerung beim Aufruf des `GenerationTools`, dem die Namen der auszuführenden Refactorings als Parameter übergeben werden (siehe auch Abschnitt 7.6). Dabei ist ebenfalls eine Einschränkung der Refactorings auf die Generate aus bestimmten Modellarten möglich. Auf diese Weise lassen sich die Refactorings je nach Ausgangssprache innerhalb eines Generierungsaufrufs unterschiedlich konfigurieren. Abrufbar ist dies über die Methode `isResponsibleFor` des `RefactoringVisitors`.

Entsprechend der Konfiguration werden die Visitoren vom `Refactorizer` auf der abstrakten Syntax des Generats aufgerufen und können somit auf Instanzen bestimmter AST-Klassen durch die Implementierung von `visit`-Methoden reagieren. Die genaue Umsetzung der AST-Transformationen ist dabei vom Framework nicht vorgegeben. So kann etwa der AST über dessen Schnittstellen direkt verändert oder zusätzliche unterstützende Infrastruktur für die Transformationen eingesetzt werden.

Quellcode 7.20 zeigt die Implementierung eines `RefactoringVisitors` für die Zielsprache Java. Dieser setzt alle privaten Methoden auf `protected`, so dass sie in Subklassen überschrieben werden können. Ein solches Refactoring ist etwa sinnvoll, um für Testfälle den generierten Code nach dem Ansatz in Abschnitt 6.4 vollständig zu instrumentieren. Um im Produktivsystem die privaten Methoden zu erhalten, genügt es, die Generierung ohne dieses Refactoring auszuführen.

Java

```
1  public class PrivateMethodRefactoringVisitor extends RefactoringVisitor {
2
3    public PrivateMethodRefactoringVisitor(int id) {
4      super(id);
5    }
6
7    public void visit(ASTMethodDeclaration ast) {
8      for (ASTModifier m : ast.getModifiers()) {
9        if (m instanceof ASTPrimitiveModifier) {
10         ASTPrimitiveModifier pm = (ASTPrimitiveModifier) m;
11         if (pm.getModifier() == ASTConstantsJavaDSL.PRIVATE) {
12           pm.setModifier(ASTConstantsJavaDSL.PROTECTED);
13         }
14       }
15     }
16   }
17
18 }
```

Quellcode 7.20: Java-Implementierung eines Refactorings

Neben einem Namen kann den Refactorings bei der Registrierung zusätzlich eine Priorität[7] zugeordnet werden. Mit Hilfe dieser Zahl lassen sich die Refactorings in mehrere Durchläufe einteilen, wobei Visitoren gleicher Priorität im selben Durchlauf und ansonsten getrennt nach aufsteigender Priorität ausgeführt werden. Ein Durchlauf besteht dabei aus folgenden Schritten:

[7] engl. priority

1. Einlesen des Generats
2. Ausführung der Refactorings der entsprechenden Priorität
3. Ausgabe des transformierten ASTs über den Pretty-Printer

Dieses Vorgehen ist dann sinnvoll, wenn ein Visitor auf Informationen der Symboltabelle zugreift, diese aber aufgrund vorheriger Refactorings nicht mehr konsistent zum AST ist. Durch das erneute Einlesen der Pretty-Print-Ausgabe des vorangegangen Durchlaufs wird automatisch die Symboltabelle für den transformierten AST neu aufgebaut. Andernfalls müsste neben dem AST auch die Symboltabelle in den Visitoren transformiert werden, was fehleranfällig ist und zu einem entsprechend erhöhten Implementierungsaufwand führen würde. Dies würde selbst Refactorings betreffen, die die Symboltabelle nicht benötigen, um die Konsistenz für nachfolgende Transformationen sicher zu stellen.

Abhilfe könnte eine Infrastruktur schaffen, die über eine API eine inkrementelle Neuberechnung oder die gleichzeitige Transformation von AST und Symboltabelle erlaubt. Zwischen diesen beiden Strukturen sind jedoch vom Symboltabellenframework keine festen Beziehungen vorgesehen, so dass eine solche Infrastruktur für eine Sprache nicht ohne zusätzlichen Aufwand umsetzbar wäre. Der hier verwendete Ansatz kann hingegen unabhängig von der Zielsprache der Generierung angewendet werden.

Grundvoraussetzung für den Einsatz des Refactoringframeworks im Rahmen einer Codegenerierung ist eine MontiCore-Grammatik für die Zielsprache, auf deren Basis ein entsprechender `ConcreteRefactorizer` im verwendeten `GenerationTool` zur Verfügung gestellt werden kann. Eine Symboltabelle muss für die Zielsprache nicht notwendigerweise existieren, kann jedoch die Implementierung von Refactorings erleichtern. Das `UMLPTool` enthält bereits einen `ConcreteRefactorizer` für Java bzw. Java/P einschließlich einer Symboltabelle. In diesem Fall kann einfach das `UMLPTool` selbst wiederverwendet werden, da es ebenfalls alle notwendigen Sprachbestandteile enthält (siehe Abbildung 7.21). Für Refactorings auf diesen Zielsprachen genügt es somit, entsprechende `RefactoringVisitor`en dem Tool hinzuzufügen und bei Bedarf aufzurufen. Darüber hinaus kann das `UMLPTool` für andere Zielsprachen über die Schnittstellen des `GenerationTool`s jederzeit erweitert werden.

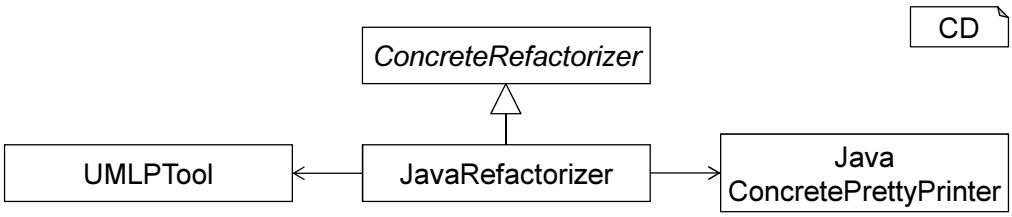


Abbildung 7.21: Umsetzung des Java-Refactorings

7.6 Nutzung des UML/P-Frameworks

Unternehmen und häufig selbst einzelne Projekte verwenden unterschiedlichste Werkzeuge für die Softwareentwicklung. Für eine Sprachfamilie wie die UML/P, die sich in allen Phasen der Softwareentwicklung einsetzen lässt, ist es daher von Vorteil, wenn sich das dazugehörige Framework flexibel in eine bestehende Werkzeuginfrastruktur integrieren und an spezifische Anforderungen anpassen lässt. Weitere Anpassungen können für eine Ausrichtung auf bestimmte Domänen oder Technologien erforderlich sein, die sich sowohl auf Umfang und Zusammensetzung der Sprachen selbst, die Validierung der Modelle oder die Codegenerierung auswirken können. Insgesamt ergeben sich daraus folgende Anwendungsszenarien des UML/P-Frameworks:

- **Modellierung und Validierung**: Die Modellierung von Softwaresystemen ist der primäre Anwendungsfall des UML/P-Frameworks. Durch die automatisierte Überprüfung von Syntax und Kontextbedingungen wird der Modellierer bei der Entwicklung konsistenter Modelle und der Vermeidung von Fehlern unterstützt (siehe Kapitel 4).
- **Generierung**: Der wohl wichtigste Aspekt von Modellen ist deren Fähigkeit zur Abstraktion. Die fehlenden Details für eine konkrete Implementierung des modellierten Systems können durch Codegeneratoren ergänzt werden (siehe Kapitel 5). Aufgrund des universellen Anspruchs der UML/P als Werkzeug zur Modellierung von Softwaresystemen und der Vielzahl heutiger sowie zukünftiger Technologien ist es jedoch unmöglich, Generatoren zu implementieren, die allen potentiellen Anforderungen genügen. Neben der Verwendung ist somit auch die Entwicklung, Anpassung und Erweiterung von Generatoren ein möglicher Bestandteil der Softwareentwicklung mit der UML/P.
- **Sprachanpassung**: Die Anpassung der Sprachen ist in der UML/P durch die Definition von Profilen vorgesehen. Für deren Umsetzung erlaubt das Framework die Verwendung beliebiger Stereotypen sowie die Erweiterung und Modifikation von Kontextbedingungen. Durch die Flexibilität des Generierungsframeworks können ebenfalls auf Profile zugeschnittene Generatoren entwickelt werden (siehe auch Abschnitt 6.2). Darüber hinaus ermöglicht es die Strukturierung der Sprachdefinitionen und eine entsprechend darauf angepasste konsequente Trennung von Kontextbedingungen, Symboltabellen und Generatoren nach Sprachen, die Zusammensetzung der Sprachen zu verändern, so dass neue Formen der UML/P entstehen oder weitere Sprachen ergänzt werden können (siehe Abschnitt 7.1 und 7.2).
- **Werkzeugintegration**: Das auf Java basierende UML/P-Framework wurde nicht an ein spezifisches Werkzeug gekoppelt. So ist für die Verifikation von Modellen sowie für die Ausführung von Generatoren und Refactorings lediglich eine Java-Laufzeitumgebung erforderlich. Bei den Modellen und Generatoren handelt es sich um einfache Textdateien, die in beliebigen Editoren entwickelt werden können. Nur für die Implementierung von Kontextbedingungen, Kalkulatoren und Refactorings wird zusätzlich ein Java-Compiler

benötigt. Diese geringen Anforderungen machen die UML/P und das Framework in vielen bestehenden Werkzeuglandschaften einsetzbar.

Die wesentlichen Konzepte für die obigen Anwendungsszenarien wurden bereits in dieser Arbeit ausführlich diskutiert. Ergänzend dazu werden nun im Folgenden die Ausführung und Beispiele für die Integration des UML/P-Frameworks behandelt.

Die zentrale Schnittstelle für die Ausführung ist das `UMLPTool`, dessen Steuerung über Parameter erfolgt. Abbildung 7.22 zeigt dies am Beispiel eines Aufrufs auf der Kommandozeile. Durch die Angabe des Classpaths über den Parameter `-cp` erhält Java Zugriff auf die benötigten Komponenten des UML/P-Frameworks (siehe Abschnitt 7.1). In dem Beispiel wurden diese im Verzeichnis `lib` abgelegt. Danach folgen das aufzurufende `UMLPTool` und dessen Parameter, wobei als erstes die zu verarbeitenden Modelle angegeben werden. Dabei kann es sich um einzelne Modelle oder Verzeichnisse handeln. Letzteres schließt Unterverzeichnisse mit ein und verarbeitet alle dem `UMLPTool` bekannten Modellarten. Alle weiteren Details und die auszuführenden Verarbeitungsschritte werden über die in Tabelle 7.23 zusammengefassten Parameter konfiguriert. Die Verarbeitungsschritte lassen sich zusätzlich auf bestimmte Modellarten durch die Angabe der entsprechenden Dateiendung einschränken, wobei `ALL` für keine Einschränkung steht (siehe Tabelle 3.37 in Abschnitt 3.8).

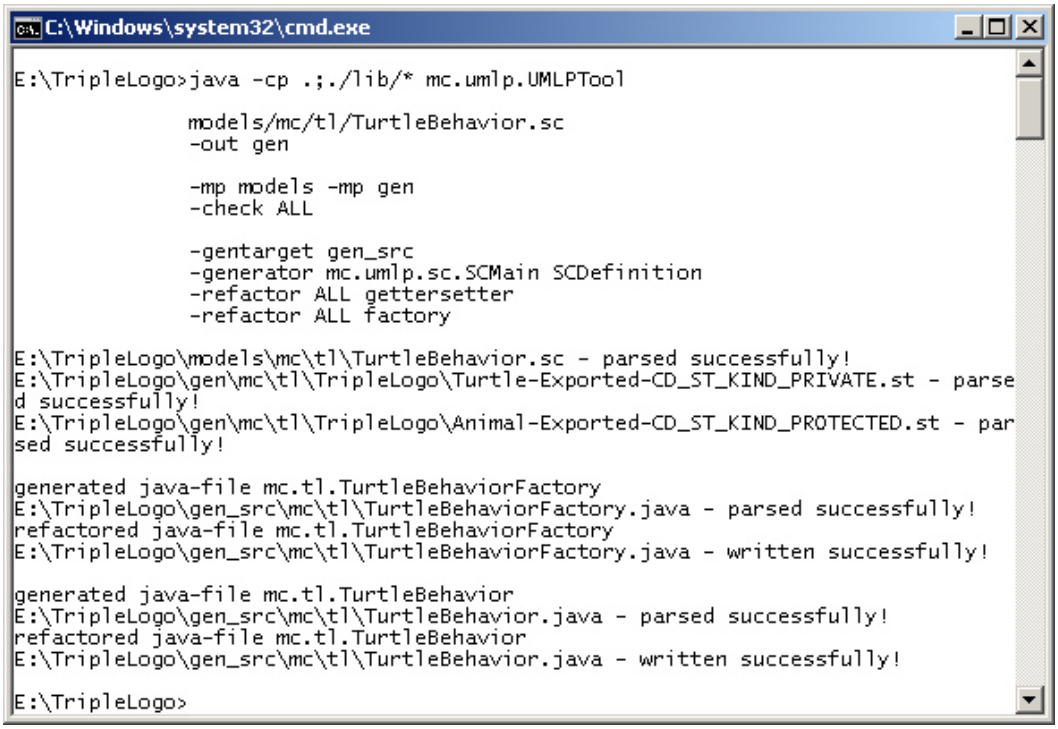


Abbildung 7.22: Generatoraufruf auf der Kommandozeile (Screenshot)

Bei der Ausführung zeigt MontiCore die verarbeiteten Modelle und die geladenen serialisierten Symboltabelleneinträge an. Generierungs- und Refactoringschritte werden ebenfalls protokolliert.

1. Modellverarbeitung	
`-out` *Dateiordner*	Ausgabeverzeichnis für generierte Dateien der Modellverarbeitung.
`-mp` *Dateiordner*	Verzeichnis, aus dem referenzierte Modelle, Modellelemente oder Symboltabelleneinträgen geladen werden können ("mp" steht für *Modelpath*).
`-analysis` *Modellart* `parse`	Ausführung des Workflows zum Einlesen von Modellen.
2. Symboltabelle und Kontextbedingungen	
`-check` *Modellart Konfiguration*	Überprüfung der Modelle anhand der Kontextbedingungen. Dies schließt die notwendigen Workflows für Modellverarbeitung, Symboltabellenaufbau und Kontextanalyse mit ein. Der letzte Parameter ist optional und dient der Auswahl einer im `UMLPTool` registrierten Konfiguration der Bedingungen. Ansonsten wird die Standardkonfiguration verwendet.
`-analysis` *Modellart* `init`	Ausführung des Workflows zur Initialisierung des Symboltabellenaufbaus.
`-analysis` *Modellart* `createExported`	Ausführung des Workflows zur Erstellung der Symboltabelle. Die Einträge werden serialisiert und in dem unter `-out` angegebenen Verzeichnis abgelegt.
`-synthesis` *Modellart* `prepareCheck`	Ausführung des Workflows zur Initialisierung der Kontextanalyse. Dabei wird die Symboltabelle des aktuellen Modells mit den Einträgen importierter Elemente verbunden.
`-synthesis` *Modellart* `check`	Ausführung des Workflows zur Kontextanalyse.
3. Codegenerierung	
`-gentarget` *Dateiordner*	Ausgabeverzeichnis für die Codegenerierung. Ansonsten wird das unter `-out` angegebene Verzeichnis verwendet.
`-generator` *Templatename Regelname*	Ausführung eines Generators ausgehend vom angegebenen Template und den Elementen der abstrakten Syntax, die dem Regelnamen entsprechen. Das Template muss sich über den Classpath des Java-Aufrufs laden lassen.
`-refactor` *Modellart Refactoringname*	Ausführung des Refactorings, das unter dem angegebenen Namen im `UMLPTool` registriert ist.

Tabelle 7.23: Parameter des `UMLPTools`

Diese Ausgaben und eventuell auftretende Fehler werden im MontiCore-Framework durch den `ErrorHandler` verwaltet, der einen zentralen Zugriff und eine Konfiguration dieser Meldungen erlaubt [Kra10].

Neben dieser einfachen Verwendung kann das `UMLPTool` über dessen Schnittstellen ebenfalls gezielt an spezielle Anforderungen angepasst und erweitert werden. So können Kontextbedingungen hinzugefügt und konfiguriert, Refactorings registriert oder die Zusammensetzung von Sprachen und Symboltabellen geändert werden. Letzteres lässt sich den Erläuterungen in Abschnitt 7.2 entnehmen. Vertiefende Details finden sich in [Kra10, Völ11].

Quellcode 7.24 zeigt eine solche Anpassung anhand einer Subklasse des `UMLPTool`s. In diesem Beispiel wird in Zeile 7-8 die Kontextbedingung aus Quellcode 7.25 im Werkzeug registriert. Diese überprüft, ob Mehrfachvererbung in einem Klassendiagramm verwendet wurde. Eine solche Bedingung könnte etwa im Rahmen der Codegenerierung gefordert werden, wenn der Generator Mehrfachvererbung im Zielsystem nicht umsetzen kann. Um in frühen Phasen der Entwicklung Mehrfachvererbung noch zuzulassen, wurde in Zeile 11 eine neue Konfiguration der Kontextbedingungen basierend auf der Standardkonfiguration des `UMLPTool`s erstellt und in Zeile 12 um die neue Bedingung erweitert. Der dabei vergebenen Name “codegen” kann über den Parameter `-check` beim Aufruf des Werkzeugs angegeben werden, um diese Konfiguration gezielt erst im Rahmen der Codegenerierung einzusetzen (vgl. Tabelle 7.23).

Java

```
1  public class MyTool extends UMLPTool {
2
3    public MyTool(String[] args) {
4      super(args);
5
6      // Register Context Condition
7      languages.addContextCondition(
8                new NoMultipleInheritanceChecker("No_Multiple_Inheritance"));
9
10     // Configure Context Condition for model processing
11     addCocoConfiguration("codegen", getCocoConfiguration());
12     getCocoConfiguration("codegen").put("No_Multiple_Inheritance", Type.ERROR);
13
14     // Register Refactoring
15     addRefactoring("privatemethods", new PrivateMethodRefactoringVisitor(1));
16   }
17
18   public static void main(String[] args) {
19     MyTool tool = new MyTool(args);
20     tool.init();
21     tool.run();
22   }
23
24 }
```

Quellcode 7.24: Customizing des `UMLPTool`s als Java-Subklasse

Das Beispiel zeigt ebenfalls, wie die Eigenschaft `checkedProperty` zur Konfiguration von Kontextbedingungen eingesetzt wird (vgl. Abbildung 7.13). In diesem Fall wird die Eigenschaft über den Konstruktor gesetzt. Bei umfangreicheren Sätzen von Kontextbedingungen bietet sich

```java
public class NoMultipleInheritanceChecker
extends ContextCondition implements ICDTypeChecker {

  public NoMultipleInheritanceChecker(String checkedProperty) {
    super(checkedProperty);
  }

  @Override
  public void check(ASTCDType cdType, CDTypeEntry typeEntry) {
    if (typeEntry.getSuperTypes().size() > 1) {
      addReport("Multiple inheritance used in Type " + typeEntry.getName(),
                cdType.get_SourcePositionStart());
    }
  }

}
```

Quellcode 7.25: Java-Implementierung einer Generator-spezifischen Kontextbedingung

stattdessen die Verwendung von Konstanten an (vgl. Quellcode 7.14). Schließlich wird in Zeile 15 noch das Refactoring aus Quellcode 7.20 registriert. Der vergebene Name “privatemethods” dient dem Aufruf des Refactorings über den Parameter `-refactor`.

Wie das Beispiel definiert auch das `UMLPTool` selbst nur die Zusammensetzung der einzelnen Bestandteile der Infrastruktur. Sämtliche Schnittstellen und allgemeine Funktionalitäten werden hingegen über das `GenerationTool` geerbt und stehen somit allen mit MontiCore entwickelten Sprachen zur Verfügung (siehe Abbildung 7.7). Die Subklassenbildung ist daher nur eine Möglichkeit für die Anpassung der Werkzeuginfrastruktur, die ebenfalls über eine Instanz des `UMLPTool`s entsprechend durchführbar ist. Dabei muss die Anpassung jedoch vor dem `init()`-Aufruf erfolgen, der anhand der einzelnen Bestandteile die vollständige Infrastruktur aufbaut (vgl. Zeile 19-21 in Quellcode 7.24). Auf die gleiche Weise kann auch das `GenerationTool` als Instanz oder Superklasse genutzt werden, um eine vollständig neue Zusammensetzung des Werkzeugs aufzubauen.

Ein allein auf Java und textuellen Sprachen basierendes Framework wie das der UML/P bietet den Vorteil, dass es sich aufgrund der geringen Anforderungen eher in andere Werkzeuge integrieren lässt, als wenn es bereits an bestimmte Editoren oder Programmumgebungen gekoppelt wäre. So lassen sich Modelle ohne weiteres über Versionsmanagementsysteme wie CVS oder SVN verwalten. Aufgrund der textuellen Darstellung können Vergleiche oder das Anzeigen und Lösen von Versionskonflikten direkt auf den Modellen selbst ausgeführt werden. Dasselbe gilt für andere Artefakte wie Kontextbedingungen oder Generatoren.

Die Aufrufe für Kontextanalyse und Codegenerierung sind überall dort integrierbar, wo eine Java-Laufzeitumgebung zur Verfügung steht. So werden vor allem bei der Entwicklung größerer Softwareprojekte Abhängigkeiten mit Hilfe von Build-Systemen wie Ant, Maven oder Make verwaltet, die sich auch für die automatisierte Qualitätssicherung einsetzen lassen. Abbildung 7.26 zeigt dies am Beispiel von Ant, das die Steuerung komplexer Abläufe über eine XML-Datei

erlaubt [Hol05]. Diese wird auch als Build-Datei bezeichnet und ist in einzelne sogenannte Targets (engl. für "Ziele") unterteilt, die gezielt aufgerufen werden können.

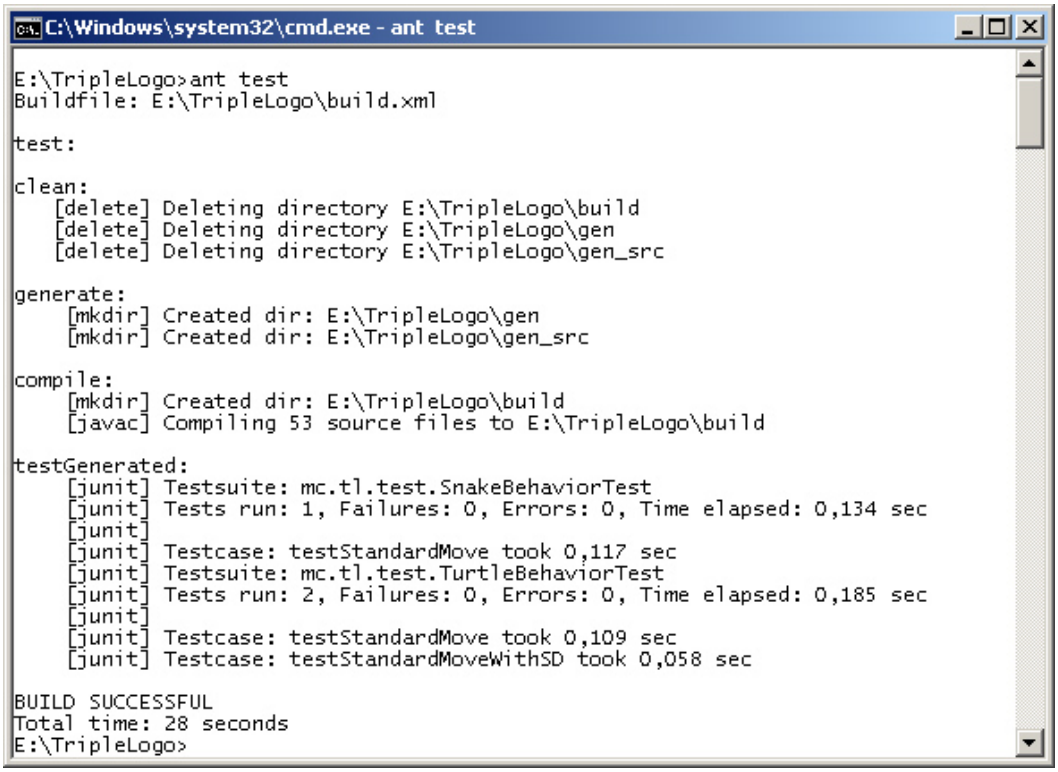


Abbildung 7.26: Generierung und Testausführung mit Ant (Screenshot)

Im Beispiel wird das Target "test" aufgerufen, welches wiederum mehrere weitere Targets ausführt. Diese löschen zuerst alle bereits generierten Dateien, starten die Kontextanalyse und Codegenerierung auf allen Modellen, kompilieren die generierten Dateien und führen die ebenfalls generierten Testfälle aus. Auf diese Weise wird sowohl die Korrektheit der Modelle in Bezug auf die Kontextbedingungen, die Kompilier- und Ausführbarkeit des generierten Codes sowie die Korrektheit des Systems gegenüber den modellierten Testfällen mit einem einzigen Aufruf überprüft.

Neben der Verwaltung von Artefakten und der Automatisierung von Generierungs- und Testabläufen sind insbesondere Werkzeuge sinnvoll, die den Modellierer bei der Arbeit unterstützen. Abbildung 7.27 zeigt dies am Beispiel der Integration des UML/P-Frameworks in die Entwicklungsumgebung Eclipse. Die entsprechenden Plugins wurden ebenfalls im Rahmen dieser Arbeit entwickelt und stehen als separate Komponenten im UML/P-Framework zur Verfügung (siehe Abschnitt 7.1). Diese bieten für jede Sprache einen Editor mit Syntaxhervorhebung, Autovervollständigung, Modellübersicht (Outline) sowie dem Ein- und Ausblenden von Details (Folding). Darüber hinaus werden Fehler und verletzte Kontextbedingungen direkt in den Modellen als Fehlermarker und Tooltip und in der Problemansicht von Eclipse angezeigt. In Abbildung 7.28

wurde etwa als Ziel einer Transition ein Name angegeben, für den kein Zustand definiert wurde. Abbildung 7.29 zeigt dieselbe Fehlermeldung bei der Ausführung auf der Kommandozeile.

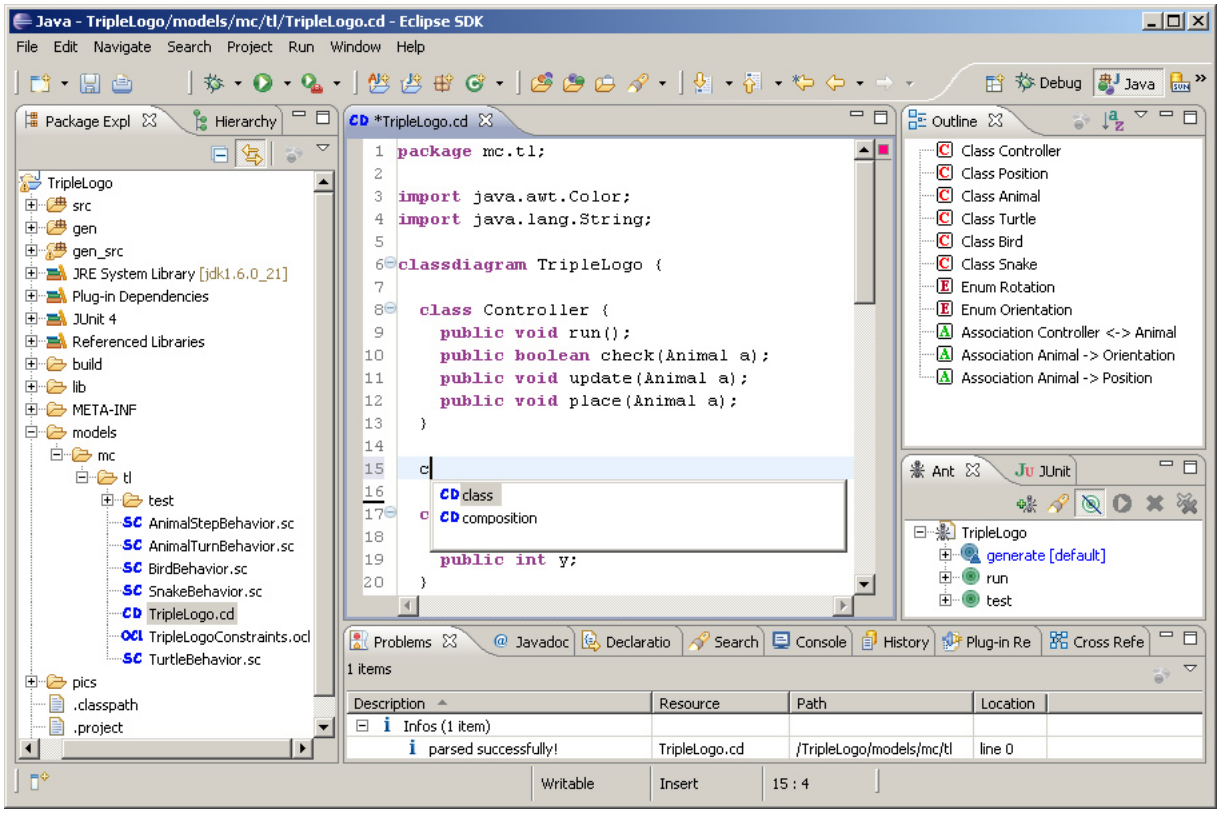


Abbildung 7.27: UML/P-Plugin für Eclipse (Screenshot)

Da Eclipse Java-Aufrufe erlaubt, ist die Ausführung von Generatoren ebenfalls möglich. Darüber hinaus ist Ant in Eclipse integriert, so dass alternativ die angesprochene Build-Datei verwendet werden kann, wie die Abbildungen 7.27 und 7.28 zeigen. Zukünftig sind noch weitere Plugins denkbar. Beispielsweise könnte eine graphische Ansicht der Modelle und entsprechende Editoren die textuelle Darstellung ergänzen. Doch bereits mit den jetzigen Plugins bietet Eclipse eine einheitliche Umgebung für die modellgetriebene Entwicklung von Softwaresystemen mit der UML/P. So können Modelle, Generatoren und Code einschließlich der Qualitätssicherung durch modellbasierte und automatisierte Tests parallel entwickelt und ausgeführt werden, ohne dass dafür ein Wechsel zwischen unterschiedlichen Werkzeugen notwendig ist. Dadurch werden Modelle und Generatoren zum integralen Bestandteil von Softwareentwicklungsprojekten und bilden gemeinsam mit Laufzeitsystemen bzw. zusätzlichem Code den eigentlichen Quellcode eines Zielsystems.

In diesem Kapitel konnten nur einige der Möglichkeiten des UML/P-Frameworks dargestellt werden. Dessen Aufteilung in einzelne Komponenten sowie Aufbau und Schnittstellen der Architektur erlauben flexible Erweiterungen und umfangreiche Modifikationen der UML/P.

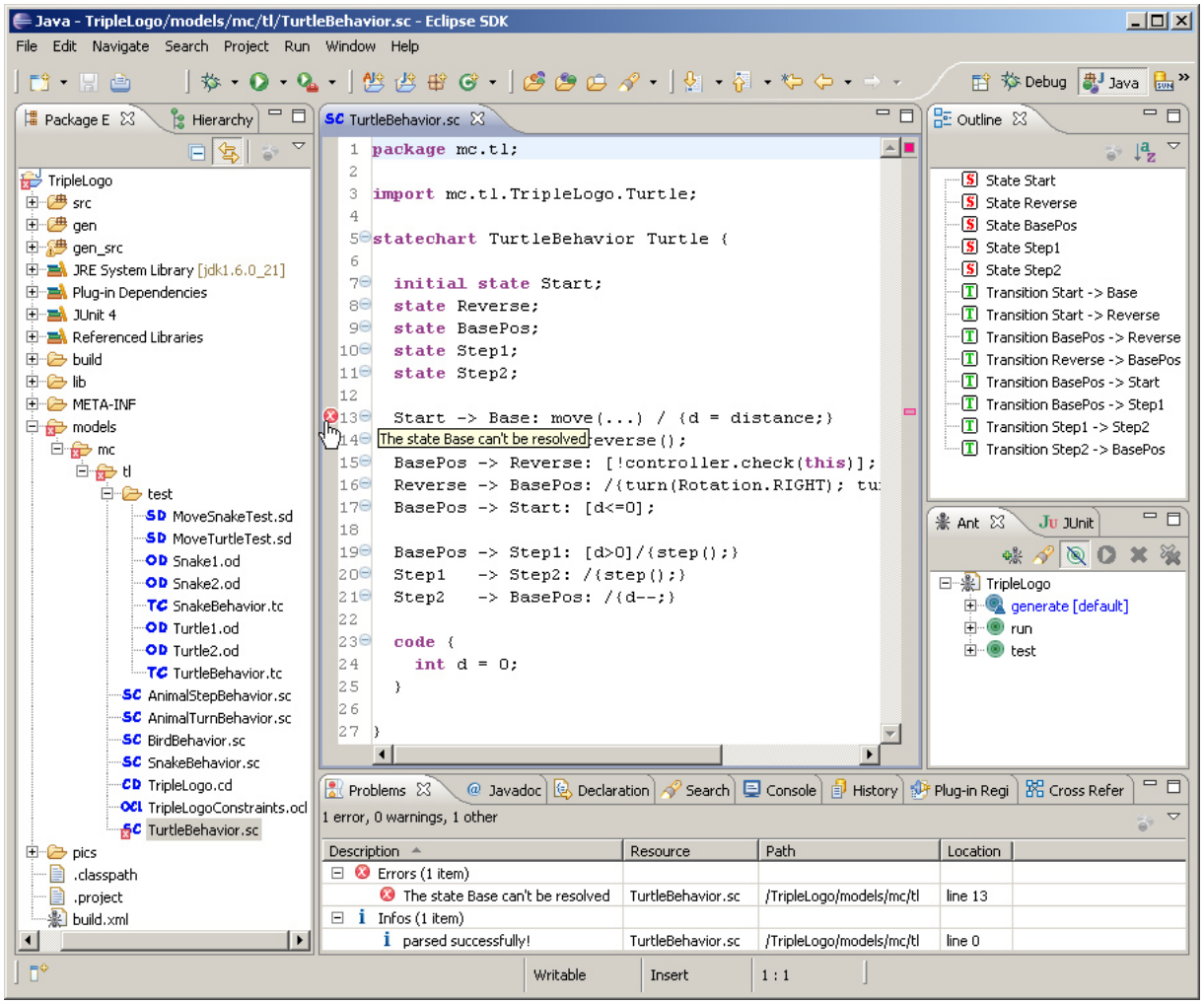


Abbildung 7.28: UML/P-Plugin: Integration der Kontextanalyse (Screenshot)

Abbildung 7.29: Verletzte Kontextbedingung auf der Kommandozeile (Screenshot)

So können Sprachen ausgetauscht oder hinzugefügt, in ihrer Bedeutung oder Interpretation verändert, die Kontextanalyse an einzelne Entwicklungsphasen angepasst sowie Generatoren für unterschiedlichste Zielplattformen entwickelt werden. Diese Möglichkeiten können selbst zu neuen Formen der UML/P führen, ohne dass dafür eine Änderung der Framework-Komponenten erforderlich ist. Die geringen Anforderungen des Frameworks erlauben es darüber hinaus, dieses in unterschiedlichste Werkzeuge zu integrieren. Auf diese Weise ist eine Infrastruktur für die UML/P entstanden, die sich agil und flexibel an Anforderungen von heutigen oder auch zukünftigen Werkzeugen, Technologien, Domänen oder Plattformen anpassen lässt.

Kapitel 8

Diskussion und verwandte Arbeiten

Auf eine ganze Reihe im Kontext dieser Arbeit relevanten und verwandten Arbeiten wurde bereits in den vorangegangenen Kapiteln eingegangen. Im Folgenden werden nun die Besonderheiten der UML/P und des Frameworks im Vergleich zu anderen Arbeiten diskutiert. In Abschnitt 8.1 werden zunächst die Sprache und die Modellierungskonzepte der UML/P behandelt. Da diese im Wesentlichen auf [Rum04a, Rum04b] basieren, liegt der Fokus hier insbesondere auf den Änderungen und Erweiterungen, die im Zuge der Umsetzung in eine textuelle Syntax entstanden sind. Darauf folgt eine Betrachtung der Definition und Umsetzung der Kontextbedingungen in Abschnitt 8.2 sowie der Implementierung der Codegenerierung in Abschnitt 8.3. Abschließend werden in Abschnitt 8.4 die Konzepte dieser Arbeit mit anderen modellgetriebenen Ansätzen und Werkzeugen verglichen.

8.1 Sprache und Modellierungskonzepte

Die Sprache und Modellierungskonzepte der UML/P wurden bereits in [Rum04a, Rum04b] ausführlich behandelt. In [Rum04a] finden sich darüber hinaus Konzepte, wie sich die UML/P für eine agile modellbasierte Vorgehensweise nutzen und entsprechend in etablierten Vorgehensmodellen wie XP und dem Rational Unified Process integrieren lässt. Dabei werden initial sehr abstrakte und unterspezifizierte Modelle entworfen und schließlich durch Verfeinerungen im Laufe des Entwicklungsprozesses zu einem Teil der Lösung. Dieses agile Vorgehen ist nun mit der im Rahmen dieser Arbeit entwickelten Werkzeuginfrastruktur erstmals praktisch für die modellgetriebene Entwicklung von Softwaresystemen einsetzbar. Dabei wird eine agile Entwicklung insbesondere durch die folgenden Eigenschaften der Werkzeuginfrastruktur unterstützt:

- Die Kontextbedingungen sind hinsichtlich Auswahl und Fehlergrad frei konfigurier- und erweiterbar. Dadurch ist eine Anpassung an die unterschiedlichen Anforderungen von Entwicklungsphasen, Sprachprofilen, Unternehmen oder Projekten flexibel möglich. Unterschiedliche Konfigurationen lassen sich speichern und bedarfsorientiert auswählen.
- Modelle und Generatoren können mit beliebigen Texteditoren erstellt und bearbeitet werden. Für die Auswahl und Ausführung von Kontextbedingungen und Generatoren genügt eine

Java-Laufzeitumgebung. Die Implementierung von Kontextbedingungen, Kalkulatoren und Refactorings erfolgt in Java, so dass in diesen Fällen zusätzlich ein Java-Compiler notwendig ist. Insgesamt ist damit ein flexibler Einsatz der UML/P in unterschiedlichsten Werkzeugumgebungen möglich.

- Die modulare Umsetzung von Kontextbedingungen und Codegenerierung ermöglicht eine inkrementelle Ausführung. So können nur geänderte Modelle verarbeitet oder eine begrenzte Auswahl an Kontextbedingungen und Generatoren angewendet werden.
- Unterstützung für die Entwicklung im Team durch konventionelle Versionsmanagement- und Buildsysteme.
- Stereotypen können beliebige Werte enthalten und bei Bedarf durch Kontextbedingungen und Generatoren ergänzt werden. Auf diese Weise entsteht eine leichtgewichtige und agile Anpassung der Sprache durch Profile.

Die Unterschiede der textuellen und graphischen Notation wurden bereits in Abschnitt 3.9 behandelt. Hervorzuheben ist dabei die Einordnung der Modelle in eine hierarchische Paketstruktur, die eine eindeutige Referenzierung von Modellen und Modellelementen über Import-Anweisungen und voll-qualifizierte Namen ermöglicht. Gleichzeitig erlaubt dies dem Modellierer das Auffinden von Modellen im Dateisystem unabhängig von einer spezifischen Werkzeuginfrastruktur. Diese Form der Verwaltung und Referenzierung von Artefakten wurde Programmiersprachen wie Java oder C++ entlehnt und hat sich dort seit langem bewährt. In graphischen Modellierungswerkzeugen für die UML werden Modelle und Beziehungen hingegen werkzeugspezifisch verwaltet. So speichern etwa Enterprise Architect [EA] oder Rhapsody [RR] Modelle und Modellelemente in einer Datenbank, die ebenfalls die Verwaltung von Referenzen übernimmt. Aus der Datenbank können Elemente über Menüstrukturen ausgewählt und zu neuen Sichten zusammengestellt werden. Diese Werkzeuge basieren somit auf einem großen Gesamtmodell im Hintergrund, das immer vollständig zur Verfügung stehen muss. Nur einzelne Modelle zu laden, wie bei einer dateibasierten Verwaltung üblich, ist bei diesem Ansatz nicht möglich. Andere Werkzeuge wie Together [Tog] speichern hingegen die Modelle als einzelne XMI-Dateien [OMG07]. In diesen werden Referenzen auf ein bestimmtes Modellelement anhand des Dateinamens und einer eindeutigen Element-ID als separate Attribute angegeben.

Eine der Besonderheiten der UML/P ist die Verwendung von Java als Aktionssprache in den Modellen. Die UML-Spezifikation beschreibt nur die Semantik von Aktionen (Action Semantics [OMG10d]) anhand des Metamodells, gibt jedoch keine konkrete Syntax vor. Ein wichtiger Aspekt der vorliegenden Arbeit und des UML/P-Frameworks ist, dass Java nicht nur eingebettet, sondern gleichzeitig auf die Modellierungsebene angehoben wurde. Dieses Java/P wird vom Framework gegenüber den Modellen validiert und kann wie diese von Generatoren verarbeitet werden. Dadurch ist der Modellierer in der Lage, Fehler bereits im Modell zu erkennen und den eingebetteten Code ohne Kenntnis des generierten Systems zu erstellen, so dass die Plattformunabhängigkeit der Modelle erhalten bleibt. Auf diese Weise wird Java/P zu einer

abstrakten Programmiersprache, ohne dass die Syntax gegenüber Java dafür geändert werden musste.

Dieses Konzept der Anhebung einer Programmiersprache lässt sich auch auf andere Sprachen anwenden. Deshalb wurde das UML/P-Framework so konzipiert, dass die Aktionssprache relativ leicht ersetzt werden kann. Für eine vollständige Integration einer neuen Aktionssprache muss dafür eine entsprechende MontiCore-Grammatik sowie Symboltabelle implementiert und über einen Adapteransatz eingebunden werden (siehe Kapitel 7).

Genauso leicht ersetz- und erweiterbar sind die anderen Sprachen der UML/P. Die Symboltabelle dient dabei als eine Art Schnittstellendefinition der jeweiligen Sprachen, die eine kompositionale Zusammenstellung von Sprachfamilien ermöglicht [Völ11]. Da die Werkzeuginfrastruktur der UML/P durchgängig kompositional umgesetzt wurde, lassen sich sämtliche Aspekte einer Sprache wie Sprachverarbeitung, Kontextbedingungen und Codegeneratoren auch in anderen Sprachkombinationen wiederverwenden.

Bereits in [MS91] wurde mit der sogenannten Shlaer-Mellor-Methode ein Ansatz beschrieben, der mehrere graphische Modellierungssprachen mit einer textuellen Aktionssprache für die Beschreibung von Struktur und Verhalten von Softwaresystemen verbindet. Diese Idee wurde in [MB02] basierend auf der Action Semantics auf eine Teilmenge der UML übertragen, was zu dem Begriff “Executable UML” (xUML) geführt hat. Der Ansatz ist eng mit der UML/P verwandt, bietet jedoch im Gegensatz dazu keine modellbasierten Tests oder OCL-Bedingungen. Darüber hinaus wird keine existierende Sprache als Aktionssprache eingesetzt, sondern eine eigene “Action Specification Language” (ASL) definiert.

Erst 2011 hat die OMG mit “Foundational UML” (fUML) einen Standard veröffentlicht, der eine präzise Semantik für ausführbare UML-Modelle beschreibt [OMG11b]. Dieser basiert ebenfalls auf einer Teilmenge der UML in Verbindung mit einer Aktionssprache. Auf dieser Basis können virtuelle Maschinen umgesetzt werden, die eine Simulation und Analyse des Verhaltens von Modellen erlauben [CD08a, CD08b]. Die Spezifikation einer textuellen Syntax der Aktionssprache ist noch nicht vollständig abgeschlossen, aber bereits in einer ersten Beta-Version verfügbar [OMG10a]. Während die OMG hierfür eine Anlehnung an Sprachen wie Java, C++ oder C# vorsieht, schlagen andere Arbeiten eine Erweiterung der OCL als Aktionssprache vor [JZM07]. Die UML/P und diese Arbeit zeigen jedoch, dass auch bestehende Sprachen in die UML integriert werden können und dabei sogar austauschbar sind.

In [HJSW10] wurde eine EMF-basierte Syntax und statische Semantik von Java entwickelt. Dadurch können EMF-basierte Werkzeuge auf Java angewendet werden. Im Gegensatz zur Java-Realisierung des UML/P-Frameworks fehlt jedoch ein Konzept für eine Komposition von Sprachen sowie eine Symboltabelle.

Eine textuelle Notation für die UML wird auch in anderen Ansätzen verwendet. So bietet die OMG mit der XMI-Spezifikation ein textuelles Dateiformat für die Speicherung von UML-Modellen [OMG07]. Das auf XML basierende Format ist jedoch wenig kompakt und enthält beim Export aus graphischen Modellen oft ebenfalls Positionsangaben zu den Modellelementen. Für die Modellierung lässt es sich dadurch nicht sinnvoll einsetzen. Ein kompakteres Format

bietet HUTN (Human-Usable Textual Notation, [OMG04]), bei der eine textuelle Notation aus einem MOF-basierten Metamodell abgeleitet wird. Bis auf eine unvollständige Implementierung in [RPKP08a] existieren bisher jedoch kaum Umsetzungen dieses OMG-Standards. Darüber hinaus besteht die Notation aufgrund der automatischen Ableitung aus dem Metamodell in großen Teilen aus Name-Wert-Paaren und kann somit hinsichtlich Lesbarkeit und Kompaktheit mit einem manuellen Sprachdesign nicht konkurrieren.

Andere textuelle UML-Notationen dienen als Unterstützung, um graphische Diagramme zu entwerfen. Dazu gehören Notationen wie yUML [yUML], MetaUML [MU] und UMLGraph [UG], die auf eine Erstellung einzelner UML-Modelle als Bilddateien ausgerichtet sind und dafür zusätzliche Layout-Anweisungen enthalten. Dagegen bieten TextUML [TU], USE [GBR07] und Umple [For10] eine zur UML/P recht ähnliche Notation, die sich jedoch zurzeit auf Klassendiagramme beschränkt (siehe auch Abschnitt 8.4). Für Umple ist darüber hinaus eine Erweiterung für Statecharts geplant [Bad10].

8.2 Kontextbedingungen

Die Kontextbedingungen der UML/P werden in [Rum04a, Rum04b] nur informell und unvollständig behandelt. Darüber hinaus ergeben sich aufgrund der in Abschnitt 3.9 beschriebenen Erweiterungen der textuellen gegenüber der graphischen Syntax in dieser Arbeit zusätzliche Bedingungen. In der UML-Spezifikation der OMG werden die Kontextbedingungen jeweils im Rahmen der Elementbeschreibungen aufgeführt und verteilen sich dadurch über mehr als 1000 Seiten [OMG10d, OMG10c, OMG10b]. Dies zu überblicken, ist keine leichte Aufgabe [FGDS06], was sich u.a. daran zeigt, dass bisher kein Werkzeug die UML-Spezifikation vollständig erfüllt [EES09]. Aus diesem Grund wurden die Kontextbedingungen in dieser Arbeit in einem eigenen Kapitel zusammengefasst und mit einer ID für eine eindeutige Referenzierung versehen.

Im UML/P-Framework werden Kontextbedingungen in Java auf Basis der abstrakten Syntax und der Symboltabelle programmiert. Das Besondere an diesem Ansatz ist, dass dadurch die Bedingungen einer Sprache jeweils in der Sprachkomponente gekapselt werden können und unabhängig von einer konkreten Sprachzusammenstellung wiederverwendbar sind. Durch den Adapteransatz der Symboltabelle gilt dies selbst für sprachübergreifende Bedingungen (siehe Abschnitt 7.3). Darüber hinaus sind die Kontextbedingungen und deren Fehlergrade frei konfigurier- und erweiterbar, so dass sie an spezifische Anforderungen von Sprachprofilen, Domänen oder Projekten angepasst werden können.

In den meisten UML-Werkzeugen sind Kontextbedingungen fest integriert, ohne dass eine Anpassung vorgesehen ist. Eine Ausnahme ist Rhapsody [RR]. Hier können Bedingungen für Klassendiagramme und Statecharts über ein Formular zusammengestellt und vor der Ausführung der Codegenerierung überprüft werden. Andere Anwendungsszenarien oder Diagramme sowie die Konfiguration der Fehlergrade werden zurzeit nicht unterstützt. Darüber hinaus sind Bedingungen nur innerhalb von Rhapsody konfigurier- und ausführbar. Zusätzliche Kontextbedingungen

können in Java, C++ oder Visual Basic gegen die abstrakte Syntax programmiert und als Plugin hinzugefügt werden [IBM09].

In anderen Ansätzen werden Kontextbedingungen in OCL auf dem Metamodell spezifiziert. So erlaubt Together [Tog] die Ergänzung von Profilen um OCL-Bedingungen basierend auf dem EMF-Metamodell Ecore. Eine Unterscheidung von Fehlergraden oder die Anpassung von allgemeinen Kontextbedingungen außerhalb von Profilen ist jedoch nicht vorgesehen [Bor11]. Dadurch fehlt auch eine Berücksichtigung von unterschiedlichen Abstraktionsgraden der Modelle im Laufe des Entwicklungsprozesses.

Obwohl es sich bei der OCL um eine auf Bedingungen spezialisierte Sprache handelt, bringt sie im Hinblick auf Kontextbedingungen einige Einschränkungen mit sich. So ist es u.a. nicht möglich, Fehlergrade bei verletzten Bedingungen zu unterscheiden oder Fehlermeldungen zu definieren [KPP09]. Auch für komplexe Berechnungen bietet OCL im Vergleich zu regulären Programmiersprachen nur begrenzte Möglichkeiten. Dies hat dazu geführt, dass an OCL angelehnte Sprachen für Kontextbedingungen um zusätzliche Konstrukte erweitert wurden. Dazu gehören die Epsilon Validation Language (EVL, [KPP09]) aus der Epsilon Sprachfamilie [KPP06] und die Sprache Check von openArchitectureWare [oAW]. Mittlerweile bieten beide Sprachen die Möglichkeit, alternativ Java-Ausdrücke für Kontextbedingungen zu verwenden [KPP08b, EFH$^+$08]. In EVL können darüber hinaus Transformationen für eine mögliche Behebung verletzter Bedingungen angegeben werden. Eine Zusammenstellung von unterschiedlichen Konfigurationen einschließlich der Anpassung von Fehlergraden oder die Kompositionalität der Bedingungen in Bezug auf verschiedene Sprachkombinationen unterstützen Check und EVL hingegen nicht.

8.3 Generatoren und Templatesprachen

Trotz der umfassenden Modellierung von Struktur und Verhalten mit der UML/P spielt die Codegenerierung eine wichtige Rolle. Anders als Compileransätze, die im Grunde eine feste Codegenerierung vorgeben, soll diese in der UML/P jederzeit an den jeweiligen Projektkontext anpassbar sein. Auf diese Weise kann die UML/P für die Softwareentwicklung unabhängig von einer konkreten Zieltechnologie oder Domäne eingesetzt werden. Insbesondere wird dadurch auch die Abstraktion der Modelle im Hinblick auf zukünftige Technologien erhalten.

Da die Generatoren die Modelle in einem Softwareentwicklungsprojekt somit ergänzen, müssen diese möglichst flexibel und einfach anpassbar sein. Darüber hinaus muss deren Entwicklung den Anspruch der UML/P hinsichtlich Agilität erfüllen. Diese Aspekte werden wie folgt unterstützt:

- **Templatezentrierte Codegenerierung**: Die Templates sind im UML/P-Framework das zentrale Artefakt der Codegenerierung. Sie bilden den Ausgangspunkt für die Implementierung und Ausführung von Generatoren und ermöglichen ebenfalls deren Steuerung und Konfiguration. Gleichzeitig werden die Vorteile von template- und visitorbasierten Ansätzen kombiniert, indem die Generierung an einem beliebigen Punkt der abstrakten Syntax gestartet werden kann. Da es sich bei den Templates genauso wie bei den Modellen um

einfache Textdateien handelt, kann die Codegenerierung entsprechend leicht angepasst und agil erweitert werden. Dafür ist es u.a. wichtig, dass die Templates möglichst gut lesbar in Bezug auf das daraus resultierende Generat sind. Dies wird durch die Auslagerung von komplexen Berechnungen in Kalkulatoren, die Reduzierung von Kontrollstrukturen durch implizite Iteration bei der Templateeinbettung und die Unterstützung der abstrakten Syntax für textuelle Ausgaben von Modellinformationen erreicht.

- **Tracing**: Fehler bei der Entwicklung von Generatoren fallen oft erst im Generat auf. Auch die Erweiterung von Generatoren kann anhand eines bestehenden Generats untersucht und anschließend in den Templates umgesetzt werden (siehe auch Abschnitt 5.3). In beiden Fällen ist es hilfreich, die Quellen für ein Generat ermitteln zu können. Dies wird durch die automatische Einbettung der beteiligten Template- und Modellnamen für jeden Codeabschnitt in Form von Kommentaren im Generat ermöglicht. Die Einbettung ist über die Templates konfigurierbar (siehe Abschnitt 7.4).
- **Kompositionale Codegenerierung**: Für eine kompositionale Codegenerierung wurden in dieser Arbeit zwei Konzepte vorgestellt: die Komposition von Templates mit Hilfe der Templateeinbettung und expliziten Erweiterungspunkten sowie die Komposition auf Generatebene. Da das Datenmodell eines Templates jeweils auf ein einzelnes Element der abstrakten Syntax fokussiert, können die Templates bei entsprechender Aufteilung ähnlich der Spracheinbettung von MontiCore kombiniert und wiederverwendet werden. Darüber hinaus können durch Anwendung dieser Konzepte die Modelle unabhängig voneinander verarbeitet werden.
- **Separate Verarbeitung**: Templates und Modelle lassen sich für die Codegenerierung gezielt auswählen, so dass bei Änderungen nur jeweils die relevanten Teile eines Systems neu generiert werden können.

Bis auf iUML setzen alle der in Tabelle 2.1 in Abschnitt 2.2 aufgeführten kommerziellen UML-Werkzeuge bei der Codegenerierung Round-trip Engineering ein. Dieses Verfahren beschränkt sich bei diesen Werkzeugen zurzeit auf Strukturmodelle wie Klassendiagramme und erlaubt die Programmierung von Methodenverhalten in geschützten Regionen innerhalb der generierten Klassen. Die geschützten Regionen werden vom Generator nicht überschrieben und bleiben so bei einer erneuten Generierung erhalten. Gleichzeitig werden strukturelle Änderungen am Generat in den Modellen aktualisiert. Dieses Verfahren bringt mehrere Einschränkungen mit sich. Die Wesentlichen sind:

- **Limitierung der Abstraktion der Modelle** [Sel03]: Wie in dieser Arbeit gezeigt, kann eine umfangreiche Abweichung der modellierten Architektur von der des Zielsystems sinnvoll sein, um etwa zusätzliche Eigenschaften wie Testbarkeit oder Erweiterbarkeit des generierten Codes zu realisieren (siehe Kapitel 6). Ohne Kenntnis über Art und Konfiguration des Generators oder zusätzliche Informationen im generierten Code über die ursprünglichen Modelle ist ein Reverse Engineering jedoch nur auf einem niedrigen Abstraktionslevel

möglich. Deshalb werden die Klassendiagramme von den Werkzeugen im Allgemeinen in eine identische Klassenstruktur einer objektorientierten Zielsprache übersetzt, um das Round-trip-Verfahren zu ermöglichen. Dabei werden allenfalls Elemente wie get- und set-Methoden für Attribute oder Standardkonstruktoren in den Modellen automatisch ausgeblendet.

- **Beschränkung der Zielsprache**: Der generierte Code muss in die Modelle rückübersetzbar sein, was von den Werkzeugen nur für wenige ausgesuchte Sprachen wie Java und C++ unterstützt wird.
- **Keine Abstraktion von Verhalten**: Der manuell implementierte Code in den geschützten Regionen bezieht sich auf das generierte System. Für dessen Umsetzung sind somit Kenntnisse des Generators und der Zielplattform erforderlich. Dadurch können Änderungen an der Implementierung oder Konfiguration des Generators umfangreiche Anpassungen in den geschützten Regionen nach sich ziehen. Darüber hinaus muss ohne eine abstrakte Beschreibung und generative Umsetzung das Verhalten für jede Zielplattform erneut implementiert werden.

Der letzte Punkt gilt insbesondere auch dann, wenn für die Ergänzung des Generats nur geschützte Regionen oder Codeannotationen an den Modellen ohne weitere Verarbeitung zum Einsatz kommen, auf Round-trip Engineering aber verzichtet wird. Codeannotationen, die sich auf das Generat beziehen, haben darüber hinaus den Nachteil, dass diese zwar auf Modellebene angegeben, hier jedoch nicht nachvollziehbar oder automatisch validierbar sind. Wird die manuelle Verhaltensimplementierung hingegen nur in den generierten Dateien angegeben, müssen diese beim Einsatz von Versionsmanagementsystemen ebenfalls gesichert werden, was zu unnötigen Konflikten in den generierten Anteilen führen kann.

Ein mit geschützten Regionen vergleichbarer Ansatz erzeugt für die generierten Anteile eine abstrakte Klasse, die von einer manuell implementierten Klasse um den zusätzlichen Verhaltenscode vervollständigt werden muss [SV06]. Dieser Ansatz umgeht zwar das Problem bei der Versionsverwaltung, da manuell implementierte und generierte Anteile getrennt bleiben, bietet jedoch ebenfalls keine Abstraktion des Verhaltens.

Andere Verfahren versuchen, mit Hilfe von zusätzlichen Informationen manuelle Änderungen im generierten Code zu erhalten. In [FK03] wird ein 3-Wege-Vergleichsverfahren vorgeschlagen, das das Ergebnis eines Generierungsaufrufs mit dem des vorherigen Aufrufs und dem aktuellen Generat vergleicht. Auf diese Weise können Änderungen im Modell und im Generat identifiziert und zum Teil automatisiert vereinigt werden. Vergleich und Vereinigung finden auf Basis des textuellen Generats statt, während [ALC08] ein ähnliches Verfahren auf dem AST des Generats vorschlägt. In [Gul09] wird hingegen ein Protokoll in einer an XML angelehnten Sprache bei Änderungen am Generat erstellt, so dass die Änderungen bei einer erneuten Generierung automatisiert eingearbeitet werden können. Die Automatisierung dieser Ansätze ist jedoch limitiert, so dass in komplexen Fällen manuelle Eingriffe notwendig sind.

Das Werkzeug UML Lab [UL] berücksichtigt die Templates ebenfalls für das Reverse Engineering. Durch deren Analyse werden generierte Anteile wie get- und set-Methoden in den Modellen nicht mit aufgeführt, so dass die Modelle trotz Round-trip Engineering eine gewisse Abstraktion bieten. Allerdings beschränkt sich der Ansatz zurzeit auf Klassendiagramme und auf die Abstraktion der internen Klassenstruktur. Umfangreichere Abstraktionen auf Klassenebene oder Verhaltensmodelle werden hingegen nicht unterstützt. Dagegen bietet Fujaba [KNNZ99] ein Round-trip Engineering auch für Storydiagramme, das allerdings nur solange funktioniert, wie bei den Änderungen die Namenskonventionen und Implementierungskonzepte von Fujaba eingehalten werden.

Keiner der bisherigen Ansätze und Werkzeuge, die Round-trip Engineering oder manuelle Ergänzungen des generierten Codes einsetzen, kann das Abstraktionspotential der Modellierung vollständig ausnutzen. In der vorliegenden Arbeit werden die Modelle und der Code des Zielsystems hingegen getrennt behandelt. Änderungen am Generat erfolgen ausschließlich über Generatoren oder Angaben innerhalb der Modelle. Ist etwa für ein Element in einem konkreten Modell eine von der allgemeinen Generierung abweichende Umsetzung im Zielsystem erforderlich, muss dies bereits aus dem Modell selbst hervorgehen. Dies kann durch eine abstrakte Auszeichnung über Stereotypen oder eine explizite Modellierung von Verhalten in Form von Statecharts oder Java/P erfolgen. Auf diese Weise wird Struktur und Verhalten eines Systems in sich konsistent und nachvollziehbar modelliert. Gleichzeitig sind Generatoren möglich, die eine konkrete Implementierung vollständig aus den Modellen ableiten. Dabei können beliebige Anpassungen und Optimierungen durchgeführt werden, so dass eine Werkzeug-bedingte Einschränkung des Abstraktionsgrades der Modelle gegenüber dem Zielsystem vermieden wird. Darüber hinaus bleiben Generatoren und damit die Zielplattform austauschbar.

Die Generatoren werden in dieser Arbeit in Form von Templates implementiert, die sich für beliebige textuelle Zielsprachen eignen. Insbesondere ermöglichen Templates eine intuitive und einfache Umsetzung von Generatoren, da sie sich aus einer Beispielimplementierung systematisch ableiten lassen (siehe Abschnitt 5.3). Die Prinzipien wurden anhand der Templateengine FreeMarker in Abschnitt 5.1.3 beschrieben. Daneben existiert eine Vielzahl weiterer Templatesprachen, die sich im Grunde sehr ähnlich sind, aber zum Teil unterschiedliche Schwerpunkte setzen. Während bei FreeMarker und Velocity beliebige Java-Klassen als Datenmodell dienen können, ist MOFScript [ONG+05] eine Templatesprache für MOF-basierte Modelle. Eine Implementierung dieser Sprache steht für das Ecore-Metamodell von EMF zur Verfügung. Die Templates erfordern eine Kompilierung und können dann aus Java heraus auf ausgewählten Modellen ausgeführt werden. Darüber hinaus ist der Aufruf von Java-Klassen mit Standard-Konstruktor sowie von statischen Methoden innerhalb der Templates möglich. Ähnliche Ansätze für Ecore sind JET (Java Emitter Templates) des Eclipse Modeling Projects [Gro09] und EGL (Epsilon Generation Language, [RPKP08b]) der Epsilon Sprachfamilie [KPP06]. Darüber hinaus hat die OMG mittlerweile mit Mof2Text einen Standard für eine MOF-basierte Templatesprache veröffentlicht [OMG08b]. Neben diesen allgemeinen Ansätzen finden sich in Modellierungs- und Sprachentwicklungswerkzeugen auch eigene Templatesprachen, die an das jeweilige Werkzeug

angepasst sind. Ein Beispiel ist xPand von openArchitectureWare [EFH+08], das außerdem einen an aspektorientierte Techniken angelehnten Ansatz für die Integration von Templates bietet [VG07]. Darüber hinaus unterstützen die erwähnten Templatesprachen bis auf FreeMarker und Velocity geschützte Regionen im Generat.

Statt eine eigene an MontiCore angepasste Templatesprache zu entwickeln, wurde in dieser Arbeit ein leichtgewichtiger Ansatz gewählt, indem mit FreeMarker eine bestehende Templateengine um unterstützende Infrastruktur erweitert wurde. Diese bietet u.a. eine gezielte Implementierung und Komposition von Generatoren auf Basis einzelner Elemente einer mit MontiCore entwickelten Sprache, die Auslagerung von komplexen Berechnungen in Java und die Steuerung der Codegenerierung über die Templates. Gleichzeitig erlaubt dieser Ansatz die Wiederverwendung von Templates entsprechend der Sprachkomposition und -vererbung von MontiCore. Für die Erweiterungen wurde FreeMarker selbst nicht verändert, so dass eine Aktualisierung weiterhin möglich ist. Auch ein Austausch der Templateengine lässt sich aufgrund der starken Kapselung im UML/P-Framework recht leicht realisieren (vgl. Abschnitt 7.1 und 7.4).

Ein alternativer Generierungsansatz ist die Verwendung von Modell-zu-Modell-Transformationen [HKGV10], für die eigene Transformationssprachen wie ATL [JABK08], Tefkat [LS06], Stratego [Vis04] oder GReAT [BNBK06] existieren. Dieser Ansatz setzt jedoch voraus, dass für die Zielsprache ein Metamodell zur Verfügung steht. Dafür kann das Ergebnis einer Transformation von weiteren Transformationen nachträglich noch verändert werden. In der vorliegenden Arbeit wird dieses Verfahren für die optionalen Refactorings verwendet (siehe Abschnitt 5.2), wobei zurzeit die Transformationen über die API des ASTs erfolgen. Dies ist vergleichbar mit dem Ansatz in [ABE+06], der Transformationssprachen einen hohen Einarbeitungsaufwand zuschreibt und deshalb als Alternative eine Java-basierte Bibliothek für Transformationen vorschlägt. Das UML/P-Framework gibt hingegen die Implementierung der Refactorings nicht vor, so dass dafür auch Transformationssprachen verwendet werden können.

Die mit dem UML/P-Framework ausgelieferten Generatoren, die auch für die Generierung der in dieser Arbeit als Beispiel dienenden TripleLogo-Applikation eingesetzt werden, basieren auf den Konzepten in [Rum04b]. Diese wurden um die in Kapitel 6 diskutierten Ansätze für eine kompositionale Codegenerierung ergänzt. Die Generatoren und Konzepte dieser Arbeit sind dabei nicht als die einzig mögliche Umsetzung für die UML/P zu verstehen. Sie dienen vielmehr als Grundlage für die Entwicklung beliebiger weiterer Generatoren, um etwa bestimmte Domänen oder Technologien zu unterstützen. In der Literatur finden sich darüber hinaus diverse Varianten für mögliche Umsetzungen einzelner UML-Diagramme und deren Elemente, die alternativ zu [Rum04b] im UML/P-Framework verwendet und so evaluiert werden können. Beispiele sind [GB99, NT05, Ali10] für Statecharts oder [Nob96, GDL03, AHM07] für Assoziationen. Mehrere verschiedene Diagramme werden dabei jedoch selten behandelt. Eine Ausnahme ist der Ansatz für eine kombinierte Generierung aus Klassendiagrammen und Statecharts in [PD07], der jedoch keine separate Verarbeitung der Modelle vorsieht. Schließlich sind auch Ansätze jenseits der Generierung von System- oder Testcode möglich. So wird etwa in [BW08, ABGR10] anstelle einer Generierung von Test- oder Prüfmethoden aus OCL eine Analyse gegenüber den

Klassendiagrammen durch Transformation der Modelle in mathematische Theorembeweiser vorgeschlagen. Eine solch formale Verifikation von UML-Modellen skaliert jedoch oft nicht für größere Modelle [FGDS06], weshalb in dieser Arbeit der Testfallgenerierung aus OCL Vorrang gegeben wurde.

8.4 Modellgetriebene Ansätze und Werkzeuge

Hinsichtlich modellgetriebener Ansätze ist die UML/P eng mit xUML (Executable UML) und fUML (Foundational UML) verwandt (siehe auch Abschnitt 8.1). Auch hier ergänzt eine Aktionssprache die Modelle, so dass daraus ein vollständiges und lauffähiges System abgeleitet werden kann. Eines der bekanntesten und umfassendsten Werkzeuge für xUML ist iUML [iUML, RFW+04], das jedoch dem xUML-Ansatz entsprechend keine Objektdiagramme oder OCL unterstützt. Darüber hinaus wird eine proprietäre Aktionssprache verwendet, die im Gegensatz zur vorliegenden Arbeit nicht austauschbar ist. Für fUML gibt es aufgrund des erst kürzlich veröffentlichten Standards [OMG11b] bisher kaum Werkzeugunterstützung. Erste Ansätze unterstützen mit Aktionen und Aktivitäten in [CD08b] sowie Klassendiagrammen und der Aktionssprache in [LLP+10] nur Teilaspekte. Die UML/P bietet mit dieser Arbeit hingegen eine umfassende Werkzeuginfrastruktur für die modellgetriebene Umsetzung und Qualitätssicherung von Softwaresystemen einschließlich anpassbarer Kontextanalyse und Codegeneratoren für sämtliche in Kapitel 3 beschriebenen Sprachen.

Die Trennung von plattformunabhängigen (PIM) und -spezifischen Modellen (PSM) wie beim MDA-Ansatz ist in der UML/P in dieser Form nicht explizit vorgesehen. Stattdessen werden die technischen Aspekte der Zielplattform in den Generatoren gekapselt, so dass auch hier die Modelle plattformunabhängig entworfen werden können. Der Umfang der Abstraktion von einer bestimmten Zielplattform wird dabei dem Modellierer überlassen. So ist es auch möglich, Modelle mit einem stärkeren Plattformbezug zu entwerfen. Ein Beispiel ist die Java/P-Klasse in Quellcode 3.34 im Abschnitt 3.7, die eine Swing-spezifische Umsetzung des Controllers der TripleLogo-Applikation enthält, diese jedoch von den abstrakteren Modellen trennt. Dennoch handelt es sich auch bei dieser Klasse um ein Modell, das von Generatoren verarbeitet wird und somit vom generierten System abstrahiert. Auf diese Weise können abstrakte Modelle durch konkretere Modelle im Laufe des Entwicklungsprozesses ergänzt werden. Die UML/P bietet damit auch das Potenzial, ein Vorgehen nach dem MDA-Konzept zu unterstützen. Gleichzeitig zeigt dieses Beispiel, wie sich mit der UML/P Modelle sukzessiv in ehemals rein codezentrierte Entwicklungsprozesse integrieren sowie mit bestehenden Frameworks, Bibliotheken und Legacy Code nutzen lassen. Insbesondere die Nutzung der umfangreichen Java-Bibliothek bietet einen beträchtlichen Vorteil gegenüber einer proprietären Aktionssprache, zumal Java aufgrund seiner Laufzeitumgebung ebenfalls von einer technischen Plattform abstrahiert. Darüber hinaus sind solche Bibliotheksreferenzen optional und können durch entsprechende Modelle ersetzt werden. Da Java als Aktionssprache in den Modellen austauschbar ist, können zukünftig Varianten der UML/P auch für andere Sprachen entstehen.

Die Entscheidung, den Controller als eine Swing-Komponente umzusetzen, hätte ebenfalls in einem projektspezifischen Generator getroffen werden können, der speziell die Umsetzung dieser Klasse realisiert. Auf diese Weise können beliebig komplexe Aspekte, die auf Modellebene keine Rolle spielen sollen, auch über Codegeneratoren ergänzt werden.

Trotz der in [GKR+07] aufgeführten Vorteile einer textbasierten Modellierung, werden textuelle Notationen für die UML bisher fast ausschließlich zur Speicherung sowie zum Im- und Export von Modellen eingesetzt. Neben Werkzeugen für das Erstellen von Bilddateien, existieren nur wenige Werkzeuge, die eine textuelle UML-Notation für die Entwicklung von Softwaresystemen einsetzen (siehe auch Abschnitt 8.1). TextUML [TU] bietet eine Ecore-basierte textuelle Modellierung von Klassendiagrammen in Eclipse mit Syntaxhervorhebung und der automatischen Aktualisierung einer graphischen Darstellung. Weitergehende Funktionen wie anpassbare Kontextanalysen oder Codegeneratoren sind nicht integriert, lassen sich aber theoretisch über andere EMF-Werkzeuge realisieren. Mit USE [GBR07] können OCL-Bedingungen auf textuellen Klassendiagrammen spezifiziert werden. Auf dieser Basis erlaubt der Ansatz die Evaluierung einer entsprechenden Implementierung und dient damit nur der Spezifikation aber nicht der Generierung von Softwaresystemen. Mit Umple [For10] kann hingegen aus einer textuellen Notation für Klassendiagramme Java-, PHP- oder Ruby-Code generiert werden, die der Ansatz als Basissprachen bezeichnet. Diese können sowohl innerhalb der Modelle verwendet, als auch Modellierungselemente wie Assoziationen innerhalb der Basissprachen angegeben werden. Bei der Verarbeitung übersetzt der Generator die Modellelemente in Code der entsprechenden Basissprache. Elemente der Basissprache werden hingegen unverarbeitet übernommen, so dass für Zugriffe auf Modellelemente die generierten Schnittstellen bekannt sein müssen. Diese werden vom Umple zusammen mit der Codegenerierung vorgegeben und sind somit nicht anpassbar. Neben Klassendiagrammen ist eine Erweiterung für Statecharts in der Entwicklung [Bad10], die innerhalb der Klassen selbst angegeben werden sollen. Eine Unterstützung für weitere Diagramme oder für Profile ist bisher hingegen nicht geplant. Insgesamt existiert somit bisher keine mit der UML/P vergleichbare Werkzeuginfrastruktur, die eine solch umfassende modellgetriebene Entwicklung auf Basis von textuellen UML-Modellen mit frei anpassbaren Codegeneratoren und Java als abstrakte Aktionssprache mit Modellbezug erlaubt.

Die Auswahl an graphischen Modellierungswerkzeugen für die UML ist deutlich umfangreicher (siehe Abschnitt 2.2). Da viele auf Round-trip Engineering setzen, ist die Codegenerierung und deren Anpassung jedoch häufig eingeschränkt (siehe Abschnitt 8.3). Eine Aktionssprache auf Modellebene ist bis auf xUML-Werkzeuge hingegen kaum zu finden. Stattdessen wird Code in geschützten Regionen im generierten Code oder als Annotationen am Modell ergänzt, der sich dann aber auf das Generat bezieht und ungeprüft übernommen wird. OCL wird selten direkt unterstützt. Es existieren jedoch diverse eigenständige Werkzeuge, die sich teilweise auch als Plugins in einzelne Modellierungswerkzeuge einbinden lassen. Ein Vergleich von OCL-Werkzeugen findet sich in [Hel10].

Ein weiteres Problem von graphischen Modellierungswerkzeugen ist die Unterstützung der Teamarbeit [GF10]. Einige kommerzielle Werkzeuge wie Enterprise Architect [EA], Together [Tog]

oder iUML [iUML] bieten zwar auch hierfür Lösungen an. Bei Enterprise Architect und iUML basieren diese jedoch auf einem zentralen Repository, wobei pro Modellelement nur für einen Nutzer Schreibrechte vergeben werden, um Konflikte zu vermeiden [Spa11, Ken04]. Dies setzt voraus, dass die Nutzer für eine Änderung der Modelle mit dem Repository verbunden sind, was die Flexibilität einer verteilten Entwicklung deutlich einschränkt. Bei Together können die Modelle hingegen als XMI-Datei gespeichert werden, so dass der Einsatz von herkömmlichen Versionsmanagementsystemen wie CVS oder SVN möglich ist [Bor11]. Durch die Speicherung von Positionsangaben der Modellelemente führt dieses Verfahren jedoch zu häufigen Konflikten. Darüber hinaus ist der Modellierer mit der XMI-Darstellung nicht unbedingt vertraut, was eine Konfliktlösung und ein Nachvollziehen von Änderungen zusätzlich erschwert. Die textuelle Notation der UML/P ermöglicht demgegenüber eine Versionsverwaltung sowie die Anzeige und das Beheben von Konflikten auf Basis der Modelle mit existierenden Versionsmanagementsystemen. Gleichzeitig können dadurch auch Unterschiede zwischen einzelnen Revisionen eines Modells untersucht und Änderungen rückgängig gemacht werden.

Auch in anderen Aspekten wurde in dieser Arbeit darauf geachtet, die Werkzeuginfrastruktur möglichst einfach und agil zu halten. Dazu gehören die kompakte Modellnotation, die Steuerung und Implementierung von Codegeneratoren auf Templatebasis sowie die Ausführung und Parametrisierung von Kontextanalyse und Generierung durch reguläre Java-Aufrufe. Da letzteres nur eine Java-Laufzeitumgebung erfordert, ermöglicht dies einen flexiblen Einsatz des UML/P-Frameworks wie z.B. die Skript-gesteuerte Ausführung im Rahmen von Nightly Builds. Ein weiteres Beispiel ist die einfache Definition von Profilen. In der UML ist dafür eine Erweiterung des Metamodells erforderlich. So bildet etwa jeder Stereotyp eine eigene Klasse im Metamodell, die durch Attribute und OCL-Bedingungen präzisiert werden kann. Ein entsprechendes Verfahren bietet etwa Together für das EMF-Metamodell an [Bor11]. In der UML/P können Stereotypen hingegen ohne weiteres angegeben und bei Bedarf durch Kontextbedingungen oder Codegeneratoren ergänzt werden.

Insgesamt bietet die Werkzeuginfrastruktur der UML/P eine vollständige Unterstützung für die modellgetriebene Entwicklung und Qualitätssicherung von Softwaresystemen einschließlich anpassbarer Sprachzusammensetzung und Kontextbedingungen, Definition von Profilen, flexibler und kompositionaler Codegenerierung mit Templates, Einbindung von Legacy Code, Editoren für die textuelle Modellierung sowie die Unterstützung von Teamarbeit durch Versionsmanagementsysteme und Nightly Builds.

Kapitel 9

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde eine umfassende Werkzeuginfrastruktur zur agilen modellgetriebenen Entwicklung mit der UML/P vorgestellt. Diese basiert auf der Motivation und den Forschungszielen in Kapitel 1. Im Vordergrund standen dabei die in [Rum04a, Rum04b] entwickelten theoretischen Konzepte und Modellierungsansätze der UML/P, bei denen insbesondere die Agilität in der Softwareentwicklung eine wichtige Rolle spielt. Für eine entsprechende Werkzeuginfrastruktur ergab sich daraus die Notwendigkeit einer effizienten Erstellung, Analyse und Verarbeitung von Modellen sowie eine flexible Anpassung von Sprache und Infrastruktur an spezifische Anforderungen von Projekten, Unternehmen, Domänen oder Technologien.

Allgemein kann die Arbeit somit dem Bereich der modellbasierten Softwareentwicklung zugeordnet werden, dessen grundlegende Begriffe und Ziele sowie existierende Konzepte und Werkzeuge in Kapitel 2 behandelt wurden. Neben einem Einblick in die Entwicklung einer werkzeugunterstützten Sprachverarbeitung wurden darüber hinaus die Unterschiede von graphischen und textuellen Sprachen diskutiert. Letztere bieten insbesondere im Hinblick auf eine agile und effiziente Entwicklung von Softwaresystemen deutliche Vorteile. Aus diesem Grund wurde in dieser Arbeit eine an Java angelehnte textuelle Syntax für die einzelnen Sprachen der UML/P entwickelt und in Kapitel 3 ausführlich beschrieben. Die insgesamt sieben Sprachen erlauben eine vollständige textuelle Modellierung von Softwaresystemen. Während Klassendiagramme und Statecharts Struktur und Verhalten beschreiben, können Objekt- und Sequenzdiagramme mittels einer Testspezifikationssprache kombiniert und zur Modellierung von Testfällen eingesetzt werden. Bedingungen in OCL/P oder Ausdrücke in Java/P ergänzen die Modelle, wobei diese Sprachen sowohl als Einbettung in den Modellen als auch separat verwendbar sind. Das Besondere an Java/P ist, dass es sich trotz der gegenüber Java unveränderten Syntax und der Möglichkeit, Legacy Code wie die Java-Bibliothek zu verwenden, auf die Modelle bezieht und wie diese vom Zielsystem abstrahiert. Die Java-Syntax wird somit als abstrakte Aktionssprache in den Modellen verwendet sowie entsprechend validiert und verarbeitet. Dies bietet bisher kein anderer bekannter Ansatz.

Die Unterschiede der in [Rum04a, Rum04b] verwendeten graphischen zur textuellen Syntax dieser Arbeit wurden in Abschnitt 3.9 zusammengefasst. Von besonderer Bedeutung ist dabei die

Einordnung der als einfache Textdateien abgespeicherten Modelle in eine hierarchische Paketstruktur wie bei Java oder C++. Über Import-Anweisungen und qualifizierte Namen können dadurch Modelle und Modellelemente eindeutig referenziert und so modellübergreifend in Beziehung gesetzt sowie im Dateisystem ermittelt werden, so dass eine Skalierbarkeit des Modellierungsansatzes auch für große Projekte sicher gestellt ist. Gleichzeitig bilden die referenzierbaren Elemente eine Art Schnittstelle für einen modell- und sprachübergreifenden Informationsaustausch.

Die formale Sprachdefinition und -verarbeitung basiert auf dem parallel zu dieser Arbeit entwickelten MontiCore-Framework [GKR+06, Kra10]. Dabei wurden die Grammatikbeschreibungen im Anhang C durch die Kontextbedingungen in Kapitel 4 ergänzt. Letztere überprüfen die Konsistenz der Modelle innerhalb und zwischen den einzelnen Sprachen, wobei sowohl Intra- als auch Inter-Modell-Bedingungen berücksichtigt wurden. Entsprechende Kategorien, die Gruppierung nach Sprachelementen und die Zuordnung von Identifikationsnummern erleichtern nicht nur das Auffinden und Referenzieren von Bedingungen, sondern auch deren Wiederverwendung für die Integration von Teilen der UML/P in andere Sprachen. Darüber hinaus bieten die Bedingungen einen allgemein sinnvollen Basissatz für die UML/P, der an unterschiedliche Abstraktionsgrade, Sprachprofile oder Nutzeranforderungen angepasst und erweitert werden kann.

Eine der grundlegenden Eigenschaften von Modellen ist deren Abstraktion gegenüber dem eigentlichen System. Ohne die Möglichkeit zur Abstraktion wäre die UML/P nur eine weitere Programmiersprache. Daher können die Modelle trotz der umfassenden Darstellung von Struktur und Verhalten das System nicht allein bestimmen. Stattdessen ergibt sich dieses erst durch Auswahl und Konfiguration von Generatoren, die fehlende Details der Modelle ergänzen. Um die Abstraktionsfähigkeit der UML/P auch zukünftig für beliebige Domänen und Technologien zu erhalten, sowie die Codegenerierung auch im Projektkontext agil anpassen zu können, ist eine einfache Entwicklung und Erweiterung von Generatoren essenziell. Hierfür wurde in Kapitel 5 ein Ansatz vorgestellt, bei dem Templates das zentrale Artefakt für die Entwicklung, Konfiguration und Ausführung der Codegenerierung bilden. Durch die Kombination von template- und visitorbasierten Konzepten können Generatoren gezielt für einzelne Sprachelemente entwickelt und zu komplexeren Generatoren komponiert oder auch separat ausgeführt werden. Eine möglichst einfache und lesbare Erstellung von Templates wurde dabei u.a. durch die Anbindung von Java für komplexe Berechnungen und die Unterstützung für textuelle Ausgaben von Modellinformationen erreicht. Darüber hinaus ergänzen eine Vorgehensweise für die Entwicklung und Qualitätssicherung sowie Konzepte für die Erweiterung und Komposition von Generatoren den Ansatz.

Konkrete Konzepte für die Generierung von Produktiv- und Testcode aus den verschiedenen Modellarten der UML/P wurden in Kapitel 6 diskutiert. Der Fokus lag hier insbesondere auf einer kompositionalen, erweiterbaren und testbaren Architektur, um eine modulare Codegenerierung zu erreichen. Ebenfalls behandelt wurde der Umgang mit Unterspezifikation und projektspezifischer Variabilität im Rahmen der Codegenerierung sowie ein Ansatz für eine sprachunabhängige Messung der Testüberdeckung auf Modellen.

Die als Framework realisierte Werkzeuginfrastruktur zeichnet insbesondere eine durchgängig kompositionale und leichtgewichtige Umsetzung aus, deren Aufbau, Architektur und Nutzung in Kapitel 7 zusammengefasst wurden. So wurde jede Sprache als eigene Komponente entwickelt, die Grammatik, Symboltabelle, Kontextbedingungen und Generatoren unabhängig von anderen Sprachen enthält. Diese wurden zu weiteren Komponenten kombiniert, um die Spracheinbettung, die UML/P als Sprachfamilie oder Zusatzfunktionalitäten wie Editoren zu realisieren. Auf die gleiche Weise können zukünftig weitere Sprachen der UML/P hinzugefügt werden oder neue Sprachfamilien entstehen, ohne bestehende Komponenten zu verändern. Selbst bei einer Änderung der eingebetteten Sprachen können viele Komponenten wiederverwendet werden. Mit der Werkzeuginfrastruktur dieser Arbeit ist somit gleichzeitig ein Sprachbaukasten für die Entwicklung von Variationen der UML/P entstanden.

Aber auch im Hinblick auf die modellgetriebene Softwareentwicklung ist das UML/P-Framework hochgradig flexibel. Modelle und Generatoren lassen sich in beliebigen Texteditoren bearbeiten. Sprachprofile können ohne weiteres in den Modellen verwendet und bei Bedarf durch Kontextbedingungen oder Generatoren ergänzt werden. Für die Ausführung der Kontextanalyse und Generatoren genügt eine Java-Laufzeitumgebung. Nur für die Änderung oder Erweiterung von Kontextbedingungen oder die Verwendung von Java im Rahmen der Generatorimplementierung ist zusätzlich ein Java-Compiler erforderlich. Durch diese geringen Anforderungen können Profile, Kontextbedingungen und Generatoren agil im Laufe des Entwicklungsprozesses eines Softwaresystems an den Projektkontext angepasst oder das Framework in andere Werkzeuge eingebunden werden. Darüber hinaus erlaubt die modulare Modellverarbeitung eine gezielte Ausführung und Verarbeitung von Modellen, Kontextanalyse und Generatoren.

In Kapitel 8 wurden schließlich die zentralen Ergebnisse dieser Arbeit diskutiert und mit anderen Arbeiten sowie existierenden Werkzeugen verglichen. Danach sind die wichtigsten Konzepte und Ergebnisse der vorliegenden Arbeit, die in diesem Umfang bisher kein anderes UML-Werkzeug bietet, wie folgt:

- Vollständige textuelle Sprachfamilie für die modellgetriebene Entwicklung von Struktur, Verhalten und Qualitätssicherung von Softwaresystemen.
- Referenzielle Beziehungen zwischen Modellen und Einordnung in eine Paketstruktur, die gemeinsam mit der Symboltabelle die Basis und Schnittstellen für einen Informationsaustausch zwischen den Modellen bilden.
- Anhebung einer existierenden Programmiersprache als abstrakte Aktionssprache auf Modellebene.
- Vollständige sprachinterne und sprachübergreifende sowie modellinterne und modellübergreifende kompositionale Kontextanalyse.
- Konzepte und Framework für eine kompositionale und erweiterbare Codegenerierung.
- Modulare Codegenerierung durch separate Verarbeitung von Modellen und Generatoren.
- Verfahren zur sprachunabhängigen Messung der Testüberdeckung auf Modellen.

- Kompositionale und leichtgewichtige Werkzeuginfrastruktur mit variabler Sprachzusammensetzung, konfigurier- und erweiterbaren Kontextbedingungen mit Berücksichtigung unterschiedlicher Abstraktionsebenen der Modelle, Definition von Sprachprofilen, vollständiger und flexibel anpassbarer Codegenerierung, Einbindung von Legacy Code, Tracing der an einem Generat beteiligten Artefakte, flexiblem Einsatz durch Konsolensteuerung und Unterstützung von Teamarbeit.
- Optionales Plugin für Eclipse mit Komfortfunktionen wie Syntaxhervorhebung, Autovervollständigung oder Fehleranzeige.

Die Anwendbarkeit des Frameworks wurde anhand eines durchgehenden Beispiels demonstriert, das mittels der beschriebenen Codegenerierungskonzepte in eine vollständig lauffähige Applikation und ausführbare Testfälle überführt wurde. Darüber hinaus werden Teile der UML/P und des Frameworks bereits in anderen Projekten eingesetzt:

- *Modal Object Diagrams* [MRR11c]: Sprachprofil der UML/P, das eine Kennzeichnung von Objektdiagrammen als positive und negative Beispiele oder Invarianten erlaubt und diese gegenüber Klassendiagrammen validiert.
- *CDDiff* [MRR11a, MRR11b]: Vergleicht zwei Klassendiagramme und generiert ein Objektdiagramm, das nur mit einem der Klassendiagramme konsistent ist.
- *MontiWeb* [DRRS09]: Sprache zur Modellierung von Websystemen, bei der Klassendiagramme zur Beschreibung der Datenstruktur verwendet werden.
- *MontiCoreTF*: Eine in der Entwicklung befindliche Transformationssprache für MontiCore. Bei dieser beschreiben Objektdiagramme die linke und rechte Regelseite von Graphersetzungsregeln auf den ASTs der beteiligten Modelle. Zusätzlich können Änderungsoperationen zwischen den Regelseiten angegeben werden, die auf einem Klassendiagramm als Datenmodell basieren.
- *MontiArc* [HRR10]: Sprache für die Architekturbeschreibung von verteilten, interaktiven Systemen, die auf der `Common`-Grammatik aufbaut und sich somit mit der UML/P denselben Sprachkern teilt (siehe Abschnitt 7.1). Darüber hinaus wird ebenfalls das Generierungsframework eingesetzt.
- Δ-*MontiArc* [HRRS11]: Sprache zur Entwicklung von Produktlinien, die das UML/P-Framework ähnlich wie MontiArc nutzt.
- *MontiCore*: Die für die Sprachverarbeitung notwendige Infrastruktur wird in MontiCore über das Generierungsframework aus den Grammatiken erzeugt. Darüber hinaus steht ein alternativer Generator für die Erstellung einer EMF-kompatiblen abstrakten Syntax zur Verfügung.

Wie die obigen Beispiele zeigen, ist das in dieser Arbeit entstandene Framework insbesondere durch dessen modularen Aufbau vielfältig einsetzbar, so dass selbst Anwendungsformen jenseits

der Entwicklung von Softwaresystemen möglich sind. Neben diesem Einsatz als Forschungsplattform kann zukünftig die Werkzeuginfrastruktur auch für die modellgetriebene Softwareentwicklung weiter ausgebaut werden. So sind Bibliotheken von Sprachprofilen und Generatoren denkbar, die auf bestimmte Domänen oder Technologien zugeschnittene Komponenten zur Verfügung stellen. Solche Komponenten können auch einzelne Aspekte wie Entwurfsmuster oder Codeinstrumentierung kapseln (siehe Abschnitte 5.4 und 6.4). Dabei ist zu untersuchen, wie sich Abhängigkeiten etwa zwischen einzelnen Profilen und Generatoren für solch eine Bibliothek geeignet darstellen lassen, wobei Feature-Diagramme [CE00] ein möglicher Ansatz sind.

Moderne IDEs bieten oft Unterstützung für ein Refactoring von Quellcode. Ein entsprechender Ansatz könnte auch für Modelle entwickelt werden. Neben dem automatischen Umbenennen von Referenzen oder dem Extrahieren von Klassen und Methoden, ist u.a. ein Hinzufügen oder Entfernen der Hierarchie in Statecharts [Rum04a] sowie ein Ableiten von Klassendiagrammen aus Objektdiagrammen oder Statecharts aus Sequenzdiagrammen denkbar. Erste Ansätze, die ein sprachübergreifendes Refactoring anbieten, existieren bereits [MB05].

Insgesamt ist mit dieser Arbeit eine umfassende Infrastruktur für die agile modellgetriebene Entwicklung von Softwaresystemen mit der UML/P entstanden. Die textuelle Modellierung, die weitreichende Anpassbarkeit von Sprachzusammensetzung, Kontextanalyse und Codegeneratoren, sowie die leichtgewichtige und modulare Werkzeuginfrastruktur erlauben einen flexiblen und effizienten Einsatz für unterschiedlichste Technologien und Domänen. Nicht zuletzt besitzt die UML/P damit das Potenzial für eine Anwendung auch im Rahmen von zukünftigen Entwicklungen und Forschungsvorhaben in der Softwaretechnik.

Anhang A

Abkürzungen

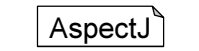
AOP	Aspect-Oriented Programming
API	Application Programming Interface (Programmierschnittstelle)
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
BE	Back-End
CASE	Computer-Aided Software Engineering
CD	Class Diagram
CmD	Component Diagram
DD	Deployment Diagram
DSC	deterministic Statechart
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
ETS	Extensible Type System
FE	Front-End
GLI	Generation and Language Infrastructure
GPL	General Purpose Language
GPML	General Purpose Modeling Language
Java/P	Java auf Modellebene der UML/P
MBD	Model-Based Engineering
MBE	Model-Based Engineering
MBSE	Model-Based Software Engineering
MBSD	Model-Based Software Development
MBTM	Model-Based Testing Metrics
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MDSE	Model-Driven Software Engineering

MOF	Meta Object Facility
OCL	Object Constraint Language
OCL/P	Object Constraint Language der UML/P
OD	Object Diagram
OMG	Object Management Group
OSTP	Online Software Transformation Platform
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query / Views / Transformations
RE	Runtime Environment
SC	Statechart
SD	Sequence Diagram
TC	Test Case (Kurzbezeichnung für die Testspezifikationssprache)
TE_RE	Text Editor Runtime Environment
UML	Unified Modeling Language
UML/P	Unified Modeling Language / Programmier-geeignet
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Anhang B

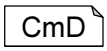
Diagramm- und Quellcodemarkierungen

Im Folgenden sind Notationen aufgeführt, die innerhalb von Abbildungen und Quellcode zur Kennzeichnung der zugrundeliegenden Sprache oder für Anmerkungen verwendet werden.



AspectJ-Quellcode

Klassendiagramm



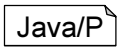
Komponentendiagramm

Verteilungsdiagramm

Freemarker-Template

Graph

Java-Quellcode



Java/P-Quellcode

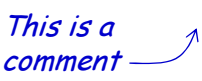
MontiCore-Grammatik

Modell oder DSL in einer beliebigen Notation

Objektdiagramm

Statechart

Sequenzdiagramm



Kommentar mit Verweis auf das kommentierte Element

Anhang C

Grammatiken der UML/P

Im Folgenden sind die vollständigen Grammatiken in der von MontiCore bereitgestellten Syntax aufgeführt. Tabelle C.1 fasst die verwendete Spracheinbettung der vorliegenden Fassung der UML/P zusammen. Beim `InvariantContent` kann durch die Angabe von `java:` bzw. `ocl:` vor der Bedingung zwischen Java-Ausdrücken und OCL-Invarianten unterschieden werden (vgl. Quellcode C.4, Zeile 184).

Sprache	*Einbettungspunkt*	*Einbettung*
CD	`Body`	`mc.javadsl.JavaDSL.BlockStatement`
	`Value`	`mc.javadsl.JavaDSL.Expression`
	`InvariantContent`	`mc.javadsl.JavaDSL.Expression` und `mc.umlp.ocl.OCL.OCLConstraint`
OD	`Value`	`mc.javadsl.JavaDSL.Expression`
	`InvariantContent`	`mc.javadsl.JavaDSL.Expression` und `mc.umlp.ocl.OCL.OCLInvariant`
SC	`Expression`	`mc.javadsl.JavaDSL.Expression`
	`Statements`	`mc.javadsl.JavaDSL.BlockStatement`
	`InvariantContent`	`mc.javadsl.JavaDSL.Expression` und `mc.umlp.ocl.OCL.OCLExpression`
SD	`Expression`	`mc.javadsl.JavaDSL.Expression`
	`InvariantContent`	`mc.javadsl.JavaDSL.Expression` und `mc.umlp.ocl.OCL.OCLExpression`
TC	`Code`	`mc.javadsl.JavaDSL.Statement`

Tabelle C.1: Spracheinbettung in der UML/P

C.1 Literale

MC

```
1  package mc.literals;
2
3  /**
4   * This grammar defines Java compliant literals, e.g., Boolean, Char, String, ... .
5   * The scope of this grammar is to ease the reuse of literals structures in
6   * Java-like sublanguages, e.g., by grammar inheritance or grammar embedment.
7   *
8   * @author Martin Schindler
9   * @version 1.3
10  */
11 grammar Literals {
12
13     /*=================================================================*/
14     /*============================== OPTIONS ==========================*/
15     /*=================================================================*/
16
17     options {
18         parser lookahead=3
19         lexer lookahead=3
20         nostring noident nows
21     }
22
23     /*=================================================================*/
24     /*======================== INTERFACE DEFINITIONS ==================*/
25     /*=================================================================*/
26
27     ast BooleanLiteral =
28         method public boolean getValue(){
29           return this.source == ASTConstantsLiterals.TRUE;
30         };
31
32     ast CharLiteral =
33         method public char getValue() {
34           try {
35             return
36                 mc.literals.LiteralsHelper.getInstance().decodeChar(getSource());
37           }
38           catch (java.io.CharConversionException e) {
39             return ' ';
40           }
41         };
42
43     ast StringLiteral =
44         method public String getValue() {
45           try {
46             return
47                 mc.literals.LiteralsHelper.getInstance().decodeString(getSource());
48           }
49           catch (Exception e) {
50             return "";
51           }
52         };
53
54     ast IntLiteral =
55         method public int getValue() {
56           try {
57             return
58                 mc.literals.LiteralsHelper.getInstance().decodeInt(getSource());
59           }
60           catch (NumberFormatException e) {
61             return 0;
62           }
63         };
```

```
64      ast SignedIntLiteral astextends IntLiteral =
65          method public int getValue() {
66            if (negative) {
67              return - super.getValue();
68            }
69            return super.getValue();
70          };
71
72      ast LongLiteral =
73          method public long getValue() {
74            try {
75              return
76                  mc.literals.LiteralsHelper.getInstance().decodeLong(getSource());
77            }
78            catch (NumberFormatException e) {
79              return 0;
80            }
81          };
82
83      ast SignedLongLiteral astextends LongLiteral =
84          method public long getValue() {
85            if (negative) {
86              return - super.getValue();
87            }
88            return super.getValue();
89          };
90
91      ast FloatLiteral =
92          method public float getValue() {
93            try {
94              return
95                  mc.literals.LiteralsHelper.getInstance().decodeFloat(getSource());
96            }
97            catch (NumberFormatException e) {
98              return 0f;
99            }
100         };
101
102     ast SignedFloatLiteral astextends FloatLiteral =
103         method public float getValue() {
104           if (negative) {
105             return - super.getValue();
106           }
107           return super.getValue();
108         };
109
110     ast DoubleLiteral =
111         method public double getValue() {
112           try {
113             return
114                 mc.literals.LiteralsHelper.getInstance().decodeDouble(getSource());
115           }
116           catch (NumberFormatException e) {
117             return 0d;
118           }
119         };
120
121     ast SignedDoubleLiteral astextends DoubleLiteral =
122         method public double getValue() {
123           if (negative) {
124             return - super.getValue();
125           }
126           return super.getValue();
127         };
128
129     /** ASTLiteral is the interface for all literals
130     */
131     interface Literal;
```

```
132     /** ASTSignedLiteral is the interface for all literals (NullLiteral,
133         BooleanLiteral, CharLiteral, StringLiteral and all NumericLiterals).
134         Compared to Literal it also includes negative NumericLiterals
135     */
136     interface SignedLiteral;
137
138     /** The interface ASTNumericLiteral combines the numeric literal types for
139         Integer, Long, Float and Double
140     */
141     interface NumericLiteral extends Literal;
142
143     /** The interface ASTNumericLiteral combines the numeric literal types for
144         Integer, Long, Float and Double.
145         Compared to NumericLiteral it also includes negative numbers.
146     */
147     interface SignedNumericLiteral extends SignedLiteral;
148
149     /*========================================================================*/
150     /*============================== PARSER RULES ============================*/
151     /*========================================================================*/
152
153     /** ASTNullLiteral represents 'null'
154     */
155     NullLiteral implements Literal, SignedLiteral =
156         "null";
157
158     /** ASTBooleanLiteral represents "true" or "false"
159         @attribute source String-representation (including '"').
160     */
161     BooleanLiteral implements Literal, SignedLiteral =
162         source:["true" | "false"];
163
164     /** ASTCharLiteral represents any valid character parenthesized with "'".
165         @attribute source String-representation (including "'").
166     */
167     CharLiteral implements Literal, SignedLiteral =
168         source:Char;
169
170     /** ASTStringLiteral represents any valid character sequence parenthesized
171         with '"'.
172         @attribute source String-representation (including '"').
173     */
174     StringLiteral implements Literal, SignedLiteral =
175         source:String;
176
177     /** ASTIntLiteral represents a positive Integer number.
178         @attribute source String-representation (including '"').
179     */
180     IntLiteral implements NumericLiteral =
181         source:Num_Int;
182
183     /** ASTSignedIntLiteral represents a positive or negative Integer number.
184         @attribute source String-representation (including '"').
185     */
186     SignedIntLiteral implements (("-")? Num_Int)=> SignedNumericLiteral =
187         (negative:["-"])? source:Num_Int;
188
189     /** ASTLongLiteral represents a positive Long number.
190         @attribute source String-representation (including '"').
191     */
192     LongLiteral implements NumericLiteral =
193         source:Num_Long;
194
195     /** ASTSignedLongLiteral represents a positive or negative Long number.
196         @attribute source String-representation (including '"').
197     */
198     SignedLongLiteral implements (("-")? Num_Long)=> SignedNumericLiteral =
199         (negative:["-"])? source:Num_Long;
```

```
200     /** ASTFloatLiteral represents a positive Float number.
201         @attribute source String-representation (including '"').
202     */
203     FloatLiteral implements NumericLiteral =
204         source:Num_Float;
205
206     /** ASTSignedFloatLiteral represents a positive or negative Float number.
207         @attribute source String-representation (including '"').
208     */
209     SignedFloatLiteral implements (("-")? Num_Float)=> SignedNumericLiteral =
210         (negative:["-"])? source:Num_Float;
211
212     /** ASTDoubleLiteral represents a positive Double number.
213         @attribute source String-representation (including '"').
214     */
215     DoubleLiteral implements NumericLiteral =
216        source:Num_Double;
217
218     /** ASTSignedDoubleLiteral represents a positive or negative Double number.
219         @attribute source String-representation (including '"').
220     */
221     SignedDoubleLiteral implements (("-")? Num_Double)=> SignedNumericLiteral =
222         (negative:["-"])? source:Num_Double;
223
224     /*=========================================================================*/
225     /*============================ LEXER RULES ================================*/
226     /*=========================================================================*/
227
228     // Definition of IDENTs (Java-style)
229     token Name
230         options{testLiterals=true;} = // check Literals first
231         ('a'..'z' | 'A'..'Z' | '_' | '$')
232         ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$')*;
233
234     // Character literals
235     token Char =
236         '\'' ( ESC | ~('\'' | '\n' | '\r' | '\\') ) '\'';
237
238     // String literals
239     token String =
240         '"' (ESC | ~('"' | '\\' | '\n' | '\r'))* '"';
241
242     // Hexadecimal digit (used inside Numbers)
243     protected token Hex_Digit =
244         ('0'..'9'|'A'..'F'|'a'..'f');
245
246     // Suffix of float numbers (used inside Numbers)
247     protected token Float_Suffix =
248         'f'|'F'|'d'|'D';
249
250     // Exponent for decimal numbers, used inside Numbers
251     protected token Decimal_Exponent =
252         ('e'|'E') ('+'|'-')? ('0'..'9')+;
253
254     // Exponent for hexadecimal numbers, used inside Numbers
255     protected token Hex_Exponent =
256         ('p'|'P') ('+'|'-')? ('0'..'9')+;
257
258     // Numbers
259     token Numbers
260         options{testLiterals=true;}
261         {boolean isDecimal = false, isHex = false, isDoubleDot = false;} =
262         ('.'
263           (
264             ('.')
265             |
266             ('.' '.')
267             |
```

```
268             (
269               ('0'..'9')+ (Decimal_Exponent)? (t=Float_Suffix)?
270               {
271                 if (t != null && t.getText().toUpperCase().indexOf('F')>=0){
272                   _ttype = Num_Float;
273                 }else{
274                   _ttype = Num_Double; // assume double
275                 }
276               }
277             )?
278           )
279         )
280
281         |
282
283         (
284           ('0' {isDecimal = true; _ttype = Num_Int;} // special case for '0'
285             (
286               // HEX
287               (('x'|'X') ((Hex_Digit)+ | ('.' (Hex_Digit)+) | Hex_Exponent)) =>
288               (
289                 ('x'|'X')
290                 // The decimal exponent and the float suffix look like
291                 // hex digits, hence the (...)* doesn't know when to stop
292                 // (-> ambig). ANTLR resolves it correctly by matching
293                 // immediately. It is therefore ok to hush warning.
294                 (options{warnWhenFollowAmbig=false;}: Hex_Digit)*
295                 {isHex = true;}
296               )
297
298               |
299               // FLOAT or DOUBLE with leading zero
300               (('0'..'9')+ ('.'|Decimal_Exponent|Float_Suffix)) => ('0'..'9')+
301
302               |
303               // OCTAL
304               ('0'..'7')+
305             )?
306           )
307
308           |
309           // NON-ZERO DECIMAL
310           (('1'..'9') ('0'..'9')*  {isDecimal = true; _ttype = Num_Int;})
311         )
312         {
313           if (LA(1)=='.' && LA(2)=='.'){
314             isDoubleDot = true;
315           }else{
316             isDoubleDot = false;
317           }
318         }
319         (
320           ('l'|'L') {_ttype = Num_Long;}
321
322           // only check to see if it's a float if looks like decimal so far
323           |
324           {isDecimal && !isDoubleDot}?
325           (
326             {isHex}?
327             ( // exponent is mandatory for floating point hex digits
328               '.' (Hex_Digit)* Hex_Exponent (t=Float_Suffix)?
329               |
330               Hex_Exponent (t=Float_Suffix)?
331               |
332               t=Float_Suffix
333             )
334             |
335             (
```

```
336                 '.' ('0'..'9')* (Decimal_Exponent)? (t=Float_Suffix)?
337                 |
338                 Decimal_Exponent (t=Float_Suffix)?
339                 |
340                 t=Float_Suffix
341               )
342             )
343             {
344               if (t != null && t.getText().toUpperCase().indexOf('F') >= 0){
345                 _ttype = Num_Float;
346               }else{
347                 _ttype = Num_Double; // assume double
348               }
349             }
350         )?;
351 
352     // Escape sequence -- note that this is protected; it can only be called
353     // from another lexer rule -- it will not ever directly return a token to
354     // the parser.
355     // There are various ambiguities hushed in this rule. The optional
356     // '0'...'9' digit matches should be matched here rather than letting them
357     // go back to String to be matched. ANTLR does the right thing by matching
358     // immediately; hence, it's ok to shut off the FOLLOW ambig warnings.
359     protected token ESC =
360         '\\'
361         (
362           'n' | 'r' | 't' | 'b' | 'f' | '"' | '\'' | '\\' | ('u')+
363           Hex_Digit Hex_Digit Hex_Digit Hex_Digit | '0'..'3'
364           (
365             options{warnWhenFollowAmbig=false;} : '0'..'7'
366             (options{warnWhenFollowAmbig=false;} : '0'..'7')?
367           )?
368           |
369           '4'..'7' (options{warnWhenFollowAmbig=false;} : '0'..'7')?
370         );
371 
372     // Whitespace -- ignored
373     token WS =
374         (
375           ' ' | '\t' | '\f'
376           |
377           (
378             options{generateAmbigWarnings=false;}:
379             "\r\n" | '\r' | '\n'
380           )
381           {newline();}
382         )+
383         {_ttype = Token.SKIP;};
384 
385 }
```

Quellcode C.2: MontiCore-Grammatik für Literale

C.2 Typen

```
1  package mc.types;                                                    MC
2  
3  /**
4   * This grammar defines Java compliant types. The scope of this grammar is to
5   * ease the reuse of type structures in Java-like sublanguages, e.g., by grammar
6   * inheritance or grammar embedment.
```

```
 * The grammar contains types from Java, e.g., primitives, void, types with
 * dimensions, reference types, generics, and type parameters.
 *
 * @author Martin Schindler
 * @version 1.3
*/
grammar Types extends mc.literals.Literals {

    /*=========================================================================*/
    /*============================== OPTIONS ==================================*/
    /*=========================================================================*/

    concept antlr {
        parser java {
            /**
             * Counts the number of LT of type parameters and type arguments.
             * It is used in semantic predicates to ensure the right number
             * of closing '>' characters; which actually may have been
             * either GT, SR (GTGT), or BSR (GTGTGT) tokens.
             */
            public int ltCounter = 0;
        }
    }

    /*=========================================================================*/
    /*======================== INTERFACE DEFINITIONS ==========================*/
    /*=========================================================================*/

    ast QualifiedName =
        method public String toString(){
          return mc.helper.NameHelper.dotSeparatedStringFromList(
              this.getParts());
        };

    ast PrimitiveType =
        method public String toString(){
          if (isBoolean()){
            return "boolean";
          }
          if (isByte()){
            return "byte";
          }
          if (isChar()){
            return "char";
          }
          if (isShort()){
            return "short";
          }
          if (isInt()){
            return "int";
          }
          if (isFloat()){
            return "float";
          }
          if (isLong()){
            return "long";
          }
          if (isDouble()){
            return "double";
          }
          return "";
        }
        method public boolean isBoolean(){
          return this.getPrimitive()==ASTConstantsTypes.BOOLEAN;
        }
        method public boolean isByte(){
          return this.getPrimitive()==ASTConstantsTypes.BYTE;
        }
```

```
75          method public boolean isChar(){
76            return this.getPrimitive()==ASTConstantsTypes.CHAR;
77          }
78          method public boolean isShort(){
79            return this.getPrimitive()==ASTConstantsTypes.SHORT;
80          }
81          method public boolean isInt(){
82            return this.getPrimitive()==ASTConstantsTypes.INT;
83          }
84          method public boolean isFloat(){
85            return this.getPrimitive()==ASTConstantsTypes.FLOAT;
86          }
87          method public boolean isLong(){
88            return this.getPrimitive()==ASTConstantsTypes.LONG;
89          }
90          method public boolean isDouble(){
91            return this.getPrimitive()==ASTConstantsTypes.DOUBLE;
92          };
93
94      ast ReferenceType astextends Type;
95
96      ast ArrayType astimplements ReferenceType =
97          dimensions:/int
98          componentType:Type;
99
100     ast QualifiedType astimplements ReferenceType;
101
102     /** ASTType defines types like primitives, Set, List, Collection, or
103         class types. It might also be an array or generic type.
104     */
105     interface Type extends TypeArgument, ReturnType;
106
107     /** ASTReferenceType defines a reference type like arrays or complex types.
108     */
109     interface ReferenceType;
110
111     /** ASTTypeArgument represents a type argument (generics).
112     */
113     interface TypeArgument;
114
115     /** ASTReturnType represents return types.
116     */
117     interface ReturnType;
118
119     /*=================================================================*/
120     /*============================ PARSER RULES ===========================*/
121     /*=================================================================*/
122
123     /** The ASTQualifiedName represents a possibly qualified name in the AST.
124         The different parts of a qualified name are separated by '.'; they are
125         stored in an ASTStringList.
126         @attribute parts A List of ASTStringList concludes all name parts
127     */
128     QualifiedName =
129         parts:Name (options{greedy=true;}: "." parts:Name)*;
130
131     /** The ASTArrayType represents an array of any type. The rule
132         ComplexArrayType itself treats all arrays except the primitive
133         ones. Especially it treats generic types.
134         @attribute componentType The kind of type which is used for the array.
135                                  Could be every complex type.
136         @attribute dimensions    Counts the number of '[]'
137     */
138     ComplexArrayType: ArrayType implements Type returns Type
139     {int ltLevel = ltCounter;} =
140         ret=ComplexType
141         (
142           {ltCounter==ltLevel}?
```

```
143             (
144               astscript{!(ComponentType=ret;);}
145               (
146                 options{greedy=true;}: "[" "]"
147                 {a.setDimensions(a.getDimensions()+1);}
148               )+
149             )
150             | /* let the dims for the enclosing type reference */
151         );
152
153     /** The ASTArrayType represents an array of any type. The rule
154         PrimitiveArrayType itself treats arrays of primitive types, such as
155         'int[]'.
156         @attribute componentType The kind of which is used for the array.
157                                  Could be every primitive type.
158         @attribute dimensions    Counts the number of '[]'
159     */
160     PrimitiveArrayType: ArrayType implements (PrimitiveType "["=>Type
161     returns Type =
162         ret=PrimitiveType
163         (
164           options{greedy=true;}:
165           astscript{!(ComponentType=ret;);}
166           (
167             options{greedy=true;}: "[" "]"
168             {a.setDimensions(a.getDimensions()+1);}
169           )+
170         )?;
171
172     /** ASTVoidType represents the return type "void".
173     */
174     VoidType implements ReturnType =
175         "void";
176
177     /** The BooleanType rule represents boolean primitive types (BOOLEAN).
178         An instance will be of type ASTPrimitiveType.
179     */
180     BooleanType:PrimitiveType =
181         primitive: ["boolean"];
182
183     /** The IntegralType rule represents integral primitive types
184         (BYTE, SHORT, INT, LONG, or CHAR).
185         An instance will be of type ASTPrimitiveType.
186     */
187     IntegralType:PrimitiveType =
188         primitive: ["byte" | "short" | "int" | "long" | "char"];
189
190     /** The FloatingPointType rule represents floating point primitive types
191         (LONG or DOUBLE).
192         An instance will be of type ASTPrimitiveType.
193     */
194     FloatingPointType:PrimitiveType =
195         primitive: ["float" | "double"];
196
197     /** The NumericType rule represents numeric types
198         (BOOLEAN, BYTE, CHAR, SHORT, INT, FLOAT, LONG, or DOUBLE).
199         An instance will be of type ASTPrimitiveType.
200     */
201     NumericType:PrimitiveType =
202         ret=FloatingPointType | ret=IntegralType;
203
204     /** ASTPrimitiveType represents every primitive type supported by Java.
205         @attribute primitive BOOLEAN, BYTE, CHAR, SHORT, INT, FLOAT, LONG,
206                              or DOUBLE
207     */
208     PrimitiveType implements Type =
209         ret=FloatingPointType | ret=IntegralType | ret=BooleanType;
210
```

```
211     /** The ComplexType rule represents a complex type (in contrast to
212         PrimitiveTypes; e.g. class or interface types). It handels
213         SimpleReferenceTypes and QualifiedTypes. An instance will be of type
214         ASTType.
215     */
216     ComplexType returns Type =
217         ret=SimpleReferenceType
218         (options{greedy=true;}: "." ret=QualifiedType->[ret])*;
219
220     /** ASTSimpleReferenceType represents types like class or interface types
221         which could have a qualified name like this: a.b.c<Arg>. The
222         qualification stored in the name (a.b) could be package or a type name.
223         The qualified name could contain type arguments only at the end.
224         a.b.c<Arg>.d would be one ASTSimpleReferenceType (a.b.c<Arg>) and one
225         ASTQualifiedType (d).
226         @attribute name          Name of the type
227                                  Note: Allthough the class name contains the
228                                  word 'simple', the name could be a qualified
229                                  one. So it is saved in an ASTStringList
230         @attribute typeArguments The types between '<...>'
231     */
232     SimpleReferenceType implements ReferenceType =
233         Name
234         (options{greedy=true;}: "." Name)*
235         (options{greedy=true;}: TypeArguments)?;
236
237     /** ASTQualifiedType represents types like class or interface types which
238         always have another ASTQualifiedType or ASTSimpleReferenceType as
239         qualification. So the qualification is in every case a type.
240         For example:
241         a.b.c<Arg>.d.e<Arg> would be one ASTSimpleReferenceType (a.b.c<Arg>)
242         and two ASTQualifiedType (d; e<Arg>)
243         @attribute name          Name of the type
244         @attribute typeArguments The types between '<...>'
245         @attribute qualification Another ASTQualifiedType or
246                                  ASTSimpleReferenceType.
247     */
248     QualifiedType [qualification:Type] =
249         Name
250         (options{greedy=true;}: TypeArguments)?;
251
252     /** ASTTypeArguments represents a list of generic arguments parenthesized
253         by '<...>'. It is also possible to nest type arguments in each other
254         like this <A<B<C>>>.
255         @attribute typeArguments List of arguments
256     */
257     TypeArguments {int currentLtLevel = 0;} =
258         {currentLtLevel = ltCounter;}
259         (
260           "<" {ltCounter++;}
261           typeArguments:TypeArgument
262           (options{greedy=true;}:
263             /*
264              * The following semantic predicates are needed to construct
265              * trees properly in case of ">>" or ">>>" tokens within nested
266              * TypeArguments (e.g., "var<O1<I1<M1>>, O2<I2>> a;").
267              */
268             {inputState.guessing !=0 || ltCounter == currentLtLevel + 1}?
269                 "," typeArguments:TypeArgument
270           )*
271           (
272             /*
273              * The token stream contains GT, GTGT, or GTGTGT tokens,
274              * so the parser has to expect one of these three possibilities.
275              * Furthermore the angle brackets are counted to check if there
276              * are some left open. After parsing the ltCounter has to be 0,
277              * what is checked by the predicate below.
278              */
```

```
279             options{generateAmbigWarnings=false;}:
280             ">" {ltCounter-=1;}
281             | ">>" {ltCounter-=2;}
282             | ">>>" {ltCounter-=3;}
283           )?
284         )
285         // This predicate checks if we have a valid angle bracket structure.
286         {(currentLtLevel != 0) || ltCounter == currentLtLevel}?;
287
288     /** ASTWildcardType represents a wildcard type in a type argument (generics)
289         It could also contain an upper- or lower bound.
290         @attribute upperBound Supertye of the type argument
291         @attribute lowerBound Subtype of the type argument
292     */
293     WildcardType implements TypeArgument =
294         "?" (
295               options{greedy=true;}:
296               ("extends" upperBound:Type) | ("super" lowerBound:Type)
297             )?;
298
299     /** ASTTypeParameters represents a list of generic parameter parenthesized
300         by '<...>' in type declarations (e.g., class-, interface-, method-, or
301         constructor declarations).
302         @attribute typeVariableDeclarations List of parameters
303     */
304     TypeParameters {int currentLtLevel = 0;} =
305         {currentLtLevel = ltCounter;}
306         (
307           options{greedy=true;}:
308           "<" {ltCounter++;}
309           typeVariableDeclarations:TypeVariableDeclaration
310           (
311             options{greedy=true;}:
312             "," typeVariableDeclarations:TypeVariableDeclaration
313           )*
314           ( // inlined typeArgumentsEnd
315             options{generateAmbigWarnings=false;}:
316             ">" {ltCounter-=1;}
317             | ">>" {ltCounter-=2;}
318             | ">>>" {ltCounter-=3;}
319           )?
320         )
321         // This predicate checks if we have a valid angle bracket structure
322         // (if we are at the "top level" of nested TypeArgument productions).
323         {(currentLtLevel != 0) || ltCounter == currentLtLevel}?
324         | /* nothing! (instead of optionality) */;
325
326     /** ASTTypeVariableDeclaration represents the generic variable declaration
327         in '<'...'>' (e.g., in front of method or constructor declarations or
328         behind the class or interface name).
329         E.g.: public <T extends SuperClass> void test(T t)
330         @attribute name        Name of the type variable
331         @attribute upperBounds Optional list of required super classes
332     */
333     TypeVariableDeclaration =
334         Name
335         (
336           options{generateAmbigWarnings=false;}:
337           "extends" upperBounds:ComplexType
338           ("&" upperBounds:ComplexType)*
339         )?;
340
341 }
```

Quellcode C.3: MontiCore-Grammatik für Typen

C.3 Gemeinsame Sprachanteile der UML/P

MC

```
1  package mc.umlp.common;
2
3  /**
4   * Grammar for common elements of the UML/P-Language-Family
5   *
6   * @author Martin Schindler
7   * @version 1.5
8   */
9  grammar Common extends mc.types.Types {
10
11     options {
12         parser lookahead=3
13         lexer lookahead=7
14     }
15
16
17     /*=========================================================================*/
18     /*======================== INTERFACE DEFINITIONS ==========================*/
19     /*=========================================================================*/
20
21     ast Stereotype =
22         method public boolean containsStereoValue(String name) {
23           return (getStereoValue(name) != null);
24         }
25         method public boolean containsStereoValue(String name, String value) {
26           for (ASTStereoValue sv : values) {
27             if (sv.getName().equals(name)
28                 && sv.getValue() != null && sv.getValue().equals(value)) {
29               return true;
30             }
31           }
32           return false;
33         }
34         method public ASTStereoValue getStereoValue(String name) {
35           for (ASTStereoValue sv : values) {
36             if (sv.getName().equals(name)) {
37               return sv;
38             }
39           }
40           return null;
41         };
42
43     ast StereoValue =
44         method public String getValue() {
45           try {
46             return
47                 mc.literals.LiteralsHelper.getInstance().decodeString(source);
48           }
49           catch (Exception e) {
50             return "";
51           }
52         }
53         method public void setValue(String value) {
54           this.source = '"' + value + '"';
55         };
56
57     ast Cardinality =
58         lowerBound:/int
59         upperBound:/int;
60
61     external InvariantContent;
62
63
```

```
64      /*=========================================================================*/
65      /*============================== GRAMMAR ==================================*/
66      /*=========================================================================*/
67
68      /** ASTStereotype represents Stereotypes in the UML/P
69          @attribute values List of Values of this Stereotype
70      */
71      Stereotype astextends /mc.umlp.common._ast.UMLPNode =
72          "<<" values:StereoValue ("," values:StereoValue)* ">>";
73
74      /** ASTStereoValue represents a Value of a Steretype in the UML/P
75          @attribute name  Name of the Stereotype-Value
76          @attribute source Source of the Stereotype (String including '"';
77                            use getValue() for decoded String)
78      */
79      StereoValue astextends /mc.umlp.common._ast.UMLPNode =
80          Name ("=" source:String)?;
81
82      /** ASTCardinality represents a Cardinality in the UML/P
83          @attribute many          True if "*" is set as Cardinality
84          @attribute lowerBound    Minimum number of associated Classes/Ojects
85          @attribute upperBound    Maximum number of associated Classes/Ojects
86          @attribute lowerBoundLit Lower bound as Literal
87          @attribute upperBoundLit Upper bound as Literal
88          @attribute noUpperLimit  True if no upper bound exists
89      */
90      Cardinality astextends /mc.umlp.common._ast.UMLPNode =
91          "["
92          (
93            many:["*"] {a.setLowerBound(0);a.setUpperBound(0);}
94            |
95            lowerBoundLit:IntLiteral
96            {
97              a.setLowerBound(a.getLowerBoundLit().getValue());
98              a.setUpperBound(a.getLowerBound());
99            }
100           (
101             ".."
102             (
103               upperBoundLit:IntLiteral
104               ({a.setUpperBound(a.getUpperBoundLit().getValue());})
105               |
106               noUpperLimit:["*"] {a.setUpperBound(0);}
107             )
108           )?
109         )
110         "]";
111
112     /** ASTCompleteness represents the completeness in the UML/P
113         Syntax: (left-completeness, right-completeness)
114         Interpretation:
115           CD: Diagramm: left: Types,      right: Assoziations
116               Types:    left: Attributes, right: Methods
117           OD: Diagramm: left: Objects,    right: Links
118               Objects:  left=right: Attributes
119           SC: Diagramm: left: States (state space coverage),
120                         right: Transitions
121               States:   left: Substates,
122                         right: Transitions of Substates
123           SD: Diagramm: left=right: Interactions:
124                                     (c) = <<match:complete>>
125                                     (...) = <<match:free>>
126               Objects:  left=right: Interactions of this Object
127         @attribute incomplete    true if left and right side are
128                                  incomplete (...)
129         @attribute complete      true if left and right side are
130                                  complete (c)
131         @attribute rightComplete true if only right side is complete (...,c)
```

```
132         @attribute leftComplete  true if only left is complete (c,...)
133     */
134     Completeness astextends /mc.umlp.common._ast.UMLPNode =
135         //the following complex lexer symbol construction is to prevent clash
136         //with ident "c"
137         complete:[COMPLETE:"(c)"]
138           | ("(" incomplete:[INCOMPLETE:"..."] ")") //separate brackets to avoid
139                                                    //lexer-symbol clash with SD/SC
140           | incomplete:[INCOMPLETE:"(...,...)"] | complete:[COMPLETE:"(c,c)"]
141           | rightComplete:[RIGHTCOMPLETE:"(...,c)"]
142           | leftComplete:[LEFTCOMPLETE:"(c,...)"];
143
144     /** ASTModifier represents a Modifier for Classes, Interfaces, Methods,
145         Constructors and Attributes in the UML/P
146         @attribute stereotype Optional Stereotype
147         @attribute public     true if Modifier is public
148                               (i.e. Modifier written as "public" or "+")
149         @attribute private    true if Modifier is private
150                               (i.e. Modifier written as "private" or "-")
151         @attribute protected  true if Modifier is protected
152                               (i.e. Modifier written as "protected" or "#")
153         @attribute final      true if Modifier is final
154                               (i.e. Modifier written as "final")
155         @attribute abstract   true if Modifier is abstract
156                               (i.e. Modifier written as "abstract")
157         @attribute local      true if Modifier is local
158                               (i.e. Modifier written as "local")
159         @attribute derived    true if Modifier is derived
160                               (i.e. Modifier written as "derived" or "/")
161         @attribute readonly   true if Modifier is readonly
162                               (i.e. Modifier written as "readonly" or "?")
163         @attribute static     true if Modifier is static
164                               (i.e. Modifier written as "static")
165     */
166     Modifier astextends /mc.umlp.common._ast.UMLPNode =
167         Stereotype?
168         (   Public:["public"]       | Public:[PUBLIC:"+"]   //type modifier
169           | Private:["private"]     | Private:[PRIVATE:"-"]
170           | Protected:["protected"] | Protected:[PROTECTED:"#"]
171           | Final:["final"]
172           | Abstract:["abstract"]
173           | Local:["local"]
174           | Derived:["derived"]     | Derived:[DERIVED:"/"] //attribute modifier
175           | Readonly:["readonly"]   | Readonly:[READONLY:"?"]
176           | Static:["static"]
177         )*;
178
179     /** ASTInvariant represents an Invariant in the UML/P
180         @attribute kind     Kind/Language of the Invariant
181         @attribute content  Content/condition of the Invariant
182     */
183     / Invariant astextends /mc.umlp.common._ast.UMLPNode =
184         (kind:Name ":")?
185         "[" content:InvariantContent(parameter kind) "]";
186
187 }
```

Quellcode C.4: MontiCore-Grammatik für Diagramm-übergreifende Elemente der UML/P

C.4 Klassendiagramme

MC

```
1  package mc.umlp.cd;
2
3  /**
4   * Grammar for UML/P-Classdiagrams
5   *
6   * @author Martin Schindler
7   * @version 1.5
8   */
9  grammar CD extends mc.umlp.common.Common {
10
11     options {
12         compilationunit CDDefinition
13     }
14
15     /*=========================================================================*/
16     /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
17     /*=========================================================================*/
18
19     /** ASTCDElement represents all elements of a UML Classdiagram */
20     abstract / CDElement astextends /mc.umlp.common._ast.UMLPNode;
21
22     /** ASTCDType represents all type-defining elements of a UML Classdiagram */
23     abstract / CDType extends CDElement;
24
25     /** ASTCDCallable represents a common abstract class for
26         ASTCDConstructor and ASTCDMethod */
27     abstract / CDCallable astextends /mc.umlp.common._ast.UMLPNode;
28
29     // for editor
30     ast CDAssociation =
31         arrow:/String
32         method public String getArrow(){
33           if (leftToRight) {
34             return "->";
35           } else if (rightToLeft) {
36             return "<-";
37           } else if (bidirectional) {
38             return "<->";
39           }
40           return "--";
41         };
42
43     // common attributes of ASTCDClass, ASTCDInterface, and ASTCDEnum
44     ast CDType =
45         Completeness
46         Modifier
47         Name
48         interfaces:/mc.types._ast.ASTReferenceTypeList
49         cDAttributes:/ASTCDAttributeList
50         cDMethods:/ASTCDMethodList;
51
52     // common attributes of ASTCDConstructor and ASTCDMethod
53     ast CDCallable =
54         Modifier
55         TypeParameters
56         Name
57         cDParameters:/ASTCDParameterList
58         exceptions:/mc.types._ast.ASTQualifiedNameList
59         body:/mc.ast.ASTNode;
60
61     external Value;
62     external Body;
63
```

```
64      /*=========================================================================*/
65      /*============================== GRAMMAR ==================================*/
66      /*=========================================================================*/
67
68      /** ASTCDDefinition represents a UML Classdiagram
69          @attribute completeness   Optional Comleteness of this Classdiagramm
70          @attribute stereotype     Optional Stereotype
71          @attribute name           Name of this Classdiagram
72          @attribute cDClasses      List of Classes of this Classdiagram
73          @attribute cDInterfaces   List of Interfaces of this Classdiagram
74          @attribute cDEnums        List of Enums of this Classdiagram
75          @attribute cDAssociations List of Associations of this Classdiagram
76          @attribute invaritants    List of Invaritants of this Classdiagram
77      */
78      / CDDefinition astextends /mc.umlp.common._ast.UMLPNode =
79          Completeness?
80          Stereotype?
81          "classdiagram" Name
82          "{"
83          (
84            (Completeness? Modifier "class")=>
85            cDClasses:CDClass
86            |
87            (Completeness? Modifier "interface")=>
88            cDInterfaces:CDInterface
89            |
90            (Completeness? Modifier "enum")=>
91            cDEnums:CDEnum
92            |
93            (Stereotype? ("association"|"aggregation"|"composition"))=>
94            cDAssociations:CDAssociation
95            |
96            invariants:Invariant
97          )*
98          "}";
99
100     /** ASTCDClass represents a Class in a UML Classdiagram
101         @attribute completeness   Optional Comleteness of this Class
102         @attribute modifier       Modifier of this Class
103         @attribute name           Name of this Class
104         @attribute typeParameters Generic type parameters of this Class
105         @attribute superclasses   List of Superclasses of this Class
106         @attribute interfaces     List of Interfaces implemented by this Class
107         @attribute cDConstructors List of Constructors of this Class
108         @attribute cDMethods      List of Methods of this Class
109         @attribute cDAttributes   List of Attributes of this Class
110     */
111     / CDClass astextends CDType =
112         Completeness?
113         Modifier
114         "class" Name (options{greedy=true;}: TypeParameters)?
115         ("extends" superclasses:ReferenceType
116                    ("," superclasses:ReferenceType)*)?
117         ("implements" interfaces:ReferenceType
118                    ("," interfaces:ReferenceType)*)?
119         (
120           ("{" (
121             (Modifier Type Name ("=" | ";"))
122             => cDAttributes:CDAttribute
123             |
124             (Modifier (options{greedy=true;}: TypeParameters)? Name "(")
125             => cDConstructors:CDConstructor
126             |
127             cDMethods:CDMethod
128           )* "}")
129           |
130           ";"
131         );
```

```
132     /** ASTCDInterface represents an Interface in a UML Classdiagram
133         @attribute completeness   Optional Comleteness of this Interface
134         @attribute modifier       Modifier of this Interface
135         @attribute name           Name of this Interface
136         @attribute typeParameters Generic type parameters of this Interface
137         @attribute interfaces     List of Interfaces extended by this Interface
138         @attribute cDMethods      List of Methods of this Interface
139         @attribute cDAttributes   List of Attributes of this Interface
140     */
141     / CDInterface astextends CDType =
142         Completeness?
143         Modifier
144         "interface" Name (options{greedy=true;}: TypeParameters)?
145         ("extends" interfaces:ReferenceType
146                    ("," interfaces:ReferenceType)*)?
147         (
148           ("{" (
149             (Modifier Type Name ("=" | ";"))
150             => cDAttributes:CDAttribute
151             |
152             cDMethods:CDMethod
153           )* "}")
154           |
155           ";"
156         );
157
158     /** ASTCDEnum represents an Enumeration (enum type) in a UML Classdiagram
159         @attribute completeness    Optional Comleteness of this Enum
160         @attribute modifier        Modifier of this Enum
161         @attribute name            Name of this Enum
162         @attribute interfaces      List of Interfaces implemented by this Enum
163         @attribute cDEnumConstants List of the Enum Constants
164         @attribute cDConstructors  List of Constructors of this Enum
165         @attribute cDMethods       List of Methods of this Enum
166         @attribute cDAttributes    List of Attributes of this Enum
167     */
168     / CDEnum astextends CDType =
169         Completeness?
170         Modifier
171         "enum" Name
172         ("implements" interfaces:ReferenceType
173                       ("," interfaces:ReferenceType)*)?
174         (
175             ("{"(
176               cDEnumConstants:CDEnumConstant
177               ("," cDEnumConstants:CDEnumConstant)* ";"
178               (
179                 (Modifier Type Name ("=" | ";"))
180                 => cDAttributes:CDAttribute
181                 |
182                 (Modifier ("<" TypeVariableDeclaration
183                 ("," TypeVariableDeclaration)* (">" | ">>" | ">>>")?)?
184                 Name "(")
185                 => cDConstructors:CDConstructor
186                 |
187                 cDMethods:CDMethod
188               )*
189             )?"}")
190             |
191             ";"
192         );
193
194     /** ASTCDEnumConstant represents a Constant of an Enumeration (enum type)
195         in a UML Classdiagram
196         @attribute name             Name of this Constant
197         @attribute cDEnumParameters List of optional parameters of this Constant
198     */
199     / CDEnumConstant astextends /mc.umlp.common._ast.UMLPNode =
```

```
200         Name ("("
201           cDEnumParameters:CDEnumParameter
202           ("," cDEnumParameters:CDEnumParameter)*
203         ")")?;
204
205     /** ASTCDEnumParameter represents a Parameter of an Enumeration Constant
206         @attribute Value Value of this Parameter
207     */
208     / CDEnumParameter astextends /mc.umlp.common._ast.UMLPNode =
209         Value;
210
211     /** ASTCDMethod represents a Method of a Class or Interface
212         @attribute modifier       Modifier of this Method
213         @attribute typeParameters Generic type parameters of this Method
214         @attribute returnType     Return-Type of the return value of this Method
215         @attribute name           Name of this Method
216         @attribute cDParameters   List of Parameters of this Method
217         @attribute exceptions     List of Exceptions thrown by this Method
218         @attribute body           Body of this Method
219     */
220     / CDMethod astextends CDCallable =
221         Modifier
222         (options{greedy=true;}: TypeParameters)?
223         ReturnType
224         Name "(" (
225           cDParameters:CDParameter
226           (options{greedy=true;}: "," cDParameters:CDParameter)*
227         )? ")"
228         (("throws")=> "throws"
229           exceptions:QualifiedName
230           (options{greedy=true;}: "," exceptions:QualifiedName)*)?
231         (("{")=>Body | ";");
232
233     /** ASTCDConstructor represents a Constructor of a Class or Interface
234         @attribute modifier       Modifier of this Constructor
235         @attribute typeParameters Generic type parameters of this Constructor
236         @attribute name           Name of this Constructor
237         @attribute cDParameters   List of Parameters of this Constructor
238         @attribute exceptions     List of Exceptions thrown by this Constructor
239         @attribute body           Body of this Constructor
240     */
241     / CDConstructor astextends CDCallable =
242         Modifier
243         (options{greedy=true;}: TypeParameters)?
244         Name
245         "(" (cDParameters:CDParameter ("," cDParameters:CDParameter)*)? ")"
246         (("throws")=> "throws"
247           exceptions:QualifiedName
248           (options{greedy=true;}: "," exceptions:QualifiedName)*)?
249         (("{")=> Body | ";");
250
251     /** ASTCDParameter represents a Parameter of a Constructor or Method
252         @attribute type Type of this Parameter
253         @attribute name Name of this Parameter
254     */
255     / CDParameter astextends /mc.umlp.common._ast.UMLPNode =
256         Type Name;
257
258     /** ASTCDAttribute represents an Attribute of a Class or Interface
259         @attribute modifier Modifier of this Attribute
260         @attribute type     Type of this Attribute
261         @attribute name     Name of this Attribute
262         @attribute value    Value of this Attribute
263     */
264     / CDAttribute astextends /mc.umlp.common._ast.UMLPNode =
265         Modifier Type Name
266         ("=" Value)? ";";
267
```

```
268     /** ASTQualifier represents a Qualifier of an Association
269         @attribute name      Abstract name of the Qualifier if not a type
270         @attribute type      Type of the Qualifier if not an abstract name
271                              (i.e. attribute name)
272     */
273     / CDQualifier astextends /mc.umlp.common._ast.UMLPNode =
274         (Name)=> Name | Type;
275
276     /** ASTCDAssociation represents an Association between Classes or Interfaces
277         @attribute stereotype          Optional Stereotype
278         @attribute Association         True if Association is of type "association"
279         @attribute Aggregation         True if Association is of type "aggregation"
280         @attribute Composition         True if Association is of type "composition"
281         @attribute type                Type of the Association (Association,
282                                        Aggregation, or Composition)
283         @attribute derived             True if this is a derived Association
284         @attribute name                Name of this Association
285         @attribute leftModifier        Optional left side Modifier
286         @attribute leftCardinality     Cardinality of the left side of this
287                                        Association
288         @attribute leftReferenceName   Name of the Class or Interface on the left
289                                        side of this Association
290         @attribute leftQualifier       Qualifier of the left side of this
291                                        Association
292         @attribute leftRole            Role of the Class or Interface on the left
293                                        side of this Association
294         @attribute leftToRight         True if Association is navigable from
295                                        left to right ("->")
296         @attribute rightToLeft         True if Association is navigable from
297                                        right to left ("<-")
298         @attribute bidirectional       True if Association is navigable in
299                                        both directions ("<->")
300         @attribute simple              True if navigation of Association is
301                                        not specified  ("--")
302         @attribute rightRole           Role of the Class or Interface on the right
303                                        side of this Association
304         @attribute rightQualifier      Qualifier of the right side of this
305                                        Association
306         @attribute rightReferenceName  Name of the Class or Interface on the right
307                                        side of this Association
308         @attribute rightCardinality    Cardinality of the right side of this
309                                        Association
310         @attribute rightModifier       Optional right side Modifier
311     */
312     / CDAssociation astextends CDElement =
313         Stereotype?
314         (
315           Association:["association"]
316           | Aggregation:["aggregation"]
317           | Composition:["composition"]
318         )
319         (options{greedy=true;}: Derived:[DERIVED:"/"])?
320         //predicate necessary to prevent clash with LeftReference
321         ((Name Modifier Cardinality? QualifiedName)=> Name |)
322         leftModifier:Modifier
323         leftCardinality:Cardinality?
324         leftReferenceName:QualifiedName
325         ("[" leftQualifier:CDQualifier "]")?
326         ("(" leftRole:Name ")")?
327         (
328           leftToRight:["->"]
329           | rightToLeft:["<-"]
330           | bidirectional:["<->"]
331           | simple:["--"]
332         )
333         ("(" rightRole:Name ")")?
334         ("[" rightQualifier:CDQualifier "]")?
335         rightReferenceName:QualifiedName
```

```
336         rightCardinality:Cardinality?
337         rightModifier:Modifier ";";
338
339 }
```

Quellcode C.5: MontiCore-Grammatik für Klassendiagramme

C.5 Objektdiagramme

MC

```
1  package mc.umlp.od;
2
3  /**
4   * Grammar for UML/P-Objectdiagrams
5   *
6   * @author Martin Schindler
7   * @version 1.5
8   */
9  grammar OD extends mc.umlp.common.Common {
10
11     options {
12         compilationunit ODDefinition
13     }
14
15     /*=========================================================================*/
16     /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
17     /*=========================================================================*/
18
19     /** ASTODElement represents all Elements of a UML Objectdiagram
20     */
21     abstract / ODElement astextends /mc.umlp.common._ast.UMLPNode;
22
23     // for editor
24     ast ODLink =
25         arrow:/String
26         leftReferenceNamesString:/String
27         rightReferenceNamesString:/String
28         method public String getArrow(){
29           if (leftToRight) {
30             return "->";
31           } else if (rightToLeft) {
32             return "<-";
33           } else if (bidirectional) {
34             return "<->";
35           }
36           return "--";
37         }
38         method public String getLeftReferenceNamesString(){
39           return mc.helper.NameHelper.separatedStringFromList(
40               leftReferenceNames, ", ");
41         }
42         method public String getRightReferenceNamesString(){
43           return mc.helper.NameHelper.separatedStringFromList(
44               rightReferenceNames, ", ");
45         };
46
47     external Value;
48
49     /*=========================================================================*/
50     /*============================== GRAMMAR ==================================*/
51     /*=========================================================================*/
52
```

```
53      /** ASTODDefinition represents a UML Objectdiagram
54          @attribute completeness Optional Comleteness of this Objectdiagramm
55          @attribute stereotype   Optional Stereotype
56          @attribute name         Name of this Objectdiagram
57          @attribute oDObjects    List of Objects of this Objectdiagram
58          @attribute oDLinks      List of Links of this Objectdiagram
59          @attribute invaritants  List of Invaritants of this Objectdiagram
60      */
61      / ODDefinition astextends /mc.umlp.common._ast.UMLPNode =
62          Completeness?
63          Stereotype?
64          "objectdiagram" Name
65          "{"
66          (
67            (Completeness? Modifier
68            (Name (":" ReferenceType)? | (":" ReferenceType)) ("{"|";"))=>
69            oDObjects:ODObject
70            |
71            (Stereotype? ("link"|"aggregation"|"composition"))=>
72            oDLinks:ODLink
73            |
74            invariants:Invariant
75          )*
76          "}";
77
78      /** ASTODObject represents an Object in a UML Objectdiagram
79          @attribute completeness Optional Comleteness of this Object
80          @attribute modifier     Modifier of this Object
81          @attribute name         Optional Name of this Object
82          @attribute type         Optional Type of this Object
83          @attribute oDAttributes List of Attributes of this Object
84      */
85      / ODObject astextends ODElement =
86          Completeness?
87          Modifier
88          (Name (":" type:ReferenceType)? | (":" type:ReferenceType))
89          (
90            ("{" oDAttributes:ODAttribute* "}")
91            |
92            ";"
93          );
94
95      /** ASTODAttribute represents an Attribute of an Object
96          @attribute modifier Modifier of this Attribute
97          @attribute type     Type of this Attribute
98          @attribute name     Name of this Attribute
99          @attribute value    Value of this Attribute
100     */
101     / ODAttribute astextends /mc.umlp.common._ast.UMLPNode =
102         Modifier
103         Type?
104         Name
105         ("=" Value)? ";";
106
107     /** ASTODLinkQualifier represents a reference to a Qualifier of Links
108         between Objects. The reference can be either a concrete attribute
109         name or a concrete value.
110         @attribute name     Name of a referenced attribute (if set)
111         @attribute value    Concrete value of this qualifier (if set)
112     */
113     / ODLinkQualifier astextends /mc.umlp.common._ast.UMLPNode =
114         (Name)=> Name | Value;
115
116     /** ASTODLink represents a Link between Objects
117         @attribute stereotype          Optional Stereotype
118         @attribute Link                True if Link is of type "link"
119         @attribute Aggregation         True if Link is of type "aggregation"
120         @attribute Composition         True if Link is of type "composition"
```

```
121         @attribute derived              True if this is a derived Link
122         @attribute Name                 Name of the Association of this Link
123         @attribute leftModifier         Optional left side Modifier
124         @attribute leftReferenceNames   List of References of the Objects on the
125                                         left side of this Link
126         @attribute leftQualifier        Qualifier of the left side of this Link
127         @attribute leftRole             Role of Objects on the Links left side
128         @attribute leftToRight          True if Link is navigable from
129                                         left to right ("->")
130         @attribute rightToLeft          True if Link is navigable from
131                                         right to left ("<-")
132         @attribute bidirectional        True if Link is navigable in
133                                         both directions ("<->")
134         @attribute simple               True if navigation of Link is
135                                         not specified ("--")
136         @attribute rightRole            Role of Objects on the Links right side
137         @attribute rightQualifier       Qualifier of the right side of this Link
138         @attribute rightReferenceNames  List of References of the Objects on the
139                                         right side of this Link
140         @attribute rightModifier        Optional right side Modifier
141     */
142     / ODLink astextends ODElement =
143         Stereotype?
144         (
145           Link:["link"]
146           | Aggregation:["aggregation"]
147           | Composition:["composition"]
148         )
149         (options{greedy=true;}: Derived:[DERIVED:"/"])?
150         //predicate necessary to prevent clash with LeftReference
151         ((Name Modifier QualifiedName)=> Name |)
152         leftModifier:Modifier
153         leftReferenceNames:QualifiedName ("," leftReferenceNames:QualifiedName)*
154         ("[" leftQualifier:ODLinkQualifier "]")?
155         ("(" leftRole:Name ")")?
156         (
157           leftToRight:["->"]
158           | rightToLeft:["<-"]
159           | bidirectional:["<->"]
160           | simple:["--"]
161         )
162         ("(" rightRole:Name ")")?
163         ("[" rightQualifier:ODLinkQualifier "]")?
164         rightReferenceNames:QualifiedName ("," rightReferenceNames:QualifiedName)*
165         rightModifier:Modifier
166         ";";
167
168 }
```

Quellcode C.6: MontiCore-Grammatik für Objektdiagramme

C.6 Statecharts

MC

```
1 package mc.umlp.sc;
2
3 /**
4  * Grammar for UML/P-Statecharts
5  *
6  * @author Martin Schindler
7  * @version 1.5
8  */
```

```
9  grammar SC extends mc.umlp.common.Common {
10
11     options {
12         compilationunit SCDefinition
13     }
14
15     /*=========================================================================*/
16     /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
17     /*=========================================================================*/
18
19     /** ASTSCElement represents all Elements of a UML Statechart diagram
20     */
21     abstract / SCElement astextends /mc.umlp.common._ast.UMLPNode;
22
23     /** ASTSCEvent represents Events of Transitions in a UML Statechart diagram
24     */
25     abstract / SCEvent astextends /mc.umlp.common._ast.UMLPNode;
26
27     external Statements;
28     external Expression;
29
30     /*=========================================================================*/
31     /*============================== GRAMMAR ==================================*/
32     /*=========================================================================*/
33
34     /** ASTSCDefinition represents a UML Statechart diagram
35         @attribute completeness  Optional Comleteness of this Statechart diagram
36         @attribute stereotype    Optional Stereotype
37         @attribute name          Name of this Statechart diagram
38         @attribute className     Optional name of the class modeled with this
39                                  Statechart diagram (if it differs from the name
40                                  of the Statechart diagram)
41         @attribute sCMethod      Optional method for method Statecahrt diagrams
42         @attribute sCStates      List of States of this Statechart diagram
43         @attribute sCTransitions List of Transitions of this Statechart diagram
44         @attribute sCCode        List of Code-blocks of this Statechart diagram
45     */
46     / SCDefinition astextends /mc.umlp.common._ast.UMLPNode =
47         Completeness?
48         Stereotype?
49         "statechart" Name
50         ((QualifiedName "(")=> SCMethod | className:ReferenceType)?
51         "{"
52           (
53             (Completeness? SCModifier "state")=>
54             sCStates:SCState
55             |
56             (Stereotype? QualifiedName "->")=>
57             sCTransitions:SCTransition
58             |
59             SCCode
60           )*
61         "}";
62
63     /** ASTSCMethod represents the Method of a method Statechart diagram
64         @attribute name         Name of this Method
65         @attribute sCParameters Optional list of parameters
66     */
67     / SCMethod astextends /mc.umlp.common._ast.UMLPNode =
68         name:QualifiedName
69         "(" (
70           sCParameters:SCParameter ("," sCParameters:SCParameter)*
71         )? ")";
72
73     /** ASTSCParameter represents a Parameter of a Method
74         @attribute type Type of this Parameter
75         @attribute name Name of this Parameter
76     */
```

```
77      / SCParameter astextends /mc.umlp.common._ast.UMLPNode =
78          Type Name;
79
80      /** ASTSCAction represents the general part of do-, entry-, and exit-Actions
81          @attribute preCondition  Pre-Condition for this Action
82          @attribute Statements    Statements of this Action
83          @attribute postCondition Post-Condition for this Action
84      */
85      / SCAction astextends /mc.umlp.common._ast.UMLPNode =
86          (((Name ":") | "[")=> preCondition:Invariant)?
87          (
88            ";"
89            |
90            ("/" Statements
91              (((Name ":") | "[")=> postCondition:Invariant ";")?)
92          );
93
94      /** ASTSCDoAction represents a do-Action of States
95          @attribute sCAction Action for this do-Action
96      */
97      / SCDoAction astextends /mc.umlp.common._ast.UMLPNode =
98          "do" SCAction;
99
100     /** ASTSCEntryAction represents an entry-Action of States
101         @attribute sCAction Action for this entry-Action
102     */
103     / SCEntryAction astextends /mc.umlp.common._ast.UMLPNode =
104         "entry" SCAction;
105
106     /** ASTSCExitAction represents an exit-Action of States
107         @attribute sCAction Action for this exit-Action
108     */
109     / SCExitAction astextends /mc.umlp.common._ast.UMLPNode =
110         "exit" SCAction;
111
112     /** ASTSCModifier represents a Modifier for a State
113         @attribute stereotype Optional Stereotpye
114         @attribute initial    true if State is initial
115         @attribute final      true if State is final
116         @attribute local      true if State is local
117     */
118     //own modifier needed as common-modifier causes nondeterminism
119     //because of derived "/"
120     / SCModifier astextends /mc.umlp.common._ast.UMLPNode =
121         Stereotype?
122         (Initial:["initial"] | Final:["final"] | Local:["local"])*;
123
124     /** ASTSCState represents a (hierarchical) State in a UML Statechart diagram
125         @attribute completeness        Optional Comleteness of this State
126         @attribute sCModifier          Modifier of this State
127         @attribute name                Name of this State
128         @attribute invariant           Invariant for this State
129         @attribute sCEntryAction       entry-Action for this State
130         @attribute sCDoAction          do-Action for this State
131         @attribute sCExitAction        exit-Action for this State
132         @attribute sCStates            List of States included in this State
133         @attribute sCTransitions       List of Transitions included in this State
134         @attribute sCCode              List of Code-blocks included in this State
135         @attribute sCInternTransitions List of Intern Transitions for this State
136     */
137     / SCState astextends SCElement =
138         Completeness?
139         SCModifier
140         "state" Name
141         (
142           ("{"
143             (((Name ":") | "[" )=> Invariant ";")?
144             SCEntryAction?
```

```
145             SCDoAction?
146             SCExitAction?
147             (
148               (Completeness? SCModifier "state")=>
149               sCStates:SCState
150               |
151               (Stereotype? QualifiedName "->")=>
152               sCTransitions:SCTransition
153               |
154               ("code")=>
155               SCCode
156               |
157               sCInternTransitions:SCInternTransition
158             )*
159           "}")
160           |
161           ";"
162       );
163
164     /** ASTSCInternTransition represents an Intern Transition of a State in a
165         UML Statechart diagram
166         @attribute stereotype       Optional Stereotype
167         @attribute sCTransitionBody Body of this Transition
168     */
169     / SCInternTransition astextends /mc.umlp.common._ast.UMLPNode =
170         Stereotype? "->" ":"? SCTransitionBody;
171
172     /** ASTSCTransition represents a Transition between two States in a
173         UML Statechart diagram
174         @attribute stereotype       Optional Stereotype
175         @attribute sourceName       Name of the source of this Transition
176         @attribute targetName       Name of the target of this Transition
177         @attribute sCTransitionBody Body of this Transition
178     */
179     / SCTransition astextends SCElement =
180         Stereotype?
181         sourceName:QualifiedName "->" targetName:QualifiedName
182         (
183           (":" SCTransitionBody)
184           |
185           ";"
186         );
187
188     /** ASTSCTransitionBody represents the Body of a Transition in a
189         UML Statechart diagram
190         @attribute preCondition  Pre-Condition of this Transition Body
191         @attribute sCEvent       Event for this Transition Body to take place
192         @attribute statements    Actions of this Transition Body
193         @attribute postCondition Post-Condition of this Transition Body
194     */
195     / SCTransitionBody astextends /mc.umlp.common._ast.UMLPNode =
196         (((Name ":") | "[")=> preCondition:Invariant)?
197         SCEvent?
198         (
199           ("/" Statements
200               (((Name ":") | "[")=> postCondition:Invariant ";")?)
201           |
202           ";"
203         );
204
205     /** ASTSCMethodOrExceptionCall represents a call of a method or exception
206         of a Transition in a UML Statechart diagram
207         @attribute name        Name of this method call
208         @attribute sCArguments Optional Arguments of this method call
209     */
210     / SCMethodOrExceptionCall extends SCEvent =
211         name:QualifiedName (("(")=> SCArguments)?;
212
```

```
213     /** ASTSCReturnStatement represents a return statement of a Transition
214         in a UML Statechart diagram
215         @attribute incomplete True if return statement is
216                                  incomplete/underspecified
217         @attribute expression Expression of this return statement
218     */
219     / SCReturnStatement extends SCEvent =
220         "return"
221         (
222           ("(" Expression ")") // brackets needed to prevent clash with "/"
223           | incomplete:[INCOMPLETE:"..."]
224         )?;
225
226     /** ASTSCArguments represents Arguments of an Event in a UML Statechart
227         diagram
228         @attribute incomplete  True if Arguments are incomplete (underspecified)
229         @attribute expressions Specified Arguments as a list of Expressions
230     */
231     / SCArguments astextends /mc.umlp.common._ast.UMLPNode =
232         ("(" "..."=> ("(" incomplete:[INCOMPLETE:"..."] ")")
233         | ("(" ")"=> ("(" ")")
234         | ("(" expressions:Expression ("," expressions:Expression)* ")");
235
236     /** ASTSCCode represents user added code to the Statechart diagram or to
237         States
238         @attribute statements The code added by the user
239     */
240     / SCCode astextends SCElement =
241         "code" Statements;
242
243 }
```

Quellcode C.7: MontiCore-Grammatik für Statecharts

C.7 Sequenzdiagramme

```
1   package mc.umlp.sd;                                                    MC
2
3   /**
4    * Grammar for UML/P-Sequencediagrams
5    *
6    * @author Martin Schindler
7    * @version 1.5
8    */
9   grammar SD extends mc.umlp.common.Common {
10
11      options {
12          compilationunit SDDefinition
13      }
14
15      /*=========================================================================*/
16      /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
17      /*=========================================================================*/
18
19      /** ASTSDElement represents all Elements of a UML Sequencediagram
20      */
21      abstract / SDElement astextends /mc.umlp.common._ast.UMLPNode;
22
23      /** ASTSDActivity represents all Activities of a UML Sequencediagram
24      */
25      interface / SDActivity;
```

```
26      /** ASTSDInteraction represents Interactions of a UML Sequencediagram
27      */
28      abstract / SDInteraction extends SDElement;
29
30      /** ASTSDCallMessage represents the call of a message of an Interaction
31      */
32      abstract / SDCallMessage astextends /mc.umlp.common._ast.UMLPNode;
33
34      /** ASTSDReturnMessage represents a return message of an Interaction
35      */
36      abstract / SDReturnMessage astextends /mc.umlp.common._ast.UMLPNode;
37
38      // common attributes of SDCallInteraction and SDReturnInteraction
39      ast SDInteraction =
40          sourceReferenceName:Name
41          targetReferenceName:Name
42          Stereotype;
43
44      external Expression;
45
46      /*=================================================================*/
47      /*============================== GRAMMAR ==========================*/
48      /*=================================================================*/
49
50      /** ASTSDDefinition represents a UML Sequencediagram
51          @attribute completeness Optional Comleteness of this Sequencediagramm
52          @attribute stereotype   Optional Stereotype
53          @attribute name         Name of this Sequencediagram
54          @attribute sDObjects    List of the objects of this Sequencediagram
55          @attribute sDActivities List of the activities of this Sequencediagram
56                                  (ActivityStart and ActivityEnd, Interactions,
57                                  and Conditions)
58      */
59      / SDDefinition astextends /mc.umlp.common._ast.UMLPNode =
60          Completeness?
61          Stereotype?
62          "sequencediagram" Name
63          "{"
64            (
65              ((Name? ("->" | "<-")) | "{" | "start" | "end" | ("<" Name))=>
66              sDActivities:SDActivity
67              |
68              sDObjects:SDObject
69            )*
70          "}";
71
72      /** ASTSDObject represents an Object in a UML Sequencediagram
73          @attribute completeness Optional Comleteness of this Object
74          @attribute modifier     Modifier of this Object
75          @attribute name         Optional name of this Object
76          @attribute type         Optional type of this Object
77      */
78      / SDObject astextends SDElement =
79          Completeness?
80          Modifier
81          (Name (":" type:ReferenceType)? | (":" type:ReferenceType)) ";";
82
83      /** ASTSDComplexActivity represents a list of Activities in a UML
84          Sequencediagram
85          @attribute sDActivities List of Activities
86      */
87      / SDComplexActivity astextends /mc.umlp.common._ast.UMLPNode
88      implements SDActivity =
89          "{" sDActivities:SDActivity+ "}";
90
91      /** ASTSDActivityStart represents the beginning of an Activation-Box
92          (Activity) in a UML Sequencediagram
93          @attribute name          Name of this Activity
```

```
94          @attribute referenceName Reference of the Object associated with this
95                                   Activity (Object-Name or -Type)
96      */
97      / SDActivityStart astextends /mc.umlp.common._ast.UMLPNode
98      implements SDActivity =
99          "start" Name referenceName:Name ";";
100
101     /** ASTSDActivityEnd represents the end of an Activation-Box (Activity)
102         in a UML Sequencediagram
103         @attribute names List of Names of finished Activities
104     */
105     / SDActivityEnd astextends /mc.umlp.common._ast.UMLPNode
106     implements SDActivity =
107         "end" names:Name ("," names:Name)* ";";
108
109     /** ASTSDCallInteraction represents an Interaction Call
110         ("->"; new or MethodCall) in a UML Sequencediagram
111         @attribute sourceReferenceName Reference of the Object which is the source
112                                        of this Interaction (Object-Name or -Type);
113                                        can be null in case the call is coming from
114                                        the environment
115         @attribute activityStart       Optional Name of the Activation-Box
116                                        (Activity) started with this Interaction
117         @attribute targetReferenceName Reference of the Object which is the target
118                                        of this Interaction (Object-Name or -Type)
119         @attribute stereotype          Optional Stereotype of this Interaction
120         @attribute sDCallMessage       Contains the message of the Interaction
121     */
122     / SDCallInteraction extends (Name? "->")=>SDInteraction
123     implements (Name? "->")=>SDActivity =
124         sourceReferenceName:Name? "->" activityStart:Name? targetReferenceName:Name
125         ":"
126         (("<<")=> Stereotype)?
127         SDCallMessage ";";
128
129     /** ASTSDReturnInteraction represents an Interaction Return
130         ("<-"; return or Exception) in a UML Sequencediagram
131         @attribute sourceReferenceName Reference of the Object which is the source
132                                        of this Interaction (Object-Name or -Type)
133         @attribute activityEnd         Optional Name of the Activation-Box
134                                        (Activity) stoped with this Interaction
135         @attribute targetReferenceName Reference of the Object which is the target
136                                        of this Interaction (Object-Name or -Type);
137                                        can be null in case the return is going to
138                                        the environment
139         @attribute stereotype          Optional Stereotype of this Interaction
140         @attribute sDReturnMessage     Optional message of the Interaction
141     */
142     / SDReturnInteraction extends (Name? "<-")=>SDInteraction
143     implements (Name? "<-")=>SDActivity =
144         targetReferenceName:Name? "<-" activityEnd:Name? sourceReferenceName:Name
145         (
146           ":"
147           (("<<")=> Stereotype)?
148           SDReturnMessage
149         )?
150         ";";
151
152     /** ASTSDMethodCall represents a method call of an Interaction in a
153         UML Sequencediagram
154         @attribute methodName  Name of this method call
155         @attribute sDArguments Optional Arguments of this method call
156     */
157     / SDMethodCall extends SDCallMessage =
158         methodName:QualifiedName SDArguments?;
159
160     /** ASTSDNewStatement represents a new statement of an Interaction
161         in a UML Sequencediagram
```

```
162         @attribute type         Type of the new object to create
163         @attribute sDArguments Optional Arguments of the constructor
164     */
165     / SDNewStatement extends SDCallMessage =
166         "new" type:ReferenceType SDArguments?;
167
168     /** ASTSDException represents an exception of an Interaction in
169         a UML Sequencediagram
170         @attribute type           Exception-Type
171         @attribute sDArguments   Optional Arguments of the exception
172     */
173     / SDException extends SDReturnMessage =
174         type:ReferenceType SDArguments?;
175
176     /** ASTSDReturnStatement represents a return statement of an
177         Interaction in a UML Sequencediagram
178         @attribute incomplete True if return statement is incomplete
179                               (underspecified)
180         @attribute expression Expression of this return statement
181     */
182     / SDReturnStatement extends SDReturnMessage =
183         "return" (
184           {!LT(1).getText().startsWith("...") && !LT(1).getText().startsWith(";")}?
185           Expression
186           | incomplete:[INCOMPLETE:"..."]
187         )?;
188
189     /** ASTSDArguments represents Arguments of an Interaction/Message in a
190         UML Sequencediagram
191         @attribute incomplete  True if Arguments are incomplete (underspecified)
192         @attribute expressions Specified Arguments as a list of Expressions
193     */
194     / SDArguments astextends /mc.umlp.common._ast.UMLPNode =
195         ("(" "..."")=> ("(" incomplete:[INCOMPLETE:"..."] ")")
196         | ("(" ")"")=> ("(" ")")
197         | ("(" expressions:Expression ("," expressions:Expression)* ")");
198
199     /** ASTSDCondition represents a Condition in a UML Sequencediagram
200         @attribute range     List of references included in this Condition
201                              (Object-Name or -Type)
202         @attribute invariant The Condition itself
203     */
204     / SDCondition astextends /mc.umlp.common._ast.UMLPNode implements SDActivity =
205         "<" range:Name ("," range:Name)* ":" Invariant ">" ";";
206
207 }
```

Quellcode C.8: MontiCore-Grammatik für Sequenzdiagramme

C.8 Testspezifikationssprache

MC

```
1  package mc.umlp.tc;
2
3  /**
4   * Grammar for Testcase-Descriptions with UML/P-Diagrams
5   *
6   * @author Martin Schindler
7   * @version 1.5
8   */
9  grammar TC extends mc.umlp.common.Common {
10
```

```
11      options {
12          compilationunit TCDefinition
13      }
14
15      /*=========================================================================*/
16      /*=================== INTERFACES AND EXTERNAL SYMBOLS =====================*/
17      /*=========================================================================*/
18
19      external Code;
20
21
22      /*=========================================================================*/
23      /*============================== GRAMMAR ==================================*/
24      /*=========================================================================*/
25
26      /** ASTTCDefinition represents a Testcase using UML Objectdiagrams and Java
27          @attribute name         Name of this Testcase
28          @attribute tCUnitTests  List of Testcase-units of this Testcase
29      */
30      / TCDefinition astextends /mc.umlp.common._ast.UMLPNode =
31          "testcase" Name
32          "{" tCUnitTests:TCUnitTest* "}";
33
34
35      /** ASTTCUnitTest represents a Unit-Test of a Testcase
36          @attribute name                 Name of this Unit-Test
37          @attribute setupReferenceName   Reference to the initial setup of this
38                                          Unit-Test (e.g. an Objectdiagram)
39          @attribute code                 Optional execution part of this Unit-Test
40          @attribute eventsReferenceName Optional reference to the execution and
41                                          interaction part of this Unit-Test
42                                          (e.g. a sequencediagram)
43          @attribute resultReferenceName Reference to the expected result of this
44                                          Unit-Test (e.g. an Objectdiagram)
45      */
46      / TCUnitTest astextends /mc.umlp.common._ast.UMLPNode =
47          Name "{"
48            ("setup" setupReferenceName:QualifiedName ";")?
49            (("{")=> Code | "events" eventsReferenceName:QualifiedName ";")
50            ("result" resultReferenceName:QualifiedName ";")?
51          "}";
52
53  }
```

Quellcode C.9: MontiCore-Grammatik für die Testspezifikationssprache

C.9 OCL/P

```
1   package mc.umlp.ocl;                                                    MC
2
3   /**
4    * This grammar defines the Object Constraint Language (P-Profile).
5    *
6    * The OCL grammar uses types and literals that are inherited from super
7    * grammars.
8    *
9    * @author Claas Pinkernell
10   * @version 1.5
11   */
12  grammar OCL extends mc.umlp.common.Common {
13
```

```
14      /*==========================================================================*/
15      /*============================== OPTIONS ===================================*/
16      /*==========================================================================*/
17
18      options {
19          compilationunit OCLFile
20
21          /* This is used to embed OCL in Statecharts. OCL expressions are
22             surrounded by brackets. Since an expression can be a single token,
23             this conflicts with the lookahead of the parser. So we have to
24             define the set of following tokens manually.
25             Note that this has to be extended for further embedments.
26          */
27          follow OCLEmbedment "]";
28      }
29
30      /*==========================================================================*/
31      /*============================ PRODUCTIONS  ================================*/
32      /*==========================================================================*/
33
34      /*======================== HEADER DEFINITIONS ==============================*/
35
36      /** ASTOCLEmbedment is used for embedment of OCL in other grammars.
37          OCLEmbedment includes invariants, operation constraints and abstract
38          expressions.
39          @attribute OCLDefinition Constraint or abstract expression
40      */
41      / OCLEmbedment astextends /mc.umlp.common._ast.UMLPNode =
42          OCLDefinition;
43
44      /** ASTOCLDefinition subsumes all OCL types that are used as embedded OCL.
45          An OCLDefinition is a Constraint (Invariant or Method- or
46          ConstructorSpec) or an abstract expression (oclexpression).
47      */
48      interface OCLDefinition;
49
50      /** ASTOCLFile represents a file that contains an OCL-Constraint.
51          @attribute Stereotype     Optional Stereotype
52          @attribute name           Name of OCLFile (for MCCompilationUnit)
53          @attribute OCLConstraints List of OCL-constraints
54      */
55      // Note: Prefix should be only "ocl" (-> Context Condition). On the
56      //       other side this should not prevent using "ocl" as identifiers.
57      //       Therefore this is not defined as terminal symbol.
58      / OCLFile astextends /mc.umlp.common._ast.UMLPNode =
59          Stereotype? prefix:Name Name "{"
60            OCLConstraints:OCLConstraint*
61          "}"
62          EOF;
63
64      /** ASTOCLConstraint defines operationconstraints or invariants with
65          stereotypes.
66          @attribute Stereotype       Optional Stereotype
67          @attribute OCLRawConstraint Interface (represents ASTOCLInvariant
68                                      or ASTOCLOperationConstraint)
69      */
70      / OCLConstraint astextends /mc.umlp.common._ast.UMLPNode
71      implements (Stereotype? "context")=>OCLDefinition =
72          Stereotype?
73          OCLRawConstraint;
74
75      /** ASTOCLRawConstraint subsumes invariants and method-/constructor-
76          specifications.
77      */
78      interface OCLRawConstraint;
79
80      /** ASTOCLOperationConstraint represents the typical method or constructor
81          specifications with pre- and post-conditions.
```

```
82          @attribute Stereotype            Optional stereotype
83          @attribute OCLOperationSignature Operation signature
84          @attribute OCLLetDeclaration     Optional let-declarations
85          @attribute OCLPreStatement       Optional pre-statements
86          @attribute OCLPostStatement      Optional post-statements
87      */
88      / OCLOperationConstraint astextends /mc.umlp.common._ast.UMLPNode
89      implements ("context" Stereotype? OCLOperationSignature)=>OCLRawConstraint =
90          "context"
91          Stereotype?
92          OCLOperationSignature
93          OCLLetDeclaration?
94          OCLPreStatement?
95          OCLPostStatement?;
96
97      /** ASTOCLOperationSignature subsumes method and constructor signatures.
98      */
99      interface OCLOperationSignature;
100
101     /** ASTOCLInvariant represents the typical invariant definitions of OCL.
102         @attribute OCLClassContext Optional context definitions of the invariant
103         @attribute name            Name of the invariant
104         @attribute OCLParameters   Optional parameters of the invariant
105         @attribute statements      List of boolean expressions
106     */
107     / OCLInvariant astextends /mc.umlp.common._ast.UMLPNode
108     implements OCLRawConstraint =
109         OCLClassContext?
110         "inv" Name?
111         OCLParameters? ":"
112         (statements:OCLExpression ";")+;
113
114     /** ASTOCLClassContext represents the OCL-Context-Clause of an invariant.
115         @attribute contextType        Type of context (context or import)
116         @attribute contextDefinitions List of context definitions.
117     */
118     / OCLClassContext astextends /mc.umlp.common._ast.UMLPNode =
119         (context:["context"] | Import:["import"])
120         contextDefinitions:OCLContextDefinition
121         "," contextDefinitions:OCLContextDefinition)*;
122
123     /** ASTOCLContextDefinition represents the definition of a context-variable.
124         @attribute className Class of the context variable (only class names
125                              allowed here! No generics, primitives or Set,
126                              List, Collection)
127         @attribute name      Optional name of context variable
128     */
129     / OCLContextDefinition astextends /mc.umlp.common._ast.UMLPNode =
130         className:QualifiedName Name?;
131
132     /** ASTOCLPreStatement represents a list of pre-statements subsumed to a
133         pre-condition.
134         @attribute name       Optional name of pre-condition
135         @attribute statements List of boolean expressions
136     */
137     / OCLPreStatement astextends /mc.umlp.common._ast.UMLPNode =
138         "pre" Name? ":"
139         (statements:OCLExpression ";")+;
140
141     /** ASTOCLPostStatement represents a list of post-statements subsumed to a
142         post-condition.
143         @attribute name       Optional name of post-conditions
144         @attribute statements List of boolean expressions
145     */
146     / OCLPostStatement astextends /mc.umlp.common._ast.UMLPNode =
147         "post" Name? ":"
148         (statements:OCLExpression ";")+;
149
```

```
150     /** ASTOCLMethodSignature defines the signature of a Java 5 method.
151         @attribute TypeParameters  Optional type parameters of the given
152                                    method, e.g. sth. like <T>
153         @attribute ReturnType      Optional type that is returned by
154                                    given method.
155         @attribute className       Name of the method's class
156         @attribute Name            Name of given method
157         @attribute OCLParameters   Parameter declarations of given method
158         @attribute OCLThrowsClause Optional throwables
159     */
160     / OCLMethodSignature astextends /mc.umlp.common._ast.UMLPNode
161     implements OCLOperationSignature =
162         (options{greedy=true;}: TypeParameters)?
163         ((ReturnType (QualifiedName ".")? Name "("=> ReturnType |)
164         (className:QualifiedName ".")? Name
165         OCLParameters
166         OCLThrowsClause?;
167
168     /** ASTOCLConstructorSignature defines signature of a constructor.
169         @attribute ReferenceType   Type of the constructor
170         @attribute OCLParameters   Parameters of constructor
171         @attribute OCLThrowsClause Optional throwables
172     */
173     / OCLConstructorSignature astextends /mc.umlp.common._ast.UMLPNode
174     implements OCLOperationSignature =
175         "new" ReferenceType
176         OCLParameters
177         OCLThrowsClause?;
178
179     /** ASTOCLParameters defines a list of parameter declarations.
180         @attribute declarations Optional list of parameter declarations
181     */
182     / OCLParameters astextends /mc.umlp.common._ast.UMLPNode =
183         "(" (
184           params:OCLParameterDeclaration
185           "," params:OCLParameterDeclaration)*
186         )? ")";
187
188     /** ASTOCLParameterDeclaration defines declarations. It contains a type and
189         an identifier.
190         @attribute Type Type of declaration
191         @attribute name Name of variable
192     */
193     / OCLParameterDeclaration astextends /mc.umlp.common._ast.UMLPNode =
194         Type Name;
195
196     /** ASTOCLThrowsClause defines throwables for a method or constructor
197         signature.
198         @attribute throwables List of throwables
199     */
200     / OCLThrowsClause astextends /mc.umlp.common._ast.UMLPNode =
201         "throws" throwables:ComplexType ("," throwables:ComplexType)*;
202
203     /*============================ COMPREHENSION ==============================*/
204     /*================== (also see OCLComprehensionPrimary) ==================*/
205
206     /** ASTOCLComprehensionExpr defines comprehension kinds.
207         e.g.: - {x in y | x > 0} (OCLComprehensionVarDeclaratorStyle)
208               - {x * x | x in y} (OCLComprehensionExpressionStyle)
209               - {1..3, x..10, y} (OCLComprehensionEnumerationStyle)
210     */
211     interface OCLComprehensionExpr;
212
213     /** ASTOCLComprehensionVarDeclaratorStyle defines a comprehension with given
214         characteristic.
215         @attribute generator          A collection declaration (e.g. "int x in
216                                       y" or "Auction a" as shortform of "Auction
217                                       a in Auction.allInstances")
```

```
218         @attribute comprehensionItems Characterization of comprehension as a list
219                                       of comprehension-items
220     */
221     / OCLComprehensionVarDeclaratorStyle astextends /mc.umlp.common._ast.UMLPNode
222     implements (OCLCollectionVarDeclaration "|")=>OCLComprehensionExpr =
223         generator:OCLCollectionVarDeclaration "|"
224         comprehensionItems:OCLComprehensionItem
225         "," comprehensionItems:OCLComprehensionItem)*;
226
227     /** ASTOCLComprehensionExpressionStyle defines a comprehension with given
228         characteristic.
229         @attribute expression         An expression (e.g. "x*x")
230         @attribute comprehensionItems Characterization of comprehension as a
231                                       list of comprehension-items. This can
232                                       be generators, vardefinitions or filters.
233         Note that we assume at least one generator (e.g. x in Y) in here.
234     */
235     / OCLComprehensionExpressionStyle astextends /mc.umlp.common._ast.UMLPNode
236     implements (OCLNonGreedyEquivalentExpr "|")=>OCLComprehensionExpr =
237         //Note: we are parsing a non greedy expression in here, so we exclude
238         //single logical or.
239         expression:OCLNonGreedyEquivalentExpr "|"
240         comprehensionItems:OCLComprehensionItem
241         "," comprehensionItems:OCLComprehensionItem)*;
242
243     /** ASTOCLComprehensionEnumerationStyle is used for an enumeration of
244         comprehension elements. Note that collection items are optional.
245         @attribute collectionItems Enumerated elements as a comma separated
246                                    list (e.g.: "1..3, x, y..z")
247     */
248     / OCLComprehensionEnumerationStyle astextends /mc.umlp.common._ast.UMLPNode
249     implements OCLComprehensionExpr =
250         (
251           collectionItems:OCLCollectionItem
252           "," collectionItems:OCLCollectionItem)*
253         )?;
254
255     /** ASTOCLComprehensionItem defines items of a comprehension.
256     */
257     interface OCLComprehensionItem;
258
259     /** ASTOCLCollectionItem is a single comprehension item or a valuation,
260         e.g. {1..5}.
261         @attribute expression StartValue as an expression (can also be a
262                               primary or literal or constant)
263         @attribute extension  Optional extension to valuation
264     */
265     / OCLCollectionItem astextends /mc.umlp.common._ast.UMLPNode =
266         //Note: we are parsing a non greedy expression in here, so we exclude
267         //single logical or.
268         expression:OCLNonGreedyEquivalentExpr
269         (".." extension:OCLNonGreedyEquivalentExpr)?;
270
271
272     /* ============================== DECLARATORS ============================*/
273
274     /** ASTOCLDeclaration defines declarations of variables and methods, that
275         are used in let-constructs.
276     */
277     interface OCLDeclaration;
278
279     /** ASTOCLVariableDeclaration defines a local variable in a let declaration
280         or a comprehension.
281         @attribute Type          Optional type
282         @attribute name          Name of the variable
283         @attribute OCLExpression Definition as an expression
284                                  (right hand of the assignment)
285     */
```

```
286     / OCLVariableDeclaration astextends /mc.umlp.common._ast.UMLPNode
287     implements (Type? Name "=")=>OCLComprehensionItem,
288                (Type? Name "=")=>OCLDeclaration =
289         Type? Name "=" OCLExpression;
290
291     /** ASTOCLMethodDeclaration defines internal helping methods in let
292         constructs, e.g. "max(int a, int b) = if (a>b) then ...".
293         @attribute ReturnType    Optional returntype of method
294         @attribute name          Name of method
295         @attribute OCLParameters List of parameters (can also be emtpy)
296         @attribute OCLExpression Method definition as an expression
297                                  (right hand of the assignment)
298     */
299     / OCLMethodDeclaration astextends /mc.umlp.common._ast.UMLPNode
300     implements OCLDeclaration =
301         ReturnType?
302         Name
303         OCLParameters
304         "=" OCLExpression;
305
306     /** ASTOCLCollectionVarDeclaration defines expressions like
307         "int x in {1, 3, 5}".
308         @attribute Type       Optional type of declaration
309         @attribute name       Name of the variable
310         @attribute expression Optional right hand of the in-expression,
311                               e.g. a comprehension or a type
312     */
313     / OCLCollectionVarDeclaration astextends /mc.umlp.common._ast.UMLPNode
314     implements ((Name "in")|(Type Name))=>OCLComprehensionItem =
315         (
316           Type
317           Name
318           ("in" expression:OCLNonGreedyEquivalentExpr)?
319         )
320         |
321         (
322           Name
323           "in" expression:OCLNonGreedyEquivalentExpr
324         );
325
326     /*============================== EXPRESSIONS ==============================*/
327
328     /** ASTOCLExpression produces expressions that are used in OCL.
329         This rule is used for parsing only.
330     */
331     interface OCLExpression extends OCLDefinition, OCLComprehensionItem;
332
333     /** ASTOCLIfThenElseExpr defines an if-clause in ocl style. It differs from
334         Java 5: if ... then ... else .... /!\ else is NOT optional.
335         @attribute condition      If-expression (as a condition with boolean type)
336         @attribute thenExpression Then-expression (executed if condition is true)
337         @attribute elseExpression Else-expression (executed if condition is false)
338     */
339     / OCLIfThenElseExpr astextends /mc.umlp.common._ast.UMLPNode
340     implements OCLExpression =
341         "if" condition:OCLExpression
342         "then" thenExpression:OCLExpression
343         "else" elseExpression:OCLExpression;
344
345     /** The ASTOCLTypeIfExpr is an alternative to InstenceofExpression.
346         It combines instanceof and if-then-else.
347         @attribute unknownType    A variable as Name that will be checked
348         @attribute Type           Type that has to be matched
349         @attribute thenExpression If variable has given type then-expr
350                                   will be executed
351         @attribute elseExpression If variable has not given type
352                                   else-expr will be executed
353     */
```

```
354     / OCLTypeIfExpr astextends /mc.umlp.common._ast.UMLPNode
355     implements OCLExpression =
356         "typeif" unknownType:Name "instanceof" Type
357         "then" thenExpression:OCLExpression
358         "else" elseExpression:OCLExpression;
359
360     /** ASTOCLForAllExpr defines a quantified expression for collections,
361         e.g. "forall x in Y : ...".
362         @attribute declarations  List of collection variable declarations
363         @attribute OCLExpression The body of forall iteration as expression
364     */
365     / OCLForallExpr astextends /mc.umlp.common._ast.UMLPNode
366     implements OCLExpression =
367         "forall"
368         declarations:OCLCollectionVarDeclaration
369         "," declarations:OCLCollectionVarDeclaration)*
370         ":" OCLExpression;
371
372     /** ASTOCLExistsExpr defines a quantified expression for collections,
373         e.g. "exists x in Y : doSomething..."
374         @attribute declarations  List of collection variable declarations
375         @attribute OCLExpression The body of exists iteration as expression
376     */
377     / OCLExistsExpr astextends /mc.umlp.common._ast.UMLPNode
378     implements OCLExpression =
379         "exists"
380         declarations:OCLCollectionVarDeclaration
381         "," declarations:OCLCollectionVarDeclaration)*
382         ":" OCLExpression;
383
384     /** ASTOCLAnyExpr defines iterations with all objects of a collection,
385         e.g. "any x in set or any Auction".
386         @attribute OCLExpression A collection defined by an expression
387     */
388     / OCLAnyExpr astextends /mc.umlp.common._ast.UMLPNode
389     implements OCLExpression =
390         "any" OCLExpression;
391
392     /** ASTOCLLetinExpr are used to define local vars or methods. The defined
393         vars and methods are visible in the in-expression.
394         @attribute declarations  A list of variable or method declarations
395         @attribute OCLExpression Expression where previous declarations are used
396     */
397     / OCLLetinExpr astextends /mc.umlp.common._ast.UMLPNode
398     implements ("let" (OCLDeclaration ";")+ "in")=>OCLExpression =
399         "let" (declarations:OCLDeclaration ";")+
400         "in" OCLExpression;
401
402     /** ASTOCLLetDeclaration represents a list of let-declarations inside of a
403         let-statement. This contains variable- or method-declarations.
404         @attribute declarations List of variable- or method-declarations
405     */
406     / OCLLetDeclaration astextends /mc.umlp.common._ast.UMLPNode
407     implements OCLExpression =
408         "let" (declarations:OCLDeclaration ";")+;
409
410     /** ASTOCLIterateExpr is used to iterate collections. It differs from
411         Java 5 Iterator.
412         Example:
413             iterate{ elem in Auction; int acc=0 : acc = acc+elem.numberOfBids }.
414         @attribute iterationDeclarator The elements of a collection that will be
415                                        iterated as OCLCollectionVarDeclaration
416         @attribute initDeclarator      Definiton of a accumulation variable as
417                                        an OCLDeclaration.
418         @attribute accumulatorName     Name of the accumulation assignment variable
419         @attribute accumulatorValue    Right hand of the accumulation as an
420                                        expression
421     */
```

```
422     / OCLIterateExpr astextends /mc.umlp.common._ast.UMLPNode
423     implements OCLExpression =
424         "iterate" "{"
425         iterationDeclarator:OCLCollectionVarDeclaration ";"
426         initDeclarator:OCLDeclaration ":"
427         accumulatorName:Name "=" accumulatorValue:OCLExpression
428         "}";
429
430     /** ASTOCLConditionalExpr is equivalent to Java 5 ?-operator.
431         @attribute condition      Conditional expression
432         @attribute thenExpression Then expression
433         @attribute elseExpression Else expression
434     */
435     / OCLConditionalExpr astextends /mc.umlp.common._ast.UMLPNode
436     implements (("?" OCLConditionalExpr ":" OCLConditionalExpr)?)=>OCLExpression
437     returns OCLExpression =
438         ret=OCLEquivalentExpr
439         (
440           astscript{!(Condition=ret;);}
441           "?" thenExpression:OCLConditionalExpr
442           ":" elseExpression:OCLConditionalExpr
443         )?;
444
445     /** ASTOCLNonGreedyEquivalentExpr defines logical equivalence, e.g.
446         a && !a <=> true. This rule is used to distinguish between greedy and
447         non-greedy passes of expression hierarchy. (on non-greedy passes we do
448         not parse single logical or)
449         This rule is used for parsing only. It is not part of the AST.
450         @attribute leftHand  Left hand of equivalence
451         @attribute operator  The infix operator
452         @attribute rightHand Right hand of equivalence
453     */
454     OCLNonGreedyEquivalentExpr:OCLInfixExpression returns OCLExpression =
455         (
456           init {comprehensionStack.push(Greedy.OFF);}:
457           ret=OCLImpliesExpr
458           (
459             astscript{!(leftHand=ret;);}
460             operator:["<=>"] rightHand:OCLImpliesExpr
461           )*
462           (init {comprehensionStack.pop();}:)
463         );
464
465     /** ASTOCLEquivalentExpr defines logical equivalence, e.g. a && !a <=> true.
466         This rule is used to distinguish between greedy and non-greedy passes of
467         expression hierarchy. (on non-greedy passes we do not parse single
468         logical or)
469         This rule is used for parsing only. It is not part of the AST.
470         @attribute leftHand  Left hand of equivalence
471         @attribute operator  The infix operator
472         @attribute rightHand Right hand of equivalence
473     */
474     OCLEquivalentExpr:OCLInfixExpression returns OCLExpression =
475         (
476           //we assume greedy behaviour: parse single logical or
477           init {comprehensionStack.push(Greedy.ON);}:
478           ret=OCLImpliesExpr
479           (
480             astscript{!(leftHand=ret;);}
481             operator:["<=>"] rightHand:OCLImpliesExpr
482           )*
483           (init {comprehensionStack.pop();}:)
484         );
485
486     /** ASTOCLImpliesExpr defines logical implication, e.g. a implies c > 5.
487         This rule is used for parsing only. It is not part of the AST.
488         @attribute leftHand  Left hand of implication
489         @attribute operator  The infix operator
```

```
490         @attribute rightHand Right hand of implication
491     */
492     OCLImpliesExpr:OCLInfixExpression returns OCLExpression =
493         ret=OCLDoubleLogicalORExpr
494         (
495           astscript{!(leftHand=ret;);}
496           operator:["implies"] rightHand:OCLDoubleLogicalORExpr
497         )*;
498 
499     /** ASTOCLDoubleLogicalORExpr defines a logical OR-concatenation,
500         e.g. true || a.
501         This rule is used for parsing only. It is not part of the AST.
502         @attribute leftHand  Left hand of OR-concatenation
503         @attribute operator  The infix operator
504         @attribute rightHand Right hand of OR-concatenation
505     */
506     OCLDoubleLogicalORExpr:OCLInfixExpression returns OCLExpression =
507         ret=OCLDoubleLogicalANDExpr
508         (
509           astscript{!(leftHand=ret;);}
510           operator:["||"] rightHand:OCLDoubleLogicalANDExpr
511         )*;
512 
513     /** ASTOCLDoubleLogicalANDExpr defines a logical AND-concatenation,
514         e.g. true && a.
515         This rule is used for parsing only. It is not part of the AST.
516         @attribute leftHand  Left hand of AND-concatenation
517         @attribute operator  The infix operator
518         @attribute rightHand Right hand of AND-concatenation
519     */
520     OCLDoubleLogicalANDExpr:OCLInfixExpression returns OCLExpression =
521         ret=OCLSingleLogicalORExpr
522         (
523           astscript{!(leftHand=ret;);}
524           operator:["&&"] rightHand:OCLSingleLogicalORExpr
525         )*;
526 
527     /** ASTOCLSingleLogicalORExpr defines a binary OR-concatenation, e.g. 5 | a.
528         This rule is used for parsing only. It is not part of the AST.
529         @attribute leftHand  Left hand of OR-concatenation
530         @attribute operator  The infix operator
531         @attribute rightHand Right hand of OR-concatenation
532     */
533     OCLSingleLogicalORExpr:OCLInfixExpression returns OCLExpression =
534         //check if we assume non-greedy behaviour
535         {comprehensionStack.peek().equals(Greedy.OFF)}?
536         (
537           ret=OCLLogicalXORExpr
538           (
539             options{greedy=false;}: //be non-greedy
540             astscript{!(leftHand=ret;);}
541             operator:["|"] rightHand:OCLLogicalXORExpr
542           )*
543         )
544         |
545         (
546           ret=OCLLogicalXORExpr
547           (
548             options{greedy=true;}: //be greedy
549             astscript{!(leftHand=ret;);}
550             operator:["|"] rightHand:OCLLogicalXORExpr
551           )*
552         );
553 
554     /** ASTOCLLogicalXORExpr defines a binary XOR-concatenation, e.g. 5 ^ a.
555         This rule is used for parsing only. It is not part of the AST.
556         @attribute leftHand  Left hand of XOR-concatenation
557         @attribute operator  The infix operator
```

```
558         @attribute rightHand Right hand of XOR-concatenation
559     */
560     OCLLogicalXORExpr:OCLInfixExpression returns OCLExpression =
561         ret=OCLSingleLogicalANDExpr
562         (
563           astscript{!(leftHand=ret;);}
564           operator:["^"] rightHand:OCLSingleLogicalANDExpr
565         )*;
566
567     /** ASTOCLSingleLogicalANDExpr defines a binary AND-concatenation, e.g.
568         5 & a. This rule is used for parsing only. It is not part of the AST.
569         @attribute leftHand  Left hand of AND-concatenation
570         @attribute operator  The infix operator
571         @attribute rightHand Right hand of AND-concatenation
572     */
573     OCLSingleLogicalANDExpr:OCLInfixExpression returns OCLExpression =
574         ret=OCLRelationalExpr
575         (
576           astscript{!(leftHand=ret;);}
577           operator:["&"] rightHand:OCLRelationalExpr
578         )*;
579
580     /** ASTOCLRelationalExpr defines a relation between operands,
581         e.g. 5 == a or true != b.
582         This rule is used for parsing only. It is not part of the AST.
583         @attribute leftHand  Left hand of relation
584         @attribute operator  The infix operator
585         @attribute rightHand Right hand of relation
586     */
587     OCLRelationalExpr:OCLInfixExpression returns OCLExpression =
588         ret=OCLElemInExpr
589         (
590           astscript{!(leftHand=ret;);}
591           operator:["==" | "!="] rightHand:OCLElemInExpr
592         )?;
593
594     /** ASTOCLElemInExpr defines an element-in-collection-check,
595         e.g. 5 isin Set{4, 5}.
596         This rule is used for parsing only. It is not part of the AST.
597         @attribute leftHand  An element
598         @attribute operator  The infix operator
599         @attribute rightHand A collection
600     */
601     OCLElemInExpr:OCLInfixExpression returns OCLExpression =
602         ret=OCLCompareExpr
603         (
604           (
605             astscript{!(leftHand=ret;);}
606             operator:["isin"] rightHand:OCLCompareExpr
607           )*
608         |
609           //parameterization is necessary to leave infix expressions
610           ret=OCLInstanceofExpr->[ret]
611         );
612
613     /** ASTOCLInstanceofExpr defines an instance-of-type-check, e.g. an
614         instance of Integer.
615         This rule is used for parsing only. It is not part of the AST.
616         @attribute expression An object with unknown type
617         @attribute Type       A type
618     */
619     / OCLInstanceofExpr astextends /mc.umlp.common._ast.UMLPNode
620     implements OCLExpression [expression:OCLExpression] =
621         //note: we are implementing OCLExpression for right priority
622         //(the rule is passed through expression hierarchy)
623         //note: we are getting expression from OCLElemInExpr by parametrization
624         "instanceof" Type;
625
```

```
626     /** ASTOCLCompareExpr defines a comparison between operands,
627         e.g. 5 < a or 7 >= b.
628         This rule is used for parsing only. It is not part of the AST.
629         @attribute leftHand  Left hand of comparison
630         @attribute operator  The infix operator
631         @attribute rightHand Right hand of comparison
632     */
633     / OCLCompareExpr:OCLInfixExpression returns OCLExpression =
634         ret=OCLShiftExpr
635         (
636           astscript{!(leftHand=ret;);}
637           operator:["<" | "<=" | ">" | ">="] rightHand:OCLShiftExpr
638         )?;
639
640     /** ASTOCLShiftExpr defines a logical shift operation, e.g. a << 5.
641         This rule is used for parsing only. It is not part of the AST.
642         @attribute leftHand  Left hand of shift operation
643         @attribute operator  The infix operator
644         @attribute rightHand Right hand of shift operation
645     */
646     OCLShiftExpr:OCLInfixExpression returns OCLExpression =
647         ret=OCLBinaryPlusMinusExpr
648         (
649           astscript{!(leftHand=ret;);}
650           operator:["<<" | ">>" | ">>>"] rightHand:OCLBinaryPlusMinusExpr
651         )*;
652
653     /** ASTOCLBinaryPlusMinusExpr defines an addition or subtraction,
654         e.g. a + 5 or b - 3.
655         This rule is used for parsing only. It is not part of the AST.
656         @attribute leftHand  Left hand of addition or subtraction
657         @attribute operator  The infix operator
658         @attribute rightHand Right hand of addition or subtraction
659     */
660     OCLBinaryPlusMinusExpr:OCLInfixExpression returns OCLExpression =
661         ret=OCLBinaryMultDivModExpr
662         (
663           astscript{!(leftHand=ret;);}
664           operator:["+" | "-"] rightHand:OCLBinaryMultDivModExpr
665         )*;
666
667     /** ASTOCLBinaryMultDivModExpr defines a multiplication, division or modulo
668         operation, e.g. a * 5 or b / 3 or c % 2.
669         This rule is used for parsing only. It is not part of the AST.
670         @attribute leftHand  Left hand of the operation
671         @attribute operator  The infix operator
672         @attribute rightHand Right hand of the operation
673     */
674     OCLBinaryMultDivModExpr:OCLInfixExpression returns OCLExpression =
675         ret=OCLPrefixExpression
676         (
677           astscript{!(leftHand=ret;);}
678           operator:["*" | "/" | "%"] rightHand:OCLPrefixExpression
679         )*;
680
681     /** ASTOCLPrefixExpression defines prefix expressions, e.g. "!x" or "-a".
682         @attribute operator   The prefix operator
683         @attribute operator   The prefix operator
684         @attribute Expression The expression
685     */
686     //note: we are implementing OCLExpression for right priority
687     //(the rule is passed through expression hierarchy)
688     / OCLPrefixExpression astextends /mc.umlp.common._ast.UMLPNode
689     implements ("-" | "+" | "~" | "!"
690                   | ("(" Type ")" OCLPrefixExpression)=>
691                     "(" Type ")" OCLPrefixExpression | OCLPrimary)=>OCLExpression
692     returns OCLExpression =
693         (
```

```
694           (
695             astscript{!();}
696             operator:["-" | "+" | "~" | "!"] expression:OCLPrefixExpression
697           )
698           |
699           ("(" Type ")" OCLPrefixExpression)=>
700           ret=OCLTypeCastExpr
701           |
702           ret=OCLPrimary
703         );
704
705     /** ASTOCLTypeCastExpr defines typecast-operation, e.g. (String) s.
706         @attribute Type       Type of typecast
707         @attribute expression Argument of typecast
708     */
709     / OCLTypeCastExpr astextends /mc.umlp.common._ast.UMLPNode =
710         "(" Type ")" expression:OCLPrefixExpression;
711
712     /*============================== OCL PRIMARYS ==============================*/
713
714     /** ASTOCLPrimary defines primaries of OCL.
715     */
716     interface OCLPrimary;
717
718     /** ASTOCLParenthizedExpr defines expression within parenthesis, e.g. (4+5)*6.
719         @attribute OCLExpression Expression within parenthesis
720         @attribute qualification Optional qualification
721     */
722     / OCLParenthizedExpr astextends /mc.umlp.common._ast.UMLPNode
723     implements OCLPrimary =
724         "(" OCLExpression ")" ("." qualification:OCLPrimary)?;
725
726     /** ASTOCLComprehensionPrimary represents comprehensions.
727         @attribute Type          Optional type of comprehension,
728                                  e.g. Set, List, or Collection
729         @attribute expression    Expression of comprehension
730         @attribute qualification Optional qualification
731     */
732     / OCLComprehensionPrimary astextends /mc.umlp.common._ast.UMLPNode
733     implements (Name? ( "<" Type ">")? "{")=>OCLPrimary
734     {boolean validCollectionType = false;} =
735         //we do not use inherited Type here, because it has not the semantics
736         //of types here and we can not use keywords, so we have to use Name
737         //and a semantics predicate
738         (
739           type=Name
740           {
741             if (type != null) {
742               if (type.equals("Set")) {
743                 a.setSet(true);
744                 validCollectionType = true;
745               }
746               else if (type.equals("List")) {
747                 a.setList(true);
748                 validCollectionType = true;
749               }
750               else if (type.equals("Collection")) {
751                 a.setCollection(true);
752                 validCollectionType = true;
753               }
754             }
755           }
756         )?
757         {type == null || validCollectionType == true}?
758         ("<" referenceParameter:Type ">")?
759         "{" expression:OCLComprehensionExpr "}"
760         ("." qualification:OCLPrimary)?;
761
```

```
762     /** ASTOCLQualifiedPrimary represents qualified identifier.
763         @attribute prefixIdentifier     Value of first part of a qualified
764                                         identifier. It is null if we have
765                                         this, super or result as prefix.
766         @attribute this                 Is true if we have a this prefix
767         @attribute super                Is true if we have a super prefix
768         @attribute result               Is true if we have a result prefix
769         @attribute qualifications       List of qualifications of qualified
770                                         identifier
771         @attribute postfixQualification Optional argument-, array- or atpre,
772                                         **-qualification.
773         @attribute OCLQualifiedPrimary  Optional recursive linkage of
774                                         qualified primary.
775     */
776     / OCLQualifiedPrimary astextends /mc.umlp.common._ast.UMLPNode
777     implements OCLPrimary =
778         //prefix
779         (
780           this:["this"]
781           |
782           super:["super"]
783           |
784           res:["result"]
785           |
786           prefixIdentifier:Name
787         )
788         //qualifications
789         ("." qualifications:Name)*
790         //post qualification and recursive linkage
791         (
792           postfixQualification:OCLQualification
793           ("." OCLQualifiedPrimary)?
794         )?;
795
796     /** ASTOCLQualification defines qualification postfixes of a qualified
797         primary. A qualification is one of the three kinds:
798             - x[0] (ArrayQualification)
799             - x(y) (ArgumentQualification)
800             - x** or xAtpre (PostfixQualification)
801     */
802     interface OCLQualification;
803
804     /** ASTOCLArrayQualification represents an array qualification.
805         @attribute arguments The array arguments (indices)
806     */
807     / OCLArrayQualification astextends /mc.umlp.common._ast.UMLPNode
808     implements OCLQualification =
809         ("[" arguments:OCLExpression "]")+;
810
811     /** ASTOCLArgumentQualification represents the arguments of a method call.
812         @attribute arguments Arguments of method call
813     */
814     / OCLArgumentQualification astextends /mc.umlp.common._ast.UMLPNode
815     implements OCLQualification =
816         "(" (arguments:OCLExpression ("," arguments:OCLExpression)*)? ")";
817
818     /** ASTOCLPostfixQualification represents a ATpre or ** qualification.
819         @attribute postfix          Kind of postfix: atpre or **
820         @attribute OCLQualification Optional qualification
821     */
822     / OCLPostfixQualification astextends /mc.umlp.common._ast.UMLPNode
823     implements OCLQualification =
824         (atpre:["@pre"] | transitive:["**"])
825         OCLQualification?;
826
827     /** ASTOCLIsnewPrimary represents isnew-function.
828         @attribute OCLExpression Argument of isnew-operation
829     */
```

```
830     / OCLIsnewPrimary astextends /mc.umlp.common._ast.UMLPNode
831     implements OCLPrimary =
832         "isnew" "(" OCLExpression ")";
833
834     /** ASTOCLDefinedPrimary represents defined-function.
835         @attribute OCLExpression Argument of defined-operation
836     */
837     / OCLDefinedPrimary astextends /mc.umlp.common._ast.UMLPNode
838     implements OCLPrimary =
839         "defined" "(" OCLExpression ")";
840
841     /** ASTOCLLiteral capsulates inherited literals to implement ocl primary
842         interface.
843     */
844     / OCLLiteral astextends /mc.umlp.common._ast.UMLPNode implements OCLPrimary =
845         value:Literal;
846
847     /*=============================================================================*/
848     /*============================== ASTRULES =====================================*/
849     /*=============================================================================*/
850
851     /** ASTOCLInfixExpressions are expressions with a left hand expression, an
852         operator and a right hand expression.
853     */
854     ast OCLInfixExpression astextends /mc.umlp.common._ast.UMLPNode =
855         leftHand:OCLExpression;  //add additional field left hand
856
857     /** ASTOCLConditionalExpressions are if-then-else clauses.
858     */
859     ast OCLConditionalExpr =
860         condition:OCLExpression; //add additional field condition
861
862     /** ASTOCLPrimary represents a basic interface for all primaries and extends
863         ASTOCLExpression.
864     */
865     ast OCLPrimary astextends OCLExpression;
866
867     ast OCLComprehensionPrimary =
868         set:/boolean
869         list:/boolean
870         collection:/boolean;
871
872     ast OCLTypeCastExpr astimplements OCLExpression;
873
874     /*=============================================================================*/
875     /*============================== CONCEPTS =====================================*/
876     /*=============================================================================*/
877
878     concept antlr {
879         parser java {
880           /* We do not want to parse expressions like "a | b" greedy, while we
881              are parsing a comprehension. So we have to distinguish between
882              greedy and non greedy passes of expression hierarchy.
883              So we are managing a stack that handles our distinction.
884           */
885           public enum Greedy { ON, OFF };
886           public java.util.Stack<Greedy> comprehensionStack
887               = new java.util.Stack<Greedy>();
888         }
889     }
890 }
```

Quellcode C.10: MontiCore-Grammatik für die OCL/P (in Zusammenarbeit mit [PS07, Pin07])

C.10 Java

MC

```
package mc.javadsl;

grammar JavaDSL extends mc.types.Types {

    options {
        lexer lookahead=4
        keywords {"."}
    }

    /** ASTCompilationUnit represents a whole JAVA file
        @attribute PackageDeclaration Represents the package, the file belongs to
        @attribute ImportDeclarations A list of the ImportDeclaration
        @attribute TypeDeclarations A list of declared types in the file
    */
    CompilationUnit =
        (((Annotation)* "package")=> PackageDeclaration:PackageDeclaration)?
        (ImportDeclarations:ImportDeclaration)*
        (TypeDeclarations:TypeDeclaration | ";" )*;

    /** ASTPackageDeclaration represents a package declaration.
        @attribute Modifiers Annotations in front of the declaration
        @attribute Name Qualified package name
    */
    PackageDeclaration =
       (Modifiers:Annotation)*
       "package" Name ( "." Name )* ";";

    /** ASTImportDeclaration represents an import declaration
        @attribute Name Qualified name of the class or the package, which is
                        imported
        @attribute IsStatic Is true if keyword static is found in the declaration
        @attribute IsOnDemand Is true if declaration ends with a '*', so all
                              classes of a package are imported
    */
    ImportDeclaration =
        "import" (IsStatic:["static"])?
        Name (options{greedy=true;}: "." Name)*
        ("." IsOnDemand:["*"])? ";";

    /** ASTTypeDeclaration represents the possible top level types in JAVA, which
        can be a class declaration, enum declaration, interface declaration,
        or annotation type declaration. The rule redirects "only" to the passing
        rules.
    */
    interface TypeDeclaration extends
        (TypeDeclaration)=>AnnotationMemberDeclaration,
        (TypeDeclaration)=>MemberDeclaration;

    ast TypeDeclaration =
        Name
        Modifiers:Modifier*;

    /** ASTVariableDeclaration represents a whole variable declaration. It consist
        of modifiers, a type, and one or more VariableDeclarators separated by ';'
        @attribute Modifiers A list of Modifiers
        @attribute Type The variable type
        @attribute Declarators List of VariableDeclarators
    */
    VariableDeclaration =
        (Modifiers:Modifier)*
        Type
        Declarators:VariableDeclarator
        (options{warnWhenFollowAmbig=false;}: "," Declarators:VariableDeclarator)*;
```

```
64      /** ASTVariableDeclarationStatement represents just an VariableDeclaration
65          followed by a ";". Only a redirect to VariableDeclaration, but needed
66          to make a Statement of it
67      */
68      VariableDeclarationStatement implements (VariableDeclaration ";")=>Statement =
69          Declaration:VariableDeclaration ";";
70
71      /** ASTVariableDeclarationExpression represents just a
72          VariableDeclarationExpression. Only a redirect to VariableDeclaration,
73          but needed to make an Expression of it
74      */
75      VariableDeclarationExpression implements (VariableDeclaration)=>Expression =
76          Declaration:VariableDeclaration;
77
78      /** The interface ASTModifier redirect to an Annotation or a PrimitveModifier
79      */
80      interface Modifier;
81
82      /** ASTPrimitiveModifier represents the modifier keywords.
83          @attribute Modifier Modifier represents one of the Modifier Keywords:
84                              "private", "public", "protected", "static",
85                              "transient", "final", "abstract", "native",
86                              "threadsafe", "synchronized", "const",
87                              "volatile", "strictfp"
88      */
89      PrimitiveModifier implements Modifier =
90          Modifier:["private" | "public" | "protected" | "static" | "transient"
91          | "final" | "abstract" | "native" | "threadsafe" | "synchronized"
92          | "const" | "volatile" | "strictfp"];
93
94      /** ASTEnumDeclaration is the declaration for an enum type.
95          @attribute Modifiers A list of Modifiers
96          @attribute ImplementedTypes A list of implemented interfaces
97          @attribute EnumConstantDeclarations A list of EnumConstandDeclarations
98          @attribute MemberDeclarations A list of MemberDclarations (a class body)
99      */
100     EnumDeclaration implements (Modifier* "enum")=>TypeDeclaration =
101         (Modifiers:Modifier)* "enum" Name
102         ("implements" ImplementedTypes:Type ("," ImplementedTypes:Type)* )?
103         "{"
104           (EnumConstantDeclarations:EnumConstantDeclaration
105           (options{greedy=true;}:
106             "," EnumConstantDeclarations:EnumConstantDeclaration)*)?
107           (",")? // Note: This comma will not be stored in AST,
108                  //       so the Pretty Printer can't print it
109           (";" (MemberDeclarations:MemberDeclaration | ";")* )?
110         "}";
111
112     /** ASTEnumConstantDeclaration is in fact a class instance and can be followed
113         by the usual class declarations
114         @attribute Annotations A list of Annotations
115         @attribute Arguments A list of Expressions
116         @attribute MemberDeclarations A list of MemberDeclaration
117     */
118     EnumConstantDeclaration =
119         (Annotations:Annotation)* Name
120         ( "(" (Arguments: Expression ("," Arguments: Expression)* )? ")" )?
121         (
122           "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}"
123           {a.getMemberDeclarations().set_Existent(true);}
124         )?;
125
126     /** ASTAnnotationTypeDeclaration is a special kind of interface declaration for
127         annotations marked with an "@"
128         @attribute Modifiers A list of (Interface-) Modifiers
129         @attribute Name The name of the annotation
130         @attribute MemberDeclarations A list of MemberDeclaration, similar to
131                                       class body
```

```
132     */
133     AnnotationTypeDeclaration implements (Modifier* "@")=>TypeDeclaration =
134         //semantic: this is an interface, not all Modifiers allowed
135         (Modifiers:Modifier)* "@" "interface" Name
136         "{"
137           (MemberDeclarations:AnnotationMemberDeclaration | ";")*
138         "}";
139
140     /** ASTAnnotationMemberDeclaration represents the possible member declarations
141         which can be used in an annotation type declaration. This rule only
142         redirects to the passing rules.
143     */
144     interface AnnotationMemberDeclaration;
145
146     ast AnnotationMemberDeclaration astextends MemberDeclaration;
147
148     /** ASTAnnotationMethodDeclaration is a special kind of method declaration
149         which is only used in an annotation member declaration (and so in an
150         annotation type declaration)
151         @attribute Modifiers A list of Modifiers
152         @attribute Type The type of the method
153         @attribute Name The name of the declared method
154         @attribute DefaultValue An AnnotationMemberValue which is an optional
155                                 default value in the method declaration
156     */
157     AnnotationMethodDeclaration
158     implements ((Modifier)* ReturnType Name "(" ")")=>AnnotationMemberDeclaration =
159         //semantic check: modifiers are only allowed to be:
160         //                public, abstract, annotation
161         (Modifiers:Modifier)* ReturnType Name "(" ")"
162         ("default" DefaultValue:AnnotationMemberValue )? ";";
163
164     /** There are three kinds of Annotation-types: The MappedMemberAnnotation,
165         SingleMemberAnnotation, and the MarkerAnnotation. The Annotation rule
166         only redirects to the under rules.
167     */
168     interface Annotation extends Modifier;
169
170     ast Annotation astextends Expression, Modifier;
171
172     /** ASTMappedMemberAnnotation is the normal way to construct an annotation. It
173         exists of a TypeName, a StringList to construct qualified names, followed
174         by a list of MemberValuePairs
175         @attribute Name Name is the TypeName of an annotation
176         @attribute MemberValue MemberValue is an AnnotationMemberValuePair
177     */
178     MappedMemberAnnotation
179     implements ("@" Name ("." Name)* "(" AnnotationMemberValuePair)=>Annotation =
180         "@" Name (options{greedy=true;}:  "." Name)*
181         "("
182           MemberValues:AnnotationMemberValuePair
183           "," MemberValues:AnnotationMemberValuePair)*
184         ")";
185
186     /** ASTSingleMemberAnnotation is a short hand design of the normal
187         MappedMemberAnnotation for use with single element annotation. It exists
188         of a TypeName, a StringList to construct qualified names, followed by a
189         single AnnotationMemberValue.
190         @attribute Name Name is the TypeName of an annotation
191         @attribute MemberValue MemberValue an AnnotationMemberValue
192     */
193     SingleMemberAnnotation
194     implements ("@" Name ("." Name)* "(" AnnotationMemberValue)=>Annotation =
195         "@" Name (options{greedy=true;}: "." Name)*
196         "(" MemberValue:AnnotationMemberValue ")";
197
198     /** ASTMarkerAnnotation is a very short hand design of the normal
199         MappedMemberAnnotation for use with only the Name. It only
```

```
200         exists of a TypeName followed by an optional "()".
201         @attribute Name Name is the TypeName of an annotation
202     */
203     MarkerAnnotation implements Annotation =
204         "@" Name (options{greedy=true;}:  "." Name)*
205         ("(" ")" )?;
206
207     /** ASTAnnotationMemberValuePair is a single identifier followed by an
208         AnnotationMemberValue.
209         @attribute MemberName The member name is an identifier
210         @attribute Value The value is the type of an AnnotationMemberValue
211     */
212     AnnotationMemberValuePair =
213         MemberName:Name "=" Value:AnnotationMemberValue;
214
215     /** ASTAnnotationMemberValue only redirect to the rules: annotation,
216         conditional expression, or annotation member array initializer.
217         There never will be instances of this ASTNode.
218     */
219     AnnotationMemberValue returns Expression =
220         ret = Annotation
221         | ret = ConditionalExpression
222         | ret = AnnotationMemberArrayInitializer;
223
224     /** ASTAnnotationMemberArrayInitializer is an array which only contains
225         AnnotationMemberValues
226         @attribute Initializers List of AnnotationMemberValues
227     */
228     //Need to be an Expression because of the rule AnnotationMemberValue
229     AnnotationMemberArrayInitializer
230     implements ("{" (AnnotationMemberValue ("," AnnotationMemberValue)*)?
231                 (",")? "}")=>Expression =
232         "{"
233           (
234             Initializers:AnnotationMemberValue
235             (
236               options{warnWhenFollowAmbig=false;}:
237               "," Initializers:AnnotationMemberValue
238             )*
239           )?
240           (",")? // Note: This comma will not be stored in AST,
241                  //       so the Pretty Printer can't print it
242         "}";
243
244     /** ASTClassDeclaration represents a whole class with its block identified by
245         the keyword "class".
246         @attribute Modifiers A Modifierlist with all possible Modifiers
247         @attribute Name The class name
248         @attribute TypeParameters A list of type parameters
249         @attribute ExtendedType A Type which is extended by the class.
250                                 Only one allowed!
251         @attribute ImplementedInterfaces A list of implemented interfaces.
252         @attribute MemberDeclarations The class body is represented by the
253                                       "{ classBody }", which exists of multiple
254                                       MemberDeclaration or a ";"
255     */
256     ClassDeclaration implements ((Modifiers:Modifier)* "class")=>TypeDeclaration =
257         //semantic check: not all modifiers allowed
258         (Modifiers:Modifier)* "class" Name
259         TypeParameters
260         ("extends" ExtendedClass:Type)?
261         ("implements" ImplementedInterfaces:Type ("," ImplementedInterfaces:Type)*)?
262         "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}";
263
264     /** ASTInterfaceDeclaration represents a whole interface, which is very
265         similar to a class.
266         Identified by the keyword interface
267         @attribute Modifiers A Modifierlist with all possible Modifiers
```

```
268         @attribute Name The interface name
269         @attribute TypeParameters A list of type parameters
270         @attribute ExtendedTypes A list of extended types. Multiple allowed
271         @attribute MemberDeclarations The class body is represented by the
272                                       "{ classBody }", which exists of multiple
273                                       MemberDeclaration or a ";". In comparing
274                                       to the class declaration, here not all
275                                       possible member declarations are allowed.
276     */
277     InterfaceDeclaration implements (Modifier* "interface")=>TypeDeclaration =
278         //semantic check: not all modifiers allowed
279         (Modifiers:Modifier)* "interface" Name
280         TypeParameters
281         ("extends" ExtendedInterfaces:Type ("," ExtendedInterfaces:Type)* )?
282         "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}";
283
284     /** ASTMemberDeclaration is an interface for ASTTypeDeclaration,
285         ASTMethodDeclaration, ASTConstructorDeclaration, and ASTFieldDeclaration.
286     */
287     interface MemberDeclaration;
288
289     /** ASTTypeInitializer represents a type initializer, which is just a
290         ASTBlockStatement headed by an optional 'static' modifier
291         @attribute IsStatic Is true if 'static' keyword is present
292     */
293     TypeInitializer implements (("static")? BlockStatement)=>MemberDeclaration =
294         (IsStatic:["static"])?
295         Body:BlockStatement;
296
297     /** ASTMethodOrConstructorDeclaration is a shared interface of the two classes
298     */
299     interface MethodOrConstructorDeclaration; //This rule is never invocated
300
301     ast MethodOrConstructorDeclaration astextends MemberDeclaration =
302         Name
303         Throwables:QualifiedName*
304         Modifiers:Modifier*
305         TypeParameters
306         Parameters:ParameterDeclaration*
307         Block:BlockStatement;
308
309     /** ASTConstructorDeclaration represents the declaration of a constructor
310         @attribute Modifiers List of Modifiers or Annotations
311         @attribute TypeParameter List of (optional) generic TypeParameters
312         @attribute Name Single name of the constructor
313         @attribute Parameters List of parameters
314         @attribute Throwables List of the qualified names of exceptions,
315                                 which could be thrown by the constructor
316         @attribute Block The actual block statement
317     */
318     ConstructorDeclaration
319     implements ((Modifier)* TypeParameters Name "("=>MemberDeclaration =
320         //Modifiers and TypeParameters are moved from rule MemberDeclaration
321         (Modifiers:Modifier)*
322         TypeParameters
323         Name
324         "("
325           (Parameters:ParameterDeclaration ("," Parameters:ParameterDeclaration)*)?
326         ")"
327         ( //this is the inlined rule thowsClause
328           "throws" Throwables:QualifiedName ("," Throwables:QualifiedName)*
329         )?
330         Block:BlockStatement;
331
332     ast ConstructorDeclaration astimplements MethodOrConstructorDeclaration;
333
334     /** ASTMethodDeclaration represents the declaration of a method
335         @attribute Modifiers List of Modifiers or Annotations
```

```
336         @attribute TypeParameter List of (optional) generic TypeParameters
337         @attribute ReturnType The type of the return value of the method
338         @attribute Name Single name of the constructor
339         @attribute Parameters List of parameters
340         @attribute Throwables List of the qualified names of exceptions,
341                                which could be thrown by the constructor
342         @attribute Block The actual block statement
343     */
344     MethodDeclaration
345     implements ((Modifier)* TypeParameters ReturnType Name "(")=>MemberDeclaration =
346         //Modifiers and TypeParameters are moved from rule MemberDeclaration
347         (Modifiers:Modifier)*
348         TypeParameters
349         ReturnType Name
350         "("
351           (Parameters:ParameterDeclaration ("," Parameters:ParameterDeclaration)*)?
352         ")"
353         //this is the inlined rule optionalDims
354         (
355           options{greedy=true;}: "[" "]"
356           {a.setAdditionReturnTypeDimensions(
357               a.getAdditionReturnTypeDimensions()+1);}
358         )*
359         ( //this is the inlined rule thowsClause with the inlined rule greedyName
360           "throws" Throwables:QualifiedName ("," Throwables:QualifiedName)*
361         )?
362         ( ";" | Block:BlockStatement );
363
364     Methods =
365         MethodDeclaration+;
366
367     ast MethodDeclaration astimplements MethodOrConstructorDeclaration =
368         AdditionReturnTypeDimensions : /int
369     ;
370
371     /** ASTFieldDeclaration represents the declaration of a field. This is just
372         a VariableDeclaration followed by a ';'.
373         @attribute Declaration One VariableDeclaration
374     */
375     FieldDeclaration
376     implements ((Modifier)* Type VariableDeclarator)=>MemberDeclaration,
377                ((Modifier)* Type VariableDeclarator)=>AnnotationMemberDeclaration =
378         //Modifiers are moved from rule MemberDeclaration to VariableDeclaration
379         Declaration:VariableDeclaration ";";
380
381     /** ASTVariableDeclarator represents the part of a variable declaration
382         beginning at the name, e.g. 'varName = "someInitializer" '
383         It is also part of a field declaration.
384         @attribute Name Variable name
385         @attribute AdditionalArrayDimensions Counts the dimension if variable is
386                                              an array
387         @attribute Represents the initial assignment
388     */
389     VariableDeclarator =
390         Name
391         (
392           options{greedy=true;}: "[" "]"
393           {a.setAdditionalArrayDimensions(a.getAdditionalArrayDimensions()+1);}
394         )*
395         ("=" Initializer:Initializer)?;
396
397     ast VariableDeclarator =
398         AdditionalArrayDimensions:/int;
399
400     /** ASTArrayInitializer represents the statement in curly braces
401         (incl. these braces!), which is used to initialize an array.
402         @attribute Initializers A list of expressions or other array-initializers
403     */
```

```
404     ArrayInitializer implements Expression =
405         "{"
406         (
407           Initializers:Initializer
408           (options{warnWhenFollowAmbig=false;}: "," Initializers:Initializer)*
409         )?
410         (",")?  // Note: This comma will not be stored in AST,
411                 //       so the Pretty Printer can't print it
412         "}";
413
414     /** ASTInitializer is necessary to combine Expressions of an
415         ArrayInitializer. It represents the X from this example:
416         int[] a = { X , X }
417         Never returns an ASTInitializer object
418     */
419     Initializer returns Expression =
420         ret=Expression;
421
422     /** ASTParameterDeclaration represents the declaration of a parameter in
423         a method or constructor declaration
424         @attribute Modifiers List of Modifiers
425         @attribute Type Parameter type
426         @attribute Ellipsis True if type is declared with '...'. So parameter could
427                             accept more than one argument of this type. Note: The
428                             grammar allows things like this at the moment:
429                             method(int... i, int... j, int k). Java 1.5 allows just
430                             one of these parameters at the end of a parameter list
431         @attribute Name Parameter name
432         @attribute AdditionalArrayDimensions Counts dimensions if parameter is an
433                                              array
434     */
435     ParameterDeclaration =
436         (Modifiers:Modifier)* Type:Type
437         (Ellipsis:["..."])?
438         Name
439         (
440           options{greedy=true;}: "[" "]"
441           {a.setAdditionalArrayDimensions(a.getAdditionalArrayDimensions()+1);}
442         )*;
443
444     ast ParameterDeclaration =
445         AdditionalArrayDimensions:/int;
446
447     /** ASTBlockStatement represents a block which is surrounded by: "{...}". In
448         the Block can contain any Statement. This is used in many contexts:
449         Inside a class definition prefixed with "static": it is a class initializer.
450         Inside a class definition without "static": it is an instance initializer.
451         As the body of a method. As a completely independent braced block of code
452         inside a method:
453         it starts a new scope for variable definitions
454         @attribute Statement Represents any Statements within the Block
455     */
456     BlockStatement implements ("{")=>Statement =
457         "{" (Statements:Statement)* "}";
458
459     /** ASTStatement contains all possible Statements and redirect to
460         the passing rules
461     */
462     interface Statement;
463
464     /** ASTExpressionStatement is a lonely Expression followed by a ";"
465         @attribute Expression The Expression before a ";"
466     */
467     ExpressionStatement implements (Expression ";")=>Statement =
468         Expression:Expression ";";
469
470     /** ASTTypeDeclarationStatement is the capsule of a TypeDeclaration in
471         a Statement
```

```
472         @attribute TypeDeclaration Represents a TypeDeclaration
473     */
474     TypeDeclarationStatement implements Statement =
475         TypeDeclaration:TypeDeclaration;
476
477     /** ASTEmptyStatement represents a single semicolon
478     */
479     EmptyStatement implements Statement =
480         ";";
481
482     /** ASTContinueStatement represents the continue keyword
483         eventually followed by a jump label.
484         @attribute Label Name of the jump label.
485     */
486     ContinueStatement implements Statement =
487         "continue" (Label:Name)? ";";
488
489     /** ASTSwitchDefaultStatement represents the default keyword
490         in a switch statement.
491         */
492     SwitchDefaultStatement astimplements Statement =
493         "default";
494
495     /** ASTBreakStatement represents the break keyword
496         eventually followed by a jump label.
497         @attribute Label Name of the jump label.
498     */
499     BreakStatement implements Statement =
500         "break" (Label:Name)? ";";
501
502     /** ASTReturnStatement: "return a ;". The Expression a is optional
503         @attribute Expression The Expression which is returned is optional
504     */
505     ReturnStatement implements Statement =
506         "return" (Expression:Expression)? ";";
507
508     SwitchStatement implements Statement  =
509         "switch" "(" Expression:Expression ")"
510         "{"
511         (
512           (
513             options{greedy=true;}:
514             (Statements:CaseStatement | Statements:SwitchDefaultStatement) ":"
515           )+
516           (Statements:Statement)*
517         )*
518         "}";
519
520     ast SwitchStatement =
521         Statements:Statement*;
522
523     /** ASTCaseStatement is only the "case a" construct of a switch construct
524         @attribute Expression The Expression which is proofed for a special case
525     */
526     CaseStatement astimplements Statement =
527         "case" Expression:Expression;
528
529     /** ASTThrowStatement represents a construct like this: "throw Expression;"
530         @attribute Expression The "target" in a throwStatement
531     */
532     ThrowStatement implements Statement =
533         "throw" Expression:Expression ";";
534
535     /** ASTAssertStatement is a check for a boolean Expression - the Asserting.
536         If the value of Asserting is false the Message-Expression gets evaluated
537         @attribute Asserting The Asserting is an Expression which gets evaluated
538                              for a boolean value. A false-value throws an
539                              "AssertionError".
```

```
540         @attribute Message   If the value of the Asserting is false, the Message
541                              get evaluated and its value is converted to a String
542                              using string conversion. The "detail message"
543                              of the "AssertionError" is then the value of Message
544     */
545     AssertStatement implements Statement =
546         "assert" Assertion:Expression (":" Message:Expression)? ";";
547
548     /** ASTWhileStatement represents a loop. The Condition gets evaluated and if
549         the value is a boolean "true" the Statement gets evaluated.
550         @attribute Condition The Condition gets evaluated for a boolean value
551         @attribute Statement The Statement gets evaluated if the Condition-value
552                              is true
553     */
554     WhileStatement implements Statement =
555         "while" "(" Condition:Expression ")" Statement:Statement;
556
557     /** ASTDoWhileStatement represents a loop. The Statement is in any case
558         executed once, then the Condition gets evaluated and if the value is
559         a boolean "true" the loop is executed again.
560         @attribute Statement The Statement gets evaluated
561         @attribute Condition The Condition gets evaluated for a boolean value
562     */
563     DoWhileStatement implements Statement =
564         "do" Statement:Statement "while" "(" Condition:Expression ")" ";";
565
566     /** ASTForEachStatement is the enhanced form of a for-loop. The loop gets
567         executed for every element the iterable has. The actual value get "saved"
568         in the VariableDeclaration.
569         @attribute VariableDeclaration Variable that gets a value from the
570                                        iterable expression for each loop execution
571         @attribute Iterable The Iterable is an Expression of the type Iterable
572         @attribute Statement Statement gets executed every time the loop gets
573                              executed
574     */
575     ForEachStatement implements ("for" "(" ParameterDeclaration ":")=>Statement =
576         "for" "("
577           VariableDeclaration:ParameterDeclaration ":" Iterable:Expression
578           //semantic check: don't allow '...' in VariablenDeclaration
579         ")" Statement:Statement;
580
581     /** ASTForStatement is the normal form of a for-loop with an initializer,
582         an boolean-condition, an update expression, and an executive statement
583         @attribute Initialization An Initialization is a ForInitializer
584         @attribute Updates The Update is a list of Expressions which gets evaluated
585                            every time the loop gets executed.
586         @attribute Statement The Statement gets executed if the Condition is true,
587                              absent, or the value of it is not a boolean.
588     */
589     ForStatement implements Statement =
590         "for" "("
591           (
592             (VariableDeclaration)=> Initializations:VariableDeclarationExpression
593             |
594             Initializations:Expression ("," Initializations:Expression)*
595           )?
596           ";" (Condition:Expression)?
597           ";" (Updates:Expression ("," Updates:Expression)*)?
598         ")" Statement:Statement;
599
600     ast ForStatement =
601         Initializations:Expression*;
602
603     /** ASTIfStatement evaluates the Expression of a boolean value. In case it
604         evaluates to true the IfStatement is executed, otherwise the ElseStatement.
605         @attribute Condition The Condition gets evaluated for a boolean value
606         @attribute Statement The Statement gets executed if the Expression value is
607                              true
```

```
608         @attribute ElseStatement The ElseStatement gets evaluated if the Expression
609                                  value is false
610     */
611     IfStatement implements Statement =
612         "if" "(" Condition:Expression ")" SuccessStatement:Statement
613         (options{greedy=true;}: "else" OptionalElseStatement:Statement)?;
614
615     /** ASTLabeledStatement is a Statement where it is possible to "jump" to the
616         Statement using the label.
617         @attribute Label The Label is only the "name" of the label
618         @attribute Statement The Statement followed the Label
619     */
620     LabeledStatement implements Statement =
621         Label:Name ":" Statement:Statement;
622
623     /** ASTSynchronizedStatement locks the Expression and evaluates the Block.
624         @attribute Expression The Expression which gets locked (or gets
625                               synchronized with the Block)
626         @attribute Block The Block gets executed, when the Expression is locked
627     */
628     SynchronizedStatement implements Statement =
629         "synchronized" "(" Expression:Expression ")" Block:BlockStatement;
630
631     /** ASTTryStatement is a capsuled Block which is tried to be executed; possible
632         errors can be "catched" by the catch clauses to be executed instead.
633         The final clause gets executed in every case independent if an error
634         occurred or not and also if an error gets catched by the catch clauses.
635         @attribute Block The Block gets executed and can occur an error
636         @attribute CatchClause The CatchClause can catch a specific error and gets
637                                then executed
638         @attribute FinallyClause The FinallyClause, if exists, gets executed in
639                                  every case
640     */
641     TryStatement implements Statement =
642         "try" Block:BlockStatement
643         (CatchClause:CatchClause)*
644         (FinallyClause:FinallyClause)?;
645
646     /** ASTFinallyClause belongs to a try-catch-finally-block (see TryStatement).
647         @attribute Block See TryStatement
648     */
649     FinallyClause =
650         "finally" Block:BlockStatement;
651
652     /** ASTCatchClause belongs to a try-catch-block (see TryStatement).
653         @attribute Block see TryStatement
654     */
655     CatchClause =
656         "catch" "(" ExceptionVariable:ParameterDeclaration ")" Block:BlockStatement;
657
658     // Expressions
659     // Note that most of these expressions follow the pattern
660     //   thisLevelExpression :
661     //       nextHigherPrecedenceExpression
662     //           (OPERATOR nextHigherPrecedenceExpression)*
663     // which is a standard recursive definition for parsing an expression.
664     // The operators in java have the following precedences:
665     //    lowest  (13)  = *= /= %= += -= <<= >>= >>>= &= = |=
666     //            (12)  ?:
667     //            (11)  ||
668     //            (10)  &&
669     //            ( 9)  |
670     //            ( 8)
671     //            ( 7)  &
672     //            ( 6)  == !=
673     //            ( 5)  < <= > >=
674     //            ( 4)  << >>
675     //            ( 3)  +(binary) -(binary)
```

```
676     //              ( 2)  * / %
677     //              ( 1)  ++ -- +(unary) -(unary)  ~  !  (type)
678     //                    []   () (method call)  . (POINT -- identifier qualification)
679     //                    new   ()  (explicit parenthesis)
680
681     /** The interface ASTExpression represents all java expressions.
682     */
683     interface Expression;
684
685     /** ASTAssignmentExpression represents a "x = y"
686         with a =, +=, -=, *=, /=, %=, >>=, >>>=, <<=, &=,^=, |= as the operator
687         @attribute LeftHand The left hand is a ConditionalExpression, the "x"
688         @attribute RightHand The right hand is an AssignmentExpression, the "y"
689         @attribute AssignmentOperator = (EQUALS), += (PLUSEQUALS), -= (MINUSEQUALS),
690                                         *= (STAREQUALS), /= (SLASHEQUALS),
691                                         %= (PERCENTEQUALS), >>= (GTGTEQUALS),
692                                         >>>= (GTGTGTEQUALS), <<= (LTLTEQUALS),
693                                         &= (ANDEQUALS),^= (ROOFEQUALS),
694                                         |= (PIPEEQUALS)
695     */
696     AssignmentExpression
697     implements (ConditionalExpression
698                 (("="|"+="|"-="|"*="|"/="|"%="|">>="|">>>="|"<<="|"&="|"^="|"|=")
699                 AssignmentExpression)?)=>Expression
700     returns Expression =
701         ret=ConditionalExpression
702         (
703           astscript{!(LeftHand=ret;);}
704           AssignmentOperator:["=" | "+=" | "-=" | "*=" | "/="  | "%="
705                               | ">>=" | ">>>=" | "<<=" | "&=" | "^=" | "|="]
706           RightHand:AssignmentExpression
707         )?;
708
709     ast AssignmentExpression =
710         LeftHand:Expression;
711
712     /*  This classgen is needed to produce an attribute LeftOperand with the
713         type Expression and not ExpressionList like it would be created from
714         the expression rules
715     */
716     ast InfixExpression =
717         LeftOperand:Expression
718         RightOperand:Expression
719         Operator:/int;
720
721     /** ASTConditionalExpression is something like a "if-then-else" Expression in
722         a short way. It looks like "a? b : c.
723         @attribute Condition A LogicalOrExpression which can (must) be boolean
724                              Statement
725         @attribute ifTrue An AssignmentExpression which is executed when Condition
726                           is a "true"-Statement
727         @attribute ifFalse A ConditionalExpression which is executed when Condition
728                            is a "false"-Statement
729     */
730     ConditionalExpression implements (LogicalOrExpression)=>Expression
731     returns Expression =
732         ret=LogicalOrExpression
733         (
734           astscript{!(Condition=ret;);}
735           "?" IfTrue:AssignmentExpression
736           ":" IfFalse:ConditionalExpression
737         )?;
738
739     ast ConditionalExpression =
740         Condition:Expression;
741
742     /** ASTLogicalOrExpression represents a "x || y"
743         with a || as the operator
```

```
744         @attribute LeftOperand The left operand is a LogicalAndExpression, the "x"
745         @attribute RightOperand The right Operand is a LogicalAndExpression, the "y"
746         @attribute Operator || (PIPEPIPE)
747     */
748     LogicalOrExpression:InfixExpression
749     implements (LogicalAndExpression)=>Expression returns Expression =
750         ret=LogicalAndExpression //ret accords to the left operand
751         (
752           astscript{!(LeftOperand=ret;);}
753           Operator:["||"] RightOperand:LogicalAndExpression
754         )*;
755 
756     /** ASTLogicalAndExpression represents a "x && y"
757         with a && as the operator
758         @attribute LeftOperand The left operand is a BitWiseOrExpression, the "x"
759         @attribute RightOperand The right Operand is a BitWiseOrExpression, the "y"
760         @attribute Operator && (ANDAND)
761     */
762     LogicalAndExpression:InfixExpression
763     implements (BitWiseOrExpression)=>Expression returns Expression =
764         ret=BitWiseOrExpression //ret accords to the left operand
765         (
766           astscript{!(LeftOperand=ret;);}
767           Operator:["&&" ] RightOperand:BitWiseOrExpression
768         )*;
769 
770     /** ASTBitWiseOrExpression represents a "x | y"
771         with a | as the operator
772         @attribute LeftOperand The left operand is a XorExpression, the "x"
773         @attribute RightOperand The right Operand is a XorExpression, the "y"
774         @attribute Operator | (PIPE)
775     */
776     BitWiseOrExpression:InfixExpression
777     implements (XorExpression)=>Expression returns Expression =
778         ret=XorExpression //ret accords to the left operand
779         (
780           astscript{!(LeftOperand=ret;);}
781           Operator:["|" ] RightOperand:XorExpression
782         )*;
783 
784     /** ASTXorExpression represents a "x ^ y"
785         with a ^ as the operator
786         @attribute LeftOperand The left operand is a BitWiseAndExpression, the "x"
787         @attribute RightOperand The right Operand is a BitWiseAndExpression, the "y"
788         @attribute Operator ^ (ROOF)
789     */
790     XorExpression:InfixExpression
791     implements (BitWiseAndExpression)=>Expression returns Expression =
792         ret=BitWiseAndExpression //ret accords to the left operand
793         (
794           astscript{!(LeftOperand=ret;);}
795           Operator:["^"] RightOperand:BitWiseAndExpression
796         )*;
797 
798     /** ASTBitWiseAndExpression represents a "x & y"
799         with a & as the operator
800         @attribute LeftOperand The left operand is a EqualityExpression, the "x"
801         @attribute RightOperand The right Operand is a EqualityExpression, the "y"
802         @attribute Operator & (AND)
803     */
804     BitWiseAndExpression:InfixExpression
805     implements (EqualityExpression)=>Expression returns Expression =
806         ret=EqualityExpression //ret accords to the left operand
807         (
808           astscript{!(LeftOperand=ret;);}
809           Operator:["&" ] RightOperand:EqualityExpression
810         )*;
811 
```

```
812     /** ASTEqualityExpression represents a "x == y"
813         with a == or != as the operator
814         @attribute LeftOperand The left operand is a RelationalExpression, the "x"
815         @attribute RightOperand The right operand is a RelationalExpression, the "y"
816         @attribute Operator == (EQUALSEQUALS), != (EXCLAMATIONMARKEQUALS)
817     */
818     EqualityExpression:InfixExpression
819     implements (RelationalExpression)=>Expression returns Expression =
820         ret=RelationalExpression //ret accords to the left operand
821         (
822           astscript{!(LeftOperand=ret;);}
823           Operator:["!=" | "=="] RightOperand:RelationalExpression
824         )*;
825
826     /** ASTRelationalExpression represents a "x > y"
827         with a <, >, <= or >= as the operator
828         @attribute LeftOperand The left operand is a ShiftExpression, the "x"
829         @attribute RightOperand The right operand is a ShiftExpression, the "y"
830                                 or a "instancOfExpression"
831         @attribute Operator < (LT), > (GT), <= (LTEQUALS), >= (GTEQUALS)
832     */
833     RelationalExpression:InfixExpression
834     implements (ShiftExpression)=>Expression returns Expression =
835         ret=ShiftExpression //ret accords to the left operand
836         (
837           (
838             astscript{!(LeftOperand=ret;);}
839             Operator:["<" | ">" | "<=" | ">="] RightOperand:ShiftExpression
840           )*
841           |
842           ret=InstanceOfExpression->[ret]
843         );
844
845     /** ASTInstancOfExpression is a: "Expression instancof Type"
846         @attribute Expression The Expression which is checked to be the Type
847         @attribute Type The right hand, a Type
848     */
849     InstanceOfExpression
850     implements ("instanceof" Type)=>Expression [Expression:Expression] =
851         "instanceof" Type:Type;
852
853     /** ASTShiftExpression represents a "x >> y"
854         with a <<, >> or >>> as the operator
855         @attribute LeftOperand The left operand is a AdditiveExpression, the "x"
856         @attribute RightOperand The right operand is a AdditiveExpression, the "y"
857         @attribute Operator << (LTLT), >> (GTGT), >>> (GTGTGT)
858     */
859     ShiftExpression:InfixExpression
860     implements (AdditiveExpression)=>Expression returns Expression =
861         ret=AdditiveExpression //ret accords to the left operand
862         (
863           astscript{!(LeftOperand=ret;);}
864           Operator:["<<" | ">>" | ">>>"] RightOperand:AdditiveExpression
865         )*;
866
867     /** ASTAdditiveExpression represents a simple "x + y"
868         with a "+" or "-" as the operator
869         @attribute LeftOperand The left operand is a MultiplicativeExpression,
870                                the "x"
871         @attribute RightOperand The right operand is a MultiplicativeExpression,
872                                 the "y"
873         @attribute Operator + (PLUS), - (MINUS)
874     */
875     AdditiveExpression:InfixExpression
876     implements (MultiplicativeExpression)=>Expression returns Expression =
877         ret=MultiplicativeExpression //ret accords to the left operand
878         (
879           astscript{!(LeftOperand=ret;);}
```

```
880           Operator:["+" | "-"] RightOperand:MultiplicativeExpression
881         )*;
882
883     /** ASTMultiplicativeExpression represents a simple "x * y"
884         with a "*", "/" or "%" (= Modulo) as an operand
885         @attribute LeftOperand The left operand is a PrefixExpression, the "x"
886         @attribute RightOperand The right operand is a PrefixExpression, the "y"
887         @attribute Operator * (STAR), / (LASH) , % (PRECENT)
888     */
889     MultiplicativeExpression:InfixExpression
890     implements (PrefixExpression)=>Expression returns Expression =
891         ret = PrefixExpression
892         (
893           astscript { !(LeftOperand=ret;);}
894           Operator:["*"|"/" | "%"] RightOperand:PrefixExpression
895         )*;
896
897     /** ASTPrefixExpression represents a prefix increment, decrement, or
898         negotiation of an expression. This rule conforms approximately to
899         the unaryExpression rule of the antlr version of this grammar.
900         @attribute Expression The expression to apply the operator to.
901         @attribute Op ++ (PLUSPLUS), -- (MINUSMINUS), + (PLUS), - (MINUS),
902                       ~ (TILDE) or ! (EXCLAMATIONMARK)
903     */
904     PrefixExpression
905     implements (("++" | "--" | "-" | "+" | "~" | "!") | CastExpression)=>Expression
906     returns Expression =
907         (
908           (
909             astscript{!();}
910             Operator: ["++"  | "--" | "-" | "+" |  "~" | "!"]
911             Expression:PrefixExpression
912           )
913           |
914           ret=CastExpression
915         );
916
917     /** ASTCastExpression represents a cast of an expression.
918         This rule conforms approximately to the unaryExpressionNotPlusMinus rule
919         of the antlr version of this grammar.
920         @attribute Expression The expression to apply the cast to.
921         @attribute TargetType The cast type
922     */
923     CastExpression implements (
924                     ("(" PrimitiveArrayType ")")=> ("(" PrimitiveArrayType ")")
925                     |
926                     ("(" ComplexArrayType ")" CastExpression)=>
927                     ("(" ComplexArrayType ")" CastExpression)
928                     |
929                     PostfixExpression
930                   )=>Expression returns Expression =
931       (
932         //BinaryNot and LogicalNot handling moved to the rule PrefixExpression
933         //predicate used to skip cases like: (int.class)
934         ("(" PrimitiveArrayType ")" )=>
935         //explicit cast:
936         (
937           astscript{!();}
938           "(" TargetType:PrimitiveArrayType ")" Expression:PrefixExpression
939         )
940         //Have to backtrack to see if operator follows. If no operator
941         //follows, it's a typecast. No semantic checking needed to parse.
942         //If it _looks_ like a cast, it _is_ a cast; else it's a "(expr)"
943         |
944         ( "(" ComplexArrayType ")" CastExpression)=>
945         //explicit cast:
946         (
947           astscript{!();}
```

```
948             "(" TargetType:ComplexArrayType ")" Expression:CastExpression
949           )
950           | ret=PostfixExpression
951         );
952
953       /** ASTPostfixExpression represents a postfix increment or decrement
954           of an expression
955           @attribute Expression The expression to increment/decrement
956           @attribute Op ++ (PLUSPLUS) or -- (MINUSMINUS)
957       */
958       PostfixExpression
959       implements (Primary ("++"|"--")?)=>Expression returns Expression =
960           ret=Primary
961           (astscript { !(Expression=ret;); } Operator:["++"|"--"] )?;
962
963       ast PostfixExpression =
964           Expression:Expression;
965
966       /** ASTConstructorInvocation represents explicit constructor invocations of the
967           own class (e.g. this();) or of the super class (e.g. super();)
968           @attribute TypeArguments Generic arguments
969           @attribute Reference This- or SuperReference
970           @attribute Arguments List of parameters handed to the constructor
971       */
972       ConstructorInvocation implements
973         (TypeArguments? (ThisReference->[null] | SuperReference->[null])
974         "(" (Expression ("," Expression)*)? ")" )=>Expression =
975           //semantic: check that there are no wildcard types in the list
976           (TypeArguments:TypeArguments )?
977           (
978             Reference:ThisReference->[null]
979             | Reference:SuperReference->[null]
980           )
981           "("
982             (Arguments: Expression ("," Arguments: Expression)*)?
983           ")";
984
985       /** This rule is needed for explicit constructor invocations with a
986           qualification like 'new Outer().super()'. Qualified invocations
987           can just used with 'super'.
988       */
989       QualifiedConstructorInvocation: ConstructorInvocation
990       implements (ConstructorInvocation "." )=>Expression [qualification=Expression] =
991           (TypeArguments:TypeArguments )?
992           //semantic: check, if qualification is a legal expression
993           Reference:SuperReference->[qualification]
994           "("
995             (Arguments: Expression ("," Arguments: Expression)*)?
996           ")";
997
998       ast ConstructorInvocation =
999           Reference:Expression;
1000
1001      /** The Primary parser rule is necessary to encapsulate some other rules like
1002          ThisReference, MethodInvocation, FieldAccess... But there are never
1003          produced ASTPrimary objects.
1004      */
1005      Primary returns Expression =
1006          ret=PrimarySuffix
1007          (
1008            "." ret=ThisReference->[ret]
1009            |
1010            "."
1011            (
1012              ((TypeArguments)? MethodInvocationWithSingleName->[null,null])=>
1013              (typeArguments=TypeArguments)?
1014              ret=MethodInvocationWithSingleName->[ret, typeArguments]
1015              |
```

```
1016            (FieldAccess->[null])=>
1017            ret=FieldAccess->[ret]
1018            |
1019            (QualifiedConstructorInvocation->[null])=>
1020            ret=QualifiedConstructorInvocation->[ret]
1021            |
1022            superReference=SuperReference->[ret]  "."
1023            (
1024              ((TypeArguments)? Name "(")=>
1025              (typeArguments=TypeArguments)?
1026              ret=MethodInvocationWithSingleName->[superReference, typeArguments]
1027              |
1028              ret=FieldAccess->[superReference]
1029            )
1030            |
1031            ret=NewExpression->[ret]
1032          )
1033          |
1034          ret=ArrayAccessExpression->[ret]
1035        )*;
1036
1037    /** ASTThisReference represents just the 'this' keyword
1038        @attribute Qualification The qualification is the x in x.this
1039    */
1040    ThisReference implements ("this")=>Expression [Qualification:Expression] =
1041        "this";
1042
1043    /** ASTFieldAccess represents an access to a field.
1044        @attribute Name Name of the field
1045        @attribute Qualification Qualification of the field
1046    */
1047    FieldAccess implements (Name)=>Expression [Qualification:Expression]=
1048        Name;
1049
1050    /** ASTMethodInvocation represents a method invocation
1051        @attribute Name The single name of the method
1052        @attribute Qualification The qualification of the method is e.g. the
1053                                 expression X in: X.qualification()
1054        @attribute TypeArguments The generic arguments of the method
1055    */
1056    MethodInvocationWithSingleName:MethodInvocation
1057    implements (Name "(")=>Expression
1058    [Qualification:Expression, TypeArguments:TypeArguments] =
1059        Name
1060        "("
1061          (Arguments: Expression ("," Arguments: Expression)*)?
1062        ")";
1063
1064    /** ASTMethodInvocation represents a method invocation
1065        @attribute Name The qualified name of the method
1066        @attribute Qualification The qualification of the method is e.g. the
1067                                 expression X in: X.qualification()
1068        @attribute TypeArguments The generic arguments of the method
1069    */
1070    MethodInvocationWithQualifiedName:MethodInvocation
1071    implements (Name (options{greedy=true;}: "." Name)* "(")=>Expression
1072    [ Qualification:Expression, TypeArguments:TypeArguments] =
1073        Name
1074        (options{greedy=true;}: "." Name)*
1075        "("
1076          (Arguments: Expression ("," Arguments: Expression)*)?
1077        ")";
1078
1079    /** ASTArrayAccessExpression represents a simple: "a[b]"
1080        @attribute Receiver The Receiver is the "a" and must not be null
1081        @attribute Component The Component is the b in the brackets and
1082                             must not be null
1083    */
```

```
1084    ArrayAccessExpression
1085    implements ("[" Expression "]")=>Expression [Receiver:Expression] =
1086        "[" Component:Expression "]";
1087
1088    /** PrimarySuffix parser rule is necessary to encapsulate some
1089        other rules to guarantee a correct redirect
1090    */
1091    PrimarySuffix returns Expression =
1092        ((Type|VoidType) "." "class")=> ret=ClassAccess
1093        | (MethodInvocationWithQualifiedName->[null, null])=>
1094          (ret=MethodInvocationWithQualifiedName->[null, null])
1095        | ret=QualifiedName
1096        | ret=Literal
1097        | ret=NewExpression->[null]
1098        | (ConstructorInvocation)=>
1099          ret=ConstructorInvocation
1100        | ret=ThisReference->[null]
1101        | ret=SuperReference->[null]
1102        | ret=ParenthesizedExpression;
1103
1104    /** ASTParenthesized represents any Expression parenthesized with brackets
1105        '(...)'
1106        @attribute Expression The expression between the brackets.
1107    */
1108    ParenthesizedExpression implements ("(" AssignmentExpression ")")=>Expression =
1109        "(" Expression:AssignmentExpression ")";
1110
1111    /** ASTClassAccess represents the direct access to a class
1112        @attribute Type Type represents the Type in a "Type.class"
1113    */
1114    ClassAccess implements ((Type|VoidType) "." "class")=>Expression =
1115        (Type|VoidType) "." "class";
1116
1117    /** ASTSuperReference represents the Literal "super" and a qualification X
1118        used in the form "X.super..." which can be used for SuperMethodInvocations
1119        or FieldAccesses like used in the rule Primary
1120        @attribute Qualification Qualification is the X in "X.super..."
1121    */
1122    SuperReference implements ("super")=>Expression [Qualification:Expression] =
1123        "super";
1124
1125    /** ASTNewExpression represents a ClassInstantiation or an
1126        ArrayInstantiation. It always begins with the keyword 'new'.
1127        There are never ASTNewExpression objects.
1128    */
1129    NewExpression [qualification=Expression] returns Expression =
1130        "new" (typeArgs=TypeArguments)?
1131        (type=ComplexType | type=PrimitiveType)
1132        (
1133          ret=ClassInstantiation->[qualification, typeArgs, type]
1134          | ret=ArrayInstantiation->[qualification, typeArgs, type]
1135        );
1136
1137    /** ASTInstantiation just combines the two classes ASTArrayInstantiation and
1138        ASTClassInstantiation
1139    */
1140    interface Instantiation;
1141
1142    ast Instantiation astextends Expression =
1143        Qualification:Expression
1144        TypeArguments:TypeArguments
1145        Type:Type;
1146
1147    /** ASTClassInstantiation represents the part of a class instantiation
1148        starting from the '('.
1149        @attribute Qualification The outer class if instantiating an inner class
1150        @attribute TypeArguments The generic type arguments
1151        @attribute Type The class which is initialized
```

```
1152        @attribute MemberDeclarations The body of the class
1153        @attribute ConstructorArguments Arguments handed to the constructor
1154    */
1155    ClassInstantiation
1156    [Qualification:Expression, TypeArguments:TypeArguments, Type:Type] =
1157        "("
1158          (ConstructorArguments: Expression
1159          ("," ConstructorArguments: Expression)*)?
1160        ")"
1161        (
1162          options{greedy=true;}:
1163          "{"
1164            (MemberDeclarations:MemberDeclaration | ";")*
1165          "}"
1166          {a.getMemberDeclarations().set_Existent(true);}
1167        )?;
1168
1169    ast ClassInstantiation astimplements Instantiation;
1170
1171    /** ASTArrayInstantiation represents the part of an array instantiation
1172        starting from the '['.
1173        @attribute Qualification The outer class if instantiating an inner class
1174        @attribute TypeArguments The generic type arguments
1175        @attribute Type The class which is initialized
1176        @attribute Initializer Initial values for the Array
1177        @attribute ConstructorArguments Arguments handed to the constructor
1178    */
1179    ArrayInstantiation
1180    [Qualification:Expression, TypeArguments:TypeArguments, Type:Type] =
1181        (
1182          (options{greedy=true;}: "["(Sizes:Expression) "]")+
1183          (options{greedy=true;}: "[" "]" {a.setDims(a.getDims()+1);})*
1184          |
1185          ("[" "]" {a.setDims(a.getDims()+1);})+
1186          Initializer:ArrayInitializer
1187        );
1188
1189    ast ArrayInstantiation astimplements Instantiation =
1190        Dims:/int;
1191
1192    // Overrides mc.types.Types
1193    /** ASTQualifiedName represents a possibly qualified name in the AST.
1194        The different parts of a qualified name are separated by '.'; they
1195        are stored in an ASTStringList.
1196        @attribute parts A list of type ASTStringList concluding all name parts
1197    */
1198    QualifiedName implements Expression =
1199        parts:Name (options{greedy=true;}: "." parts:Name)*;
1200
1201    // Overrides mc.literals.Literals
1202    interface Literal extends Expression;
1203
1204    /** ASTNullLiteral represents 'null'
1205    */
1206    NullLiteral implements Literal;
1207
1208    /** ASTBooleanLiteral represents "true" or "false"
1209        @attribute source String-representation (including '"').
1210    */
1211    BooleanLiteral implements Literal;
1212
1213    /** ASTCharLiteral represents any valid character parenthesized with "'".
1214        @attribute source String-representation (including "'").
1215    */
1216    CharLiteral implements Literal;
1217
1218    /** ASTStringLiteral represents any valid character sequence parenthesized
1219        with '"'.
```

```
1220        @attribute source String-representation (including '"').
1221    */
1222    StringLiteral implements Literal;
1223
1224    /** The interface ASTNumericLiteral combines the numeric literal types for
1225        Integer, Long, Float, and Double
1226    */
1227    interface NumericLiteral extends Literal;
1228
1229    /** ASTIntLiteral represents an Integer number.
1230        @attribute source String-representation (including '"').
1231    */
1232    IntLiteral implements NumericLiteral;
1233
1234    /** ASTLongLiteral represents a Long number.
1235        @attribute source String-representation (including '"').
1236    */
1237    LongLiteral implements NumericLiteral;
1238
1239    /** ASTFloatLiteral represents a Float number.
1240        @attribute source String-representation (including '"').
1241    */
1242    FloatLiteral implements NumericLiteral;
1243
1244    /** ASTDoubleLiteral represents a Double number.
1245        @attribute source String-representation (including '"').
1246    */
1247    DoubleLiteral implements NumericLiteral;
1248
1249    concept attributes {
1250        syn JavaTypeEntry: /mc.javadsl.ets.entries.JavaTypeEntry;
1251        global SymbolTable: /interfaces2.helper.SymbolTableInterface;
1252    }
1253
1254    concept antlr {
1255        lexer java {
1256          /** flag for enabling the "assert" keyword */
1257          private boolean assertEnabled = true;
1258          /** flag for enabling the "enum" keyword */
1259          private boolean enumEnabled = true;
1260          /** Enable the "assert" keyword */
1261          public void enableAssert() { assertEnabled = true; }
1262          /** Disable the "assert" keyword */
1263          public void disableAssert() { assertEnabled = false; }
1264          /** Query the "assert" keyword state */
1265          public boolean isAssertEnabled() { return assertEnabled; }
1266          /** Enable the "enum" keyword */
1267          public void enableEnum() { enumEnabled = true; }
1268          /** Disable the "enum" keyword */
1269          public void disableEnum() { enumEnabled = false; }
1270          /** Query the "enum" keyword state */
1271          public boolean isEnumEnabled() { return enumEnabled; }
1272
1273          /** This method overrides one in the CharScannner. This is necessary,
1274           *  because the literal table returns for "enum" and "assert" always
1275           *  LITERAL_enum and LITERAL_assert. Although in case this features
1276           *  has been disabled.
1277          */
1278          @Override
1279          public int testLiteralsTable(int ttype) {
1280            int ret;
1281            ret = super.testLiteralsTable(ttype);
1282            if (!assertEnabled
1283                && "assert".equals(
1284                        new String(text.getBuffer(), 0, text.length()))) {
1285              ret = Name;
1286            }
1287            if (!enumEnabled
```

```
1288                    && "enum".equals(
1289                            new String(text.getBuffer(), 0, text.length())) {
1290                ret = Name;
1291              }
1292              return ret;
1293            }
1294          }
1295      }
1296
1297  }
```

Quellcode C.11: MontiCore-Grammatik für Java in der Version 5.0 (in Zusammenarbeit mit [FM07])

Anhang D

Übersicht der Templates

Die folgenden Abbildungen beschreiben den Aufbau der wichtigsten Templates, die innerhalb des UML/P-Frameworks die in Kapitel 6 dargestellten Konzepte der Codegenerierung realisieren. Ausgenommen sind dabei Templates für die Umsetzung von OCL-Ausdrücken, die in [Hel10, Sch10b] beschrieben sind.

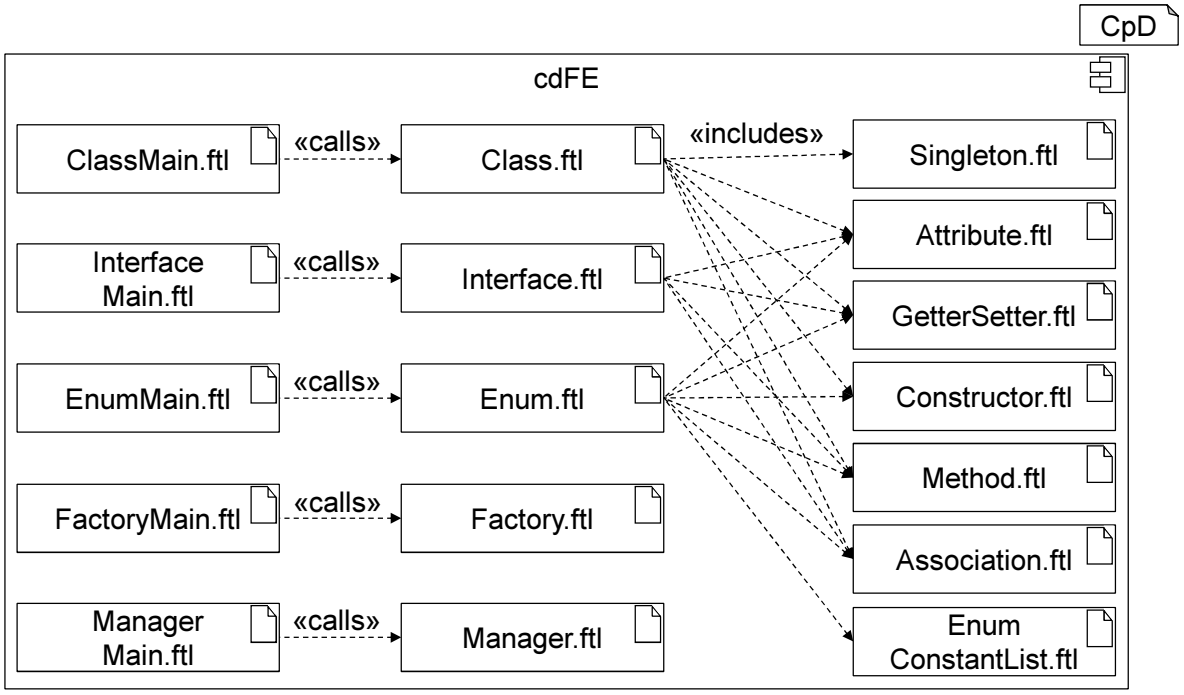


Abbildung D.1: Templates für die Codegenerierung aus Klassendiagrammen (siehe Abschnitt 6.3)

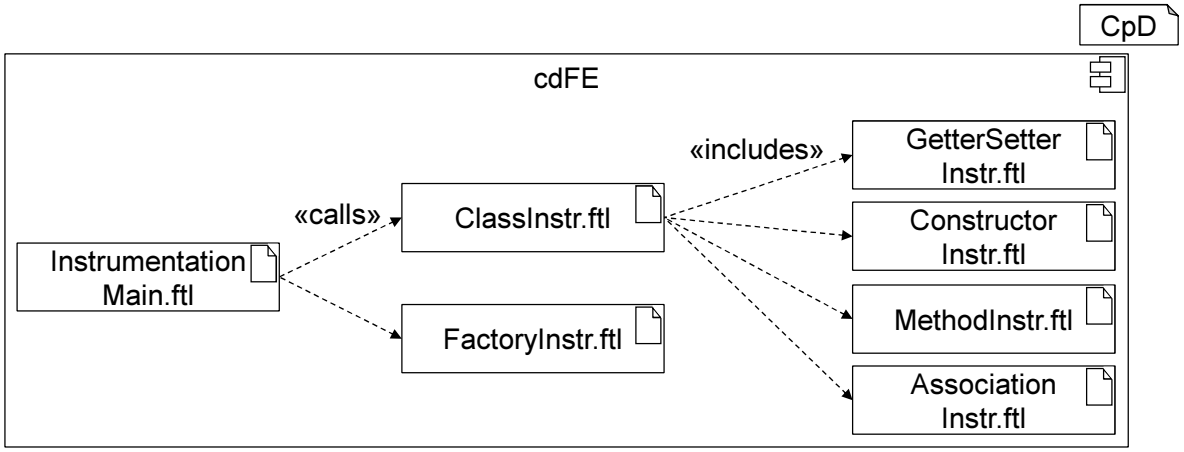


Abbildung D.2: Templates für die Generierung von instrumentierten Code aus Klassendiagrammen (siehe Abschnitt 6.4)

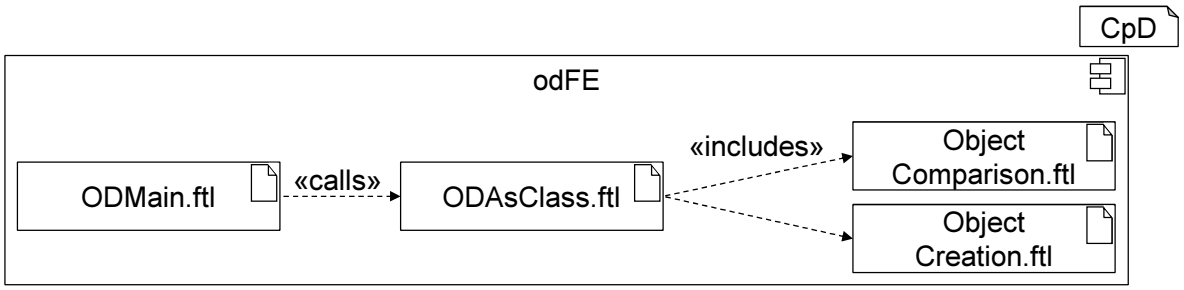


Abbildung D.3: Templates für die Codegenerierung aus Objektdiagrammen (siehe Abschnitt 6.5)

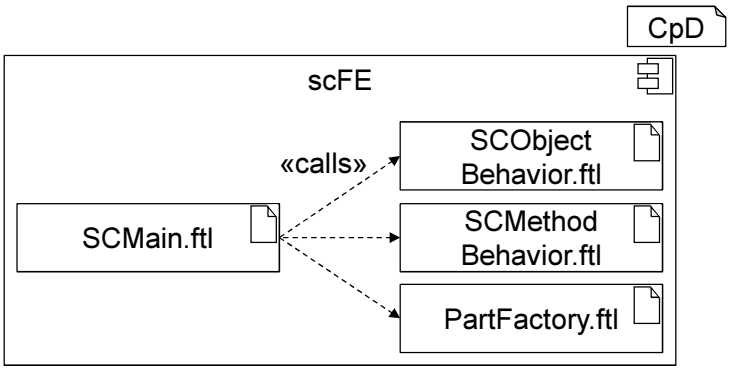


Abbildung D.4: Templates für die Codegenerierung aus Statecharts (siehe Abschnitt 6.3)

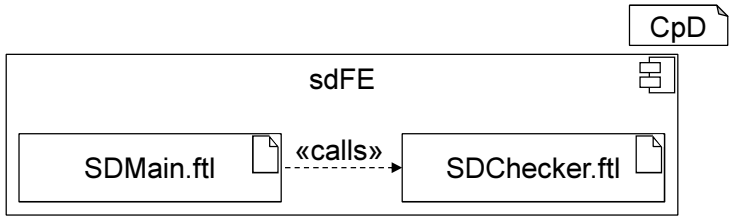


Abbildung D.5: Templates für die Codegenerierung aus Sequenzdiagrammen (siehe Abschnitt 6.5)

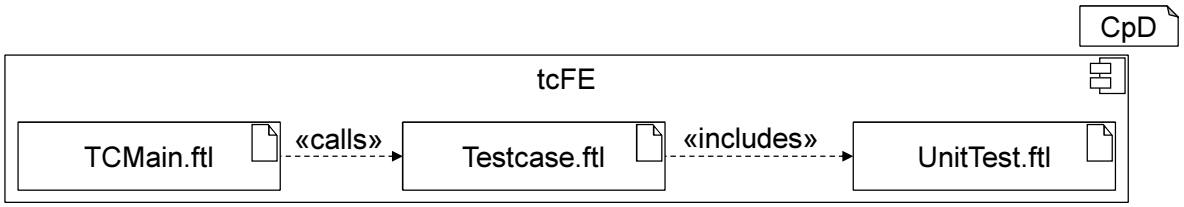


Abbildung D.6: Templates für die Codegenerierung aus der Testspezifikationssprache (siehe Abschnitt 6.5)

Anhang E

Lebenslauf

Name	Schindler
Vorname	Martin
Geburtstag	12.08.1976
Geburtsort	Dortmund
Staatsangehörigkeit	deutsch

seit 2009	Wissenschaftlicher Mitarbeiter am Lehrstuhl Informatik 3 (Software Engineering) RWTH Aachen
2004 - 2009	Wissenschaftlicher Mitarbeiter am Institut für Software Systems Engineering TU Braunschweig
2004	Abschluss als Diplom-Informatiker
2001 - 2002	Praktikum in Indien
1998 - 2004	Studium der Informatik an der TU Braunschweig
1997 - 1998	Studium der Biotechnologie an der TU Braunschweig
1996 - 1997	Zivildienst
1996	Abitur
1987 - 1996	Gymnasium in Unna
1983 - 1987	Grundschule in Fröndenberg

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[ABE+06] David H. Akehurst, Behzad Bordbar, Michael J. Evans, Gareth Howells, Klaus D. McDonald-Maier. *SiTra: Simple Transformations in Java.* In: Oscar Nierstrasz, Jon Whittle, David Harel, Gianna Reggio (Editors), *Model Driven Engineering Languages and Systems*, vol. 4199 of *Lecture Notes in Computer Science (LNCS)*, pp. 351–364, doi: 10.1007/11880240_25. Springer Berlin / Heidelberg, 2006.

[ABGR10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, Indrakshi Ray. *On challenges of model transformation from UML to Alloy.* Software and Systems Modeling (SoSyM), 9(1):69–86, doi: 10.1007/s10270-008-0110-3, 2010.

[AHM07] David H. Akehurst, Gareth Howells, Klaus D. McDonald-Maier. *Implementing associations: UML 2.0 to Java 5.* Software and Systems Modeling (SoSyM), 6(1):3–35, doi: 10.1007/s10270-006-0020-1, 2007.

[ALC08] László Angyal, László Lengyel, Hassan Charaf. *A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering.* In: *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pp. 463–472, doi: 10.1109/ECBS.2008.33, April 2008.

[Ali10] Jauhar Ali. *Implementing Statecharts using Java Enums.* In: *2nd International Conference on Education Technology and Computer (ICETC)*, vol. 4, pp. 413–417, doi: 10.1109/ICETC.2010.5529651, June 2010.

[AM] AndroMDA Website. `http://www.andromda.org/`.

[Amb04] Scott W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0.* Cambridge University Press, 3rd edition, March 2004.

[AS] Artisan Studio Website. `http://www.atego.com/products/artisan-studio/`.

[ASU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AU] ArgoUML Website. `http://argouml.tigris.org/`.

[BA04] Kent Beck, Cynthia Andres. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2004.

[Bad10] Omar Badreddin. *Umple: a Model-Oriented Programming Language*. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pp. 337–338, doi: 10.1145/1810295.1810381. ACM, New York, NY, USA, 2010.

[BB02] Wendy Boggs, Michael Boggs. *Mastering UML with Rational Rose 2002*. Sybex Inc., London, January 2002.

[BBB+] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas. *Agile Manifesto*. `http://agilemanifesto.org/`.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, Bernhard Rumpe. *Definition of the System Model*. In: Kevin Lano (Editor), *UML 2 Semantics and Applications*, pp. 61–93, doi: 10.1002/9780470522622.ch4. John Wiley & Sons, 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, Bernhard Rumpe. *Considerations and Rationale for a UML System Model*. In: Kevin Lano (Editor), *UML 2 Semantics and Applications*, pp. 43–60, doi: 10.1002/9780470522622.ch3. John Wiley & Sons, 2009.

[BCGR08] Manfred Broy, María Victoria Cengarle, Hans Grönniger, Bernhard Rumpe. *Modular Description of a Comprehensive Semantics Model for the UML (Version 2.0)*. Technical Report 2008-06, Software Systems Engineering Institute, Braunschweig University of Technology, 2008.

[Bec02] Kent Beck. *Test Driven Development - By Example*. Addison-Wesley Longman, Amsterdam, November 2002.

[BG01] Jean Bézivin, Olivier Gerbé. *Towards a Precise Definition of the OMG/MDA Framework*. International Conference on Automated Software Engineering, pp. 273–280, doi: 10.1109/ASE.2001.989813, 2001.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall, 2nd edition, 1998.

[BJN+06] Manfred Broy, Matthias Jarke, Manfred Nagl, Hans Dieter Rombach, Armin B. Cremers, Jürgen Ebert, Sabine Glesner, Martin Glinz, Michael Goedicke, Gerhard Goos, Volker Gruhn, Wilhelm Hasselbring, Stefan Jähnichen, Stefan Kowalewski, Bernd J. Krämer, Stefan Leue, Claus Lewerentz, Peter Liggesmeyer, Christoph Lüth, Barbara Paech, Helmut A. Partsch, Ilka Philippow, Lutz Prechelt, Andreas Rausch, Willem-Paul de Roever, Bernhard Rumpe, Gudula Rünger, Wilhelm Schäfer, Kurt Schneider, Andy Schürr, Walter F. Tichy, Bernhard Westfechtel, Wolf Zimmermann,

Albert Zündorf. *Dagstuhl-Manifest zur Strategischen Bedeutung des Software Engineering in Deutschland*. In: Manfred Broy, Manfred Nagl, Hans Dieter Rombach, Matthias Jarke (Editors), *Perspectives Workshop*, Dagstuhl Seminar Proceedings 05402. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[BKPS07] Manfred Broy, Ingolf H. Krüger, Alexander Pretschner, Christian Salzmann. *Engineering Automotive Software*. Proceedings of the IEEE, 95(2):356–373, doi: 10.1109/JPROC.2006.888386, February 2007.

[BLW05] Paul Baker, Shiou Loh, Frank Weil. *Model-Driven Engineering in a Large Industrial Context — Motorola Case Study*. In: *Model Driven Engineering Languages and Systems*, vol. 3713 of *Lecture Notes in Computer Science (LNCS)*, pp. 476–491, doi: 10.1007/11557432_36. Springer Berlin / Heidelberg, 2005.

[BM11] Hans Buhl, Marco Meier. *Die Verantwortung der Wirtschaftsinformatik bei IT-Großprojekten*. WIRTSCHAFTSINFORMATIK, 53(2):59–62, doi: 10.1007/s11576-011-0261-7, 2011.

[BNBK06] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, Gabor Karsai. *The Graph Rewriting and Transformation Language: GReAT*. Electronic Communications of the EASST, 1, 2006.

[Bor11] Borland Software Corporation. *Borland Together 2008 - Borland Together Modeling Guide*. Technical Report Release 3, Micro Focus, June 2011.

[BR07] Manfred Broy, Bernhard Rumpe. *Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung*. Informatik-Spektrum, 30(1):3–18, doi: 10.1007/s00287-006-0124-6, 2007.

[BRJ97] Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language for Object-Oriented Development, Documentation Set Version 1.0*. Rationale Software Corporation, Santa Clara, CA, 1997.

[Bro97] Manfred Broy. *The Specification of System Components by State Transition Diagrams*. Technical Report TUM-I9729, Technische Univerität München, 1997.

[Bro06] Manfred Broy. *Challenges in Automotive Software Engineering*. In: *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 33–42, doi: 10.1145/1134285.1134292. ACM, New York, NY, USA, 2006.

[BW08] Achim Brucker, Burkhart Wolff. *HOL-OCL: A Formal Proof Environment for UML/OCL*. In: José Fiadeiro, Paola Inverardi (Editors), *Fundamental Approaches to Software Engineering*, vol. 4961 of *Lecture Notes in Computer Science (LNCS)*, pp. 97–100, doi: 10.1007/978-3-540-78743-3_8. Springer Berlin / Heidelberg, 2008.

[Béz04] Jean Bézivin. *In Search of a Basic Principle for Model Driven Engineering.* UPGRADE - The European Journal for the Informatics Professional, 5(2):21–24, 2004.

[Béz05a] Jean Bézivin. *On the unification power of models.* Software and Systems Modeling (SoSyM), 4(2):171–188, doi: 10.1007/s10270-005-0079-0, May 2005.

[Béz05b] Jean Bézivin. *Some Lessons Learnt in the Building of a Model Engineering Platform.* In: *4th Workshop in Software Model Engineering (WISME).* Montego Bay, Jamaica, 2005.

[CCD+02] Alessandra Cavarra, Charles Crichton, Jim Davies, Alan Hartman, Laurent Mounier. *Using UML for Automatic Test Generation.* In: *International Symposium on Software Testing and Analysis (ISSTA).* Springer, 2002.

[CD08a] Michelle Crane, Jürgen Dingel. *Towards a Formal Account of a Foundational Subset for Executable UML Models.* In: Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, Markus Völter (Editors), *Model Driven Engineering Languages and Systems*, vol. 5301 of *Lecture Notes in Computer Science (LNCS)*, pp. 675–689, doi: 10.1007/978-3-540-87875-9_47. Springer Berlin / Heidelberg, 2008.

[CD08b] Michelle L. Crane, Jürgen Dingel. *Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities.* In: *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, CASCON '08, pp. 96–110, doi: 10.1145/1463788.1463799. ACM, New York, NY, USA, 2008.

[CE00] Krzysztof Czarnecki, Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[CH03] Krzysztof Czarnecki, Simon Helsen. *Classification of Model Transformation Approaches.* In: *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture.* Anaheim, California, USA, 2003.

[CH06] Krzysztof Czarnecki, Simon Helsen. *Feature-based Survey of Model Transformation Approaches.* IBM Systems Journal, 45(3):621–645, doi: 10.1147/sj.453.0621, July 2006.

[CJKW07] Steve Cook, Gareth Jones, Stuart Kent, Alan C. Wills. *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley Longman, Amsterdam, 2007.

[Dem79] Tom Demarco. *Structured Analysis and System Specification.* Prentice Hall PTR, Upper Saddle River, New Jersey, USA, 1979.

[Die10] Reinhard Diestel. *Graphentheorie.* Springer Berlin / Heidelberg, 4th edition, August 2010.

[DRRS09] Michael Dukaczewski, Dirk Reiss, Bernhard Rumpe, Mark Stein. *MontiWeb - Modular Development of Web Information Systems.* In: Matti Rossi, Jonathan Sprinkle, Jeff Gray, Juha-Pekka Tolvanen (Editors), *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, DSM'09, 2009.

[EA] Enterprise Architect Website. `http://www.sparxsystems.com.au/`.

[EBNF] EBNF Spezifikation ISO/IEC 14977:1996. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153`.

[Ecl] Eclipse Website. `http://www.eclipse.org/`.

[EES09] Holger Eichelberger, Yilmaz Eldogan, Klaus Schmid. *A Comprehensive Survey of UML Compliance in Current Modelling Tools.* In: Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Editors), *Software Engineering*, vol. 143 of *Lecture Notes in Informatics (LNI)*, pp. 39–50. Gesellschaft für Informatik (GI), March 2009. available at: `http://www.uni-hildesheim.de/index.php?id=1388`.

[EFH+08] Sven Efftinge, Peter Friese, Arno Haase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, Patrick Schönbach, Moritz Eysholdt, Dennis Hübner, Steven Reinisch. *openArchitectureWare User Guide.* Technical Report Version 4.3.1, openArchitectureWare.org, 2008.

[EMF] Eclipse Modeling Framework Project Website. `http://www.eclipse.org/modeling/emf/`.

[ET] EMFText Website. `http://emftext.org/`.

[FGDS06] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, Arnor Solberg. *Model-Driven Development Using UML 2.0: Promises and Pitfalls.* Computer, 39(2):59–66, doi: 10.1109/MC.2006.65, February 2006.

[FK03] Kresimir Fertalj, Damir Kalpic. *Preservation of Manually Written Source Code in Case of Repeated Code Generation.* In: *Proceedings of the IASTED International Conference on Computer Science and Technology*, pp. 38–42, May 2003.

[FL72] Wallace Feurzeig, George Lukas. *LOGO - A programming language for teaching mathematics.* In: *Educational Technology 12*, pp. 39–46, March 1972.

[FM] FreeMarker Website. `http://freemarker.org/`.

[FM07] Christoph Ficek, Fabian May. *Umsetzung der Java 5 Grammatik für MontiCore.* Studienarbeit, Software Systems Engineering Institute, Braunschweig University of Technology, 2007.

[For10] Andrew Forward. *The Convergence of Modeling and Programming: Facilitating the Representation of Attributes and Associations in the Umple Model-Oriented Programming Language*. PhD Thesis, University of Ottawa. AAT NR74233, 2010.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman, Boston, MA, USA, 3rd edition, September 2003.

[FP10] Martin Fowler, Rebecca Parsons. *Domain Specific Languages*. Addison-Wesley Longman, Amsterdam, September 2010.

[FR07] Robert France, Bernhard Rumpe. *Model-driven Development of Complex Software: A Research Roadmap*. In: *2007 Future of Software Engineering*, FOSE '07, pp. 37–54, doi: 10.1109/FOSE.2007.14. IEEE Computer Society, Washington, DC, USA, 2007.

[Fri06] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, 3rd edition, August 2006.

[Fuj] Fujaba Website. `http://www.fujaba.de/`.

[GA] Gentleware Apollo Website. `http://www.gentleware.com/apollo.html`.

[GB99] Jilles van Gurp, Jan Bosch. *On the Implementation of Finite State Machines*. In: *Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications*, pp. 172–178. Press, 1999.

[GBR07] Martin Gogolla, Fabian Büttner, Mark Richters. *USE: A UML-Based Specification Environment for Validating UML and OCL*. Science of Computer Programming, 69(1-3):27–34, doi: 10.1016/j.scico.2007.01.013, 2007.

[GC03] Joseph D. Gradecki, Jim Cole. *Mastering Apache Velocity*. Wiley, 2003.

[GDL03] Gonzalo Génova, Carlos Ruiz Del Castillo, Juan Llorens. *Mapping UML Associations into Java Code*. Journal of Object Technology, 2(5):135–162, 2003.

[GF10] Generative Software GmbH, FZI (Forschungszentrum Informatik, Karlsruhe). *Umfrage zu Verbreitung und Einsatz modellgetriebener Softwareentwicklung - Abschlussbericht*, 2010. `http://www.mdsd-umfrage.de/mdsd-report-2010.pdf`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GJS05] James Gosling, Bill Joy, Guy L. Steele. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[GK10] Joel Greenyer, Ekkart Kindler. *Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars.* Software and Systems Modeling (SoSyM), 9(1):21–46, doi: 10.1007/s10270-009-0121-8, 2010.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. *MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen.* Technical Report 2006-04, Software Systems Engineering Institute, Braunschweig University of Technology, 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. *Textbased Modeling.* In: *4th International Workshop on Software Language Engineering*, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. *MontiCore: A Framework for the Development of Textual Domain Specific Languages.* In: *30th International Conference on Software Engineering*, ICSE'08, pp. 925–926. Leipzig, Germany, May 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler. *Integration von Modellen in einen codebasierten Softwareentwicklungsprozess.* In: Heinrich C. Mayr, Ruth Breu (Editors), *Proceedings of Modellierung 2006*, vol. P-82 of *Lecture Notes in Informatics (LNI)*, pp. 67–81. Gesellschaft für Informatik (GI), 2006.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Longman, Amsterdam, March 2009.

[Grö10] Hans Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten.* PhD Thesis, RWTH Aachen University. Aachener Informatik-Berichte, Software Engineering Band 4, Shaker Verlag, 2010.

[GS79] Christopher P. Gane, Trish Sarson. *Structured Systems Analysis: Tools and Techniques.* Prentice-Hall Software. Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[Gul09] Jens Gulden. *Minimal-invasive generative Entwicklung von Modellierungswerkzeugen mit dem Eclipse Graphical Modeling Framework (GMF).* In: Stefan Fischer, Erik Maehle, Rüdiger Reischuk (Editors), *INFORMATIK 2009 - Im Focus das Leben*, vol. P-154 of *Lecture Notes in Informatics (LNI)*, pp. 3001–3015. Gesellschaft für Informatik (GI), September 2009.

[Göd31] Kurt Gödel. *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I.* Monatshefte für Mathematik, 38(1):173–198, doi: 10.1007/BF01700692, 1931.

[Hab04] Mehran Habibi. *Java Regular Expressions: Taming the Java.Util.Regex Engine.* Apress, February 2004.

[Har87] David Harel. *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, 8(3):231–274, doi: 10.1016/0167-6423(87)90035-9, June 1987.

[Hel10] Ulrich Helker. *Integration der Object Constraint Language in die UML/P.* Diplomarbeit, Department of Computer Science 3 (Software Engineering), RWTH Aachen University, 2010.

[Hen05] Brian Henderson-Sellers. *UML – the Good, the Bad or the Ugly? Perspectives from a panel of experts.* Software and Systems Modeling (SoSyM), 4(1):4–13, doi: 10.1007/s10270-004-0076-8, 2005.

[Her06] Christoph Herrmann. *Online Software Transformation Platform.* Diplomarbeit, Software Systems Engineering Institute, Braunschweig University of Technology, 2006.

[HJSW10] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende. *Closing the Gap between Modelling and Java.* In: Mark van den Brand, Dragan Gaševic, Jeff Gray (Editors), *Software Language Engineering*, vol. 5969 of *Lecture Notes in Computer Science (LNCS)*, pp. 374–383, doi: 10.1007/978-3-642-12107-4_25. Springer Berlin / Heidelberg, 2010.

[HKGV10] Zef Hemel, Lennart Kats, Danny Groenewegen, Eelco Visser. *Code generation by model transformation: a case study in transformation modularity.* Software and Systems Modeling (SoSyM), 9(3):375–402, doi: 10.1007/s10270-009-0136-1, 2010.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. *An Algebraic View on the Semantics of Model Composition.* In: David H. Akehurst, Régis Vogel, Richard F. Paige (Editors), *Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, Lecture Notes in Computer Science (LNCS) 4530, pp. 99–113, doi: 10.1007/978-3-540-72901-3_8. Springer Berlin / Heidelberg, Haifa, Israel, June 2007.

[HMU02] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie.* Addison-Wesley, Munich, Germany, 2nd edition, 2002.

[Hol05] Steve Holzner. *Ant: The Definitive Guide.* O'Reilly Media, 2nd edition, April 2005.

[HR00] David Harel, Bernhard Rumpe. *Modeling Languages: Syntax, Semantics and All That Stuff (Part I: The Basic Stuff).* Technical Report MCS00-16, Mathematics & Computer Sience, Weizmann Institute of Sience, 2000.

[HR04a] Grégoire Hamon, John Rushby. *An Operational Semantics for Stateflow.* In: Michel Wermelinger, Tiziana Margaria-Steffen (Editors), *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science (LNCS) 2984, pp. 229–243, doi: 10.1007/978-3-540-24721-0_17. Springer Berlin / Heidelberg, 2004.

[HR04b] David Harel, Bernhard Rumpe. *Meaningful Modeling: What's the Semantics of "Semantics"?* Computer, 37(10):64–72, 2004.

[HRR10] Arne Haber, Jan Oliver Ringert, Bernhard Rumpe. *Towards Architectural Programming of Embedded Systems.* In: *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI*, pp. 13–22. fortiss GmbH, Munich, Germany, February 2010.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer. *Delta Modeling for Software Architectures.* In: *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pp. 1–10. fortiss GmbH, Munich, Germany, February 2011.

[HRW11] John Hutchinson, Mark Rouncefield, Jon Whittle. *Model-Driven Engineering Practices in Industry.* In: *Proceeding of the 33rd International Conference on Software Engineering*, ICSE'11, pp. 633–642, doi: 10.1145/1985793.1985882. ACM, New York, NY, USA, 2011.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, Steinar Kristoffersen. *Empirical Assessment of MDE in Industry.* In: *Proceeding of the 33rd International Conference on Software Engineering*, ICSE'11, pp. 471–480, doi: 10.1145/1985793.1985858. ACM, New York, NY, USA, 2011.

[IBM09] IBM. *Rational Rhapsody User Guide.* Technical Report Version 7.5, IBM Corporation, 2009.

[IEEE93] *IEEE Standard for a Software Quality Metrics Methodology.* IEEE Std 1061-1992, doi: 10.1109/IEEESTD.1993.115124, 1993.

[iUML] iUML Website. `http://www.kc.com/PRODUCTS/iuml/`.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev. *ATL: A model transformation tool.* Science of Computer Programming, 72(1-2):31–39, doi: 10.1016/j.scico.2007.08.002, 2008.

[JUH10] Atif Aftab Ahmed Jilani, Muhammad Usman, Zahid Halim. *Model Transformations in Model Driven Architecture.* Universal Journal of Computer Science and Engineering Technology, 1:50–54, October 2010.

[JZM07] Ke Jiang, Lei Zhang, Shigeru Miyake. *An Executable UML with OCL-based Action Semantics Language*. Asia-Pacific Software Engineering Conference, pp. 302–309, doi: 10.1109/ASPEC.2007.21, 2007.

[KBC05] Audris Kalnins, Janis Barzdins, Edgars Celms. *Model Transformation Language MOLA*. In: Uwe Aßmann, Mehmet Aksit, Arend Rensink (Editors), *Model Driven Architecture*, vol. 3599 of *Lecture Notes in Computer Science (LNCS)*, p. 900, doi: 10.1007/11538097_5. Springer Berlin / Heidelberg, 2005.

[Ken04] Kennedy Carter Limited. *Supporting Model Driven Architecture with eXecutable UML*. Technical Report CTN 80 v2.2, November 2004.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. *An Overview of AspectJ*. In: Jørgen Knudsen (Editor), *ECOOP 2001 — Object-Oriented Programming*, vol. 2072 of *Lecture Notes in Computer Science (LNCS)*, pp. 327–354, doi: 10.1007/3-540-45337-7_18. Springer Berlin / Heidelberg, 2001.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, Steven Völkel. *Design Guidelines for Domain Specific Languages*. In: Matti Rossi, Jonathan Sprinkle, Jeff Gray, Juha-Pekka Tolvanen (Editors), *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, DSM'09, 2009.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin. *Aspect-Oriented Programming*. In: *ECOOP*. Springer, 1997.

[KNNZ99] Thomas Klein, Ulrich A. Nickel, Jörg Niere, Albert Zündorf. *From UML to Java And Back Again*. Technical Report tr-ri-00-216, University of Paderborn, Germany, September 1999.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, Fiona A. C. Polack. *Eclipse Development Tools for Epsilon*. In: *Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.

[KPP08a] Dimitrios Kolovos, Richard Paige, Fiona Polack. *The Epsilon Transformation Language*. In: Antonio Vallecillo, Jeff Gray, Alfonso Pierantonio (Editors), *Theory and Practice of Model Transformations*, vol. 5063 of *Lecture Notes in Computer Science (LNCS)*, pp. 46–60, doi: 10.1007/978-3-540-69927-9_4. Springer Berlin / Heidelberg, 2008.

[KPP08b] Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. *Novel Features in Languages of the Epsilon Model Management Platform*. In: *Proceedings of the 2008 international workshop on Models in software engineering*, MiSE '08, pp. 69–73, doi: 10.1145/1370731.1370748. ACM, New York, NY, USA, 2008.

[KPP09] Dimitrios Kolovos, Richard Paige, Fiona Polack. *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*. In: Jean-Raymond Abrial, Uwe Glässer (Editors), *Rigorous Methods for Software Construction and Analysis*, vol. 5115 of *Lecture Notes in Computer Science (LNCS)*, pp. 204–218, doi: 10.1007/978-3-642-11447-2_13. Springer Berlin / Heidelberg, 2009.

[KR05] Vinay Kulkarni, Sreedhar Reddy. *Model-Driven Development of Enterprise Applications*. In: *UML Modeling Languages and Applications*, vol. 3297 of *Lecture Notes in Computer Science (LNCS)*, pp. 118–128, doi: 10.1007/978-3-540-31797-5_13. Springer Berlin / Heidelberg, 2005.

[KR06] Holger Krahn, Bernhard Rumpe. *Techniques for Lightweight Generator Refactoring*. In: Ralf Lämmel, João Saraiva, Joost Visser (Editors), *Generative and Transformational Techniques in Software Engineering*, Lecture Notes in Computer Science (LNCS) 4143, pp. 437–446, doi: 10.1007/11877028_19. Springer Berlin / Heidelberg, 2006.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. PhD Thesis, RWTH Aachen University. Aachener Informatik-Berichte, Software Engineering Band 1, Shaker Verlag, 2010.

[KRV07a] Holger Krahn, Bernhard Rumpe, Steven Völkel. *Efficient Editor Generation for Compositional DSLs in Eclipse*. In: *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling 2007*, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, Steven Völkel. *Integrated Definition of Abstract and Concrete Syntax for Textual Languages*. In: *Proceedings of Models 2007*, pp. 286–300, 2007.

[KRV08a] Holger Krahn, Bernhard Rumpe, Steven Völkel. *Mit Sprachbaukästen zur schnelleren Softwareentwicklung: Domänenspezifische Sprachen modular entwickeln*. Objektspektrum, 4:42–47, 2008.

[KRV08b] Holger Krahn, Bernhard Rumpe, Steven Völkel. *MontiCore: Modular Development of Textual Domain Specific Languages*. In: *Proceedings of Tools Europe*, vol. 11 of *Lecture Notes in Business Information Processing*. Springer Berlin / Heidelberg, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, Steven Völkel. *MontiCore: a Framework for Compositional Development of Domain Specific Languages*. International Journal on Software Tools for Technology Transfer (STTT), 12(5):353–372, doi: 10.1007/s10009-010-0142-1, September 2010.

[KT08] Steven Kelly, Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.

[Küh05] Thomas Kühne. *What is a Model?* In: Jean Bezivin, Reiko Heckel (Editors), *Language Engineering for Model-Driven Software Development*, Dagstuhl Seminar Proceedings 04101. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[Küh06] Thomas Kühne. *Matters of (Meta-) Modeling.* Software and Systems Modeling (SoSyM), 5(4):369–385, doi: 10.1007/s10270-006-0017-9, 2006.

[KWB03] Anneke G. Kleppe, Jos Warmer, Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman, Boston, MA, USA, 2003.

[Kön05] Alexander Königs. *Model Transformation with Triple Graph Grammars.* In: *Model Transformations in Practice Workshop at MoDELS 2005*, October 2005.

[Lem98] Richard Lemesle. *Transformation Rules Based on Meta-modelling.* In: *Second International Enterprise Distributed Object Computing Workshop*, EDOC'98, pp. 113–122, doi: 10.1109/EDOC.1998.723247, November 1998.

[LG08] Juan de Lara, Esther Guerra. *Pattern-Based Model-to-Model Transformation.* In: Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, Gabriele Taentzer (Editors), *Graph Transformations*, vol. 5214 of *Lecture Notes in Computer Science (LNCS)*, pp. 426–441, doi: 10.1007/978-3-540-87405-8_29. Springer Berlin / Heidelberg, 2008.

[Lig09] Peter Liggesmeyer. *Software-Qualität. Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, 2nd edition, June 2009.

[Lin05] Johannes Link. *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung.* dpunkt, Heidelberg, Germany, 2nd edition, 2005.

[LL77] George Lukas, Joan Lukas. *The LOGO Language: Learning Mathematics Through Programming.* Entelek, June 1977.

[LLP+10] Codrut-Lucian Lazar, Ioan Lazar, Bazil Pârv, Simona-Claudia Motogna, Istvan-Gergely Czibula. *Tool Support for fUML Models.* International Journal of Computers Communications & Control (IJCCC), 5(5):775–782, 2010.

[LLPM06] Björn Lundell, Brian Lings, Anna Persson, Anders Mattsson. *UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2.* In: Oscar Nierstrasz, Jon Whittle, David Harel, Gianna Reggio (Editors), *Model Driven Engineering Languages and Systems*, vol. 4199 of *Lecture Notes in Computer Science (LNCS)*, pp. 619–630, doi: 10.1007/11880240_43. Springer Berlin / Heidelberg, 2006.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, Peter Volgyesi. *The Generic*

Modeling Environment. In: *International Workshop on Intelligent Signal Processing (WISP)*. IEEE, 2001.

[LS06] Michael Lawley, Jim Steel. *Practical Declarative Model Transformation with Tefkat*. In: Jean-Michel Bruel (Editor), *Satellite Events at the MoDELS 2005 Conference*, vol. 3844 of *Lecture Notes in Computer Science (LNCS)*, pp. 139–150, doi: 10.1007/11663430_15. Springer Berlin / Heidelberg, 2006.

[MB02] Stephen J. Mellor, Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman, Boston, MA, USA, 2002.

[MB05] Slaviša Markovic, Thomas Baar. *Refactoring OCL Annotated UML Class Diagrams*. In: Lionel Briand, Clay Williams (Editors), *Model Driven Engineering Languages and Systems*, vol. 3713 of *Lecture Notes in Computer Science (LNCS)*, pp. 280–294, doi: 10.1007/11557432_21. Springer Berlin / Heidelberg, 2005.

[MCG05] Tom Mens, Krzysztof Czarnecki, Pieter Van Gorp. *A Taxonomy of Model Transformation*. In: *Proceedings Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development"*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[MD] MagicDraw UML Website. `http://www.magicdraw.com/`.

[MD08] Parastoo Mohagheghi, Vegard Dehlen. *Where Is the Proof? - A Review of Experiences from Applying MDE in Industry*. In: *Model Driven Architecture – Foundations and Applications*, vol. 5095 of *Lecture Notes in Computer Science (LNCS)*, pp. 432–443, doi: 10.1007/978-3-540-69100-6_31. Springer Berlin / Heidelberg, 2008.

[Mec04] Robert Mecklenburg. *Managing Projects with GNU Make: The Power of GNU Make for Building Anything*. O'Reilly Media, 3rd edition, November 2004.

[MHS05] Marjan Mernik, Jan Heering, Anthony M. Sloane. *When and how to develop domain-specific languages*. ACM Computing Surveys, 37(4):316–344, doi: 10.1145/1118890.1118892, December 2005.

[MRA05] Anthony MacDonald, Danny Russell, Brenton Atchison. *Model-Driven Development within a Legacy System: An Industry Experience Report*. Software Engineering Conference, Australian, pp. 14–22, doi: 10.1109/ASWEC.2005.32, 2005.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe. *A Manifesto for Semantic Model Differencing*. In: Jürgen Dingel, Arnor Solberg (Editors), *Models in Software Engineering*, vol. 6627 of *Lecture Notes in Computer Science (LNCS)*, pp. 194–203, doi: 10.1007/978-3-642-21210-9_19. Springer Berlin / Heidelberg, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe. *CDDiff: Semantic Differencing for Class Diagrams*. In: Mira Mezini (Editor), *ECOOP 2011 – Object-Oriented Programming*, vol. 6813 of *Lecture Notes in Computer Science (LNCS)*, pp. 230–254, doi: 10.1007/978-3-642-22655-7_12. Springer Berlin / Heidelberg, 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe. *Modal Object Diagrams*. In: Mira Mezini (Editor), *ECOOP 2011 – Object-Oriented Programming*, vol. 6813 of *Lecture Notes in Computer Science (LNCS)*, pp. 281–305, doi: 10.1007/978-3-642-22655-7_14. Springer Berlin / Heidelberg, 2011.

[MS] MOFScript Website. `http://www.eclipse.org/gmt/mofscript/`.

[MS91] Stephen J. Mellor, Sally Shlaer. *Object Life Cycles: Modeling the World in States*. Prentice Hall, 1991.

[MU] MetaUML Website. `http://metauml.sourceforge.net/`.

[Nob96] James Noble. *Some Patterns for Relationships*. In: *Technology of Object-Oriented Languages and Systems*. Prentice-Hall, 1996.

[NT05] Iftikhar Azim Niaz, Jiro Tanaka. *An Object-Oriented Approach to generate Java Code from UML Statecharts*. International Journal of Computer & Information Sciences (IJCIS), 6(2), June 2005.

[oAW] openArchitectureWare Website. `http://www.openarchitectureware.com/`.

[obj] objectiF Website. `http://www.microtool.de/objectiF/`.

[OJ90] William F. Opdyke, Ralph E. Johnson. *Refactoring: An aid in designing application frameworks and evolving object-oriented systems*. In: *Proceedings of the Symposium on Object-Oriented Programming emphasizing Practical Applications (SOOPA)*. ACM Press, September 1990.

[OMG03] Object Management Group. *MDA Guide Version 1.0.1*, June 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`.

[OMG04] Object Management Group. *Human-Usable Textual Notation (HUTN) Specification - Version 1.0*, August 2004. `http://www.omg.org/spec/HUTN/1.0/PDF/`.

[OMG06] Object Management Group. *Meta Object Facility (MOF) Core Specification - Version 2.0*, January 2006. `http://www.omg.org/spec/MOF/2.0/PDF/`.

[OMG07] Object Management Group. *MOF 2.0/XMI Mapping, Version 2.1.1*, December 2007. `http://www.omg.org/spec/XMI/2.1.1/PDF/`.

[OMG08a] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1*, January 2008. http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF/.

[OMG08b] Object Management Group. *MOF Model to Text Transformation Language, Version 1.0*, January 2008. http://www.omg.org/spec/MOFM2T/1.0/PDF/.

[OMG10a] Object Management Group. *Action Language for Foundational UML (Alf) - Version 1.0 Beta 1*, October 2010. http://www.omg.org/spec/ALF/1.0/Beta1/PDF.

[OMG10b] Object Management Group. *Object Constraint Language - Version 2.2*, February 2010. http://www.omg.org/spec/OCL/2.2/PDF.

[OMG10c] Object Management Group. *Unified Modeling Language: Infrastructure - Version 2.3*, May 2010. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/.

[OMG10d] Object Management Group. *Unified Modeling Language: Superstructure - Version 2.3*, May 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[OMG11a] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification - Version 1.1*, January 2011. http://www.omg.org/spec/QVT/1.1/PDF/.

[OMG11b] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0*, February 2011. http://www.omg.org/spec/FUML/1.0/PDF/.

[ONG+05] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal, Arne-J. Berre. *Toward Standardised Model to Text Transformations*. In: Alan Hartman, David Kreische (Editors), *Model Driven Architecture – Foundations and Applications*, vol. 3748 of *Lecture Notes in Computer Science (LNCS)*, pp. 239–253, doi: 10.1007/11581741_18. Springer Berlin / Heidelberg, 2005.

[Pap] Papyrus Website. http://www.eclipse.org/modeling/mdt/papyrus/.

[Par72] David L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules*. Commun. ACM, 15(12):1053–1058, doi: 10.1145/361598.361623, 1972.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, Thomas Stauner. *Software Engineering for Automotive Systems: A Roadmap*. In: *FOSE '07: 2007 Future of Software Engineering*, pp. 55–71, doi: 10.1109/FOSE.2007.22. IEEE Computer Society, Washington, DC, USA, 2007.

[PD07] Romuald Pilitowski, Anna Derezinska. *Code Generation and Execution Framework for UML 2.0 Classes and State Machines*. In: Tarek Sobh (Editor), *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, pp. 421–427, doi: 10.1007/978-1-4020-6268-1_75. Springer Netherlands, 2007.

[Pin07] Class Pinkernell. *Kontextbedingungen in Zustandsautomaten mit der OCL.* Diplomarbeit, Software Systems Engineering Institute, Braunschweig University of Technology, 2007.

[Pos] Poseidon for UML Website. `http://www.gentleware.com/`.

[PQ95] Terence Parr, Russell Quong. *ANTLR: A Predicated-LL(k) Parser Generator.* Journal of Software Practice and Experience, 25(7):789–810, July 1995.

[PS07] Class Pinkernell, Sören Schulze. *Integration der Object Constraint Language in die UML/P.* Studienarbeit, Software Systems Engineering Institute, Braunschweig University of Technology, 2007.

[Qua02] Terry Quatrani. *Visual Modeling with Rational Rose 2002 and UML.* Addison-Wesley Longman, Amsterdam, 3rd edition, November 2002.

[Rai05] Chris Raistrick. *Applying MDA and UML in the Development of a Healthcare System.* In: *UML Modeling Languages and Applications*, vol. 3297 of *Lecture Notes in Computer Science (LNCS)*, pp. 203–218, doi: 10.1007/978-3-540-31797-5_21. Springer Berlin / Heidelberg, 2005.

[RBP+91] James Rumbaughand, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Upper Saddle River, NJ, USA, October 1991.

[RFW+04] Chris Raistrick, Paul Francis, John Wright, Colin Carter, Ian Wilkie. *Model Driven Architecture with Executable UML.* Cambridge University Press, New York, NY, USA, 2004.

[RPKP08a] Louis Rose, Richard Paige, Dimitrios Kolovos, Fiona Polack. *Constructing Models with the Human-Usable Textual Notation.* In: Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, Markus Völter (Editors), *Model Driven Engineering Languages and Systems*, vol. 5301 of *Lecture Notes in Computer Science (LNCS)*, pp. 249–263, doi: 10.1007/978-3-540-87875-9_18. Springer Berlin / Heidelberg, 2008.

[RPKP08b] Louis Rose, Richard Paige, Dimitrios Kolovos, Fiona Polack. *The Epsilon Generation Language.* In: Ina Schieferdecker, Alan Hartman (Editors), *Model Driven Architecture – Foundations and Applications*, vol. 5095 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–16, doi: 10.1007/978-3-540-69100-6_1. Springer Berlin / Heidelberg, 2008.

[RQZ07] Chris Rupp, Stefan Queins, Barbara Zengler. *UML 2 glasklar: Praxiswissen für die UML-Modellierung.* Hanser, Munich, Germany, 3rd edition, 2007.

[RR] IBM Rational Rhapsody Website. `http://www.ibm.com/software/awdtools/rhapsody/`.

[Rum04a] Bernhard Rumpe. *Modellierung mit UML*. Springer Berlin / Heidelberg, 2004.

[Rum04b] Bernhard Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer Berlin / Heidelberg, 2004.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. PhD Thesis, Technische Universität München. 1996.

[Rum02] Bernhard Rumpe. *≪Java≫ OCL Based on New Presentation of the OCL-Syntax*. In: Tony Clark, Jos Warmer (Editors), *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, vol. 2263 of *Lecture Notes in Computer Science (LNCS)*, pp. 451–454, doi: 10.1007/3-540-45669-4_10. Springer Berlin / Heidelberg, 2002.

[Rum03] Bernhard Rumpe. *Model-Based Testing of Object-Oriented Systems*. In: Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, Willem-Paul de Roever (Editors), *Formal Methods for Components and Objects*, vol. 2852 of *Lecture Notes in Computer Science (LNCS)*, pp. 380–402. Springer Verlag, 2003.

[RVR+09] Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, Jeremy Bennett, Linus Tolke, Michiel van der Wulp. *ArgoUML User Manual - A tutorial and reference description*. Technical Report ArgoUML-0.32, 2009.

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Longman, Amsterdam, 2nd edition, December 2008.

[Sch95] Andy Schürr. *Specification of Graph Translators with Triple Graph Grammars*. In: Ernst Mayr, Gunther Schmidt, Gottfried Tinhofer (Editors), *Graph-Theoretic Concepts in Computer Science*, vol. 903 of *Lecture Notes in Computer Science (LNCS)*, pp. 151–163, doi: 10.1007/3-540-59071-4_45. Springer Berlin / Heidelberg, 1995.

[Sch06] Douglas C. Schmidt. *Guest Editor's Introduction: Model-Driven Engineering*. Computer, 39(2):25–31, doi: 10.1109/MC.2006.58, February 2006.

[Sch10a] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, September 2010.

[Sch10b] Fabian Schwittlinsky. *Integration der OCL/P-Codegenerierung in die UML/P*. Bachelorarbeit, Department of Computer Science 3 (Software Engineering), RWTH Aachen University, 2010.

[Sei03] Ed Seidewitz. *What Models Mean*. IEEE Software, 20(5):26–32, doi: 10.1109/MS.2003.1231147, September 2003.

[Sel03] Bran Selic. *The Pragmatics of Model-Driven Development.* IEEE Software, 20(5):19–25, doi: 10.1109/MS.2003.1231146, September 2003.

[Sel06] Bran Selic. *Model-Driven Development: Its Essence and Opportunities.* IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 0:313–319, doi: 10.1109/ISORC.2006.54, 2006.

[SM91] Sally Shlaer, Stephen J. Mellor. *Object Life Cycles: Modeling the World in States.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Spa11] Sparx Systems Pty Ltd. *Enterprise Architect User Guide.* Technical Report, June 2011.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie.* Springer 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*, vol. 4199 of *Lecture Notes in Computer Science (LNCS)*, pp. 57–72, doi: 10.1007/11880240_5. Springer Berlin / Heidelberg, 2006.

[SV06] Thomas Stahl, Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley, Chichester, UK, May 2006.

[SV09] Martin Schindler, Steven Völkel. *Deliverable D4.4c: Validation Metrics - Prototype Tool.* Workpackage 4: Verification and Validation, MODELPLEX Project, 2009.

[TB03] Dave Thomas, Brian M. Barry. *Model Driven Development: The Case for Domain Oriented Programming.* In: *Companion of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'03, pp. 2–7, doi: 10.1145/949344.949346. ACM, New York, NY, USA, 2003.

[TEG+05] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Larav, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Daniel Varro, Szilvia Varro-Gyapay. *Model Transformation by Graph Transformation: A Comparative Study.* In: *Model Transformations in Practice Workshop at MoDELS 2005*, October 2005.

[Tho04] Dave Thomas. *MDA: Revenge of the Modelers or UML Utopia?* IEEE Software, 21:15–17, doi: 10.1109/MS.2004.1293067, 2004.

[Tog] Together Website. `http://www.borland.com/us/products/together/`.

[TU] TextUML Website. `http://abstratt.com/textuml/`.

[UG] UMLGraph Website. `http://www.umlgraph.org/`.

[UL] UML Lab Website. `http://www.uml-lab.com/`.

[UML97] UML Partners. *Unified Modeling Language v. 1.0*, January 1997.

[VG07] Markus Völter, Iris Groher. *Handling Variability in Model Transformations and Generators*. In: *7th OOPSLA Workshop on Domain-Specific Modeling*, OOPSLA '07, 2007.

[Vis04] Eelco Visser. *Program Transformation with Stratego/XT*. In: Christian Lengauer, Don Batory, Charles Consel, Martin Odersky (Editors), *Domain-Specific Program Generation*, vol. 3016 of *Lecture Notes in Computer Science (LNCS)*, pp. 315–349, doi: 10.1007/978-3-540-25935-0_13. Springer Berlin / Heidelberg, 2004.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. PhD Thesis, Braunschweig University of Technology. Aachener Informatik-Berichte, Software Engineering Band 9, Shaker Verlag, 2011.

[WWM+07] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin van den Berg, Kim Fleer, David Nelson, Michael Wells, Brian Mastenbrook. *Experiences in Deploying Model-Driven Engineering*. In: Emmanuel Gaudin, Elie Najm, Rick Reed (Editors), *SDL 2007: Design for Dependable Systems*, vol. 4745 of *Lecture Notes in Computer Science (LNCS)*, pp. 35–53, doi: 10.1007/978-3-540-74984-4_3. Springer Berlin / Heidelberg, 2007.

[XT] XText Website. `http://www.xtext.org/`.

[yUML] yUML Website. `http://www.yuml.me/`.

Stichwortverzeichnis

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 [Rum11], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06] 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.

Generative Software Engineering

The UML/P language family [Rum12, Rum11] 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]. 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] we show how this looks like and 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)

Many of our contributions build on UML/P described in the two books [Rum11] and [Rum12] are 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 (ADs) [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. 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 in discussed in [PFR02] on product line annotations for UML and to more general discussions and insights on how to use meta-modeling for defining and adapting the UML [EFLR99], [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] describes an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools

can be defined in modular forms [KRV08, 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 examined 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.

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 [RRW13]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11] 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. 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 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].

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. [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 on class diagrams in [CGR08]. [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] embodies 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 comprehensible sets of such transformations support software development, maintenance and [LRSS10] technologies for evolving models within a language and across languages and linking 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 [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)

Many products 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 the commonalities as well as the 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 added (that sometimes also modify the core). A set of 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] describes 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]. And 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 [RRW13]. 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 state machines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using state machines 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 specifications concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [THR+13] 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 [RRW12] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13] that perfectly fits 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. 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. Application classes like Cyber-Physical Systems [KRS12], 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. We tackle these challenges by perusing a model-based, generative approach [PR13]. 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.,c collecting data from temperature, cars etc. are easily developed.

References

[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, October 2007.

[BCGR09a] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, 2009.

[BCGR09b] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, 2009.

[BCR07a] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, February 2007.

[BCR07b] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, 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 H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, TUM-I9737, TU Munich, 1997.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In M. Schader and A. Korthaus, editors, *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*. Physica Verlag, Heidelberg, 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 M. Broy and B. Rumpe, editors, *RTSE '97: Proceedings of the International Workshop on Requirements Targeting Software and Systems Engineering*, LNCS 1526, pages 43–68, Bernried, Germany, October 1998. Springer.

[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 *Proceedings of the 10th Workshop on Automotive Software Engineering (ASE 2012)*, pages 789–798, Braunschweig, Germany, September 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*. Springer, 2012.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, CfG Fakultät, TU Braunschweig, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Model Driven Engineering Languages and Systems. Proceedings of MODELS 2009*, LNCS 5795, pages 670–684, Denver, Colorado, USA, October 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*. Kluver Academic Publisher, 1999.

[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] Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-Based Modeling of Buildings and Facilities. In *Proceedings of the 11th International Conference for Enhanced Building Operations (ICEBO' 11)*, New York City, USA, October 2011.

[FPPR12] Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Proceedings of the 7th International Conference on Energy Efficiency in Commercial Buildings (IEECB)*, Frankfurt a. M., Germany, April 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop*, Paderborn, Germany, October 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*, Toulouse, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, Informatik Bericht 2008-01, pages 76–89, CFG Fakultät, TU Braunschweig, March 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TUM, Munich, Germany, 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. Technical Report 2006-04, CfG Fakultät, TU Braunschweig, August 2006.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Proceedings der Modellierung 2006*, Lecture Notes in Informatics LNI P-82, Innsbruck, März 2006. GI-Edition.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TUM, Munich, Germany, 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems. 16th Monterey Workshop*, LNCS 6662, pages 17–32, Redmond, Microsoft Research, 2011. Springer.

[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. 18th International Working Conference, Proceedings, REFSQ 2012*, Essen, Germany, March 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Model Driven Engineering Languages and Systems, Proceedings of MODELS*, LNCS 6394, Oslo, Norway, 2010. Springer.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC), Tokyo*, pages 22–31. ACM, September 2013.

[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 *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 11–18, New York, NY, USA, 2013. ACM.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Proceedings of the Third European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2007), Haifa, Israel*, pages 99–113. Springer, 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 H. Arabnia and H. Reza, editors, *Proceedings of the 2009 International Conference on Software Engineeering in Research and Practice*, Las Vegas, Nevada, USA, 2009.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins (TOPI) at ICSE 2012*, pages 61–66, Zurich, Switzerland, June 2012. IEEE.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, Oct 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In Madhu Singh, Bertrand Meyer, Joseph Gil, and Richard Mitchell, editors, *TOOLS 26, Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Proceedings of International Software Product Lines Conference (SPLC 2011)*. IEEE Computer Society, August 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, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII, fortiss GmbH*, February 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, Oxford, UK, March 2012. Springer.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik in Berlin*, Lecture Notes in Informatics LNI 198, pages 181–192, 27. Februar - 2. März 2012.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.)*, Techreport B-108, Helsinki School of Economics, Orlando, Florida, USA, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Proceedings of the Modelling of the Physical World Workshop MOTPW'12, Innsbruck, October 2012*, pages 2:1–2:6. ACM Digital Library, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*. P. Dini, IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering Band 14. Shaker Verlag Aachen, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, Aachen, Germany, 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Proceedings of the first International Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 323–338. Chapmann & Hall, 1996.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, pages 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In J. Gray, J.-P. Tolvanen, and J. Sprinkle, editors, *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006 (DSM'06), Portland, Oregon USA*, 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 *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM' 07), Montreal, Quebec, Canada*, Technical Report TR-38, pages 8–10, Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007), Nashville, TN, USA, October 2007*, LNCS 4735. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In R. F. Paige and B. Meyer, editors, *Proceedings of the 46th International Conference Objects, Models, Components, Patterns (TOOLS-Europe), Zurich, Switzerland, 2008*, Lecture Notes in Business Information Processing 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 *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 241–270. Springer, October 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proc. Euro. Soft. Eng. Conf. and SIGSOFT Symp. on the Foundations of Soft. Eng. (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 *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 592–607, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Proc. 25th Euro. Conf. on Object Oriented Programming (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 *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 153–167. Springer, 2011.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In G. J. Chastek, editor, *Software Product Lines - Second International Conference, SPLC 2*, LNCS 2379, pages 188–197, San Diego, 2002. Springer.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, LNCS 873. Springer, October 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In J. Davies J. M. Wing, J. Woodcock, editor, *FM'99 - Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing System*, LNCS 1708, pages 96–115. Springer, 1999.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In H. Kilov and K. Baclawski, editors, *Practical foundations of business and system specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[PR13] Antonio Navarro Perez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proc. of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing. Co-located with MODELS 2013, Miami*, Sun SITE Central Europe Workshop Proceedings CEUR 1118, pages 15–24. CEUR-WS.org, 2013.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technical Report TUM-I9510, Technische Universität München, 1995.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In N. Seyff and A. Koziolek, editors, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA), May 6-10, 2013, Karlsruhe, Germany*, pages 10–12, 2013.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, ISBN 3-89675-149-2, 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, Hershey, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever, editor, *Formal Methods for Components and Objects*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future. 9th International Workshop, RISSEF 2002. Venice, Italy, October 2002*, LNCS 2941. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, second edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer, second edition, Juni 2012.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, Aachen, Germany, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 57–76, October 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 *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*, pages 461–466, Karlsruhe, Germany, May 2013. IEEE.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, Aachen, Germany, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, Aachen, Germany, 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 D. Schaefer, editor, *Proceedings of the SESAR Innovation Days*. EUROCONTROL, November 2011.