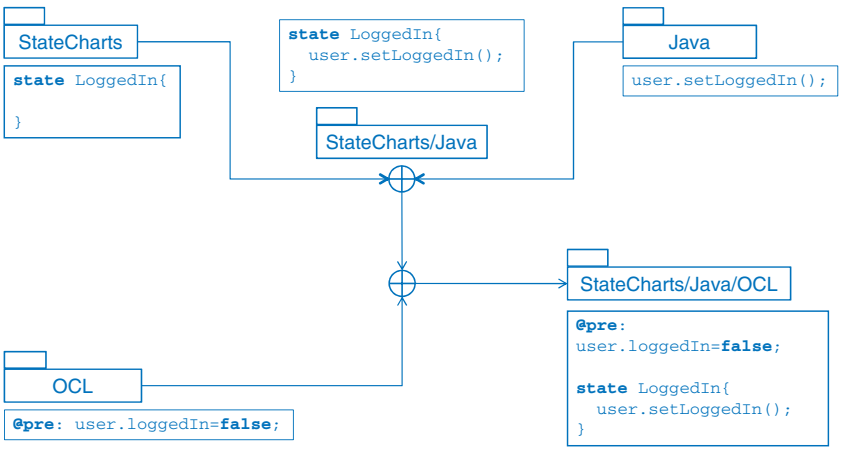
Steven Völkel

Kompositionale Entwicklung domänenspezifischer Sprachen



Aachener Informatik-Berichte,
Software Engineering

Band 9

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

Kompositionale Entwicklung domänenspezifischer Sprachen

Von der Carl-Friedrich-Gauß-Fakultät der
Technischen Universität Carolo-Wilhelmina zu Brauschweig

zur Erlangung des Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte

Dissertation

von Dipl.-Inform. Steven Völkel
geboren am 09. Dezember 1978
in Dessau.

Eingereicht am: 20.04.2011
Mündliche Prüfung am: 15.07.2011
Referentin / Referent: Universitätsprofessorin Dr. rer. nat. Ursula Goltz
Korreferentin / Korreferent: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe

Druckjahr: 2011

[Voe11] S. Völkel
Kompositionale Entwicklung domänenspezifischer Sprachen
Shaker Verlag, ISBN 978-3-8440-0328-4.
Aachener Informatik-Berichte, Software Engineering Band 9. 2011.
www.se-rwth.de/publications

Kurzfassung

Die Nutzung domänenspezifischer Sprachen (engl. Domain-Specific Languages - DSLs) bei der Erstellung komplexer Software erlaubt eine effiziente Entwicklung, einen höheren Grad der Wiederverwendung und verringert sich wiederholende und daher fehlerträchtige Entwicklungsarbeit. Zusätzlich erlauben es DSLs, Domänenexperten direkt in den Entwicklungsprozess einzubeziehen. Demgegenüber steht die oft aufwendige Entwicklung einer Sprache mit all ihren Teilen wie konkrete und abstrakte Syntax, Kontextbedingungen und -überprüfungen sowie unterstützende sprachspezifische Werkzeuge. Zusätzlich ist es aufgrund des speziellen Verwendungszwecks von DSLs üblich, mehrere Sprachen und sogar Sourcecode kooperativ einzusetzen, beginnend mit sequenziellen Werkzeugketten bis hin zur Einbettung (von Teilen) einer Sprache in eine andere. Die Auswahl der eingesetzten Sprachen, der Kombinationsform und der domänenspezifischen Adaption variiert jedoch je nach Projektkontext, Zielsetzung und beteiligten Nutzern.

Um diesen Herausforderungen zu begegnen, wird in der vorliegenden Arbeit ein Ansatz zur kompositionalen Entwicklung textueller domänenspezifischer Sprachen vorgestellt. Hierbei werden zunächst einzelne Sprachen unabhängig voneinander erstellt, um sie in einem weiteren Schritt mit geringem Aufwand auf verschiedenste Weise miteinander zu kombinieren. Als werkzeugtechnische Unterstützung werden dabei Kompositionalitätsmechanismen des MontiCore-Frameworks verwendet und um zusätzliche Konzepte erweitert.

Die wichtigsten Ergebnisse dieser Arbeit lassen sich wie folgt zusammenfassen:

- Eine Übertragung der Mechanismen komponentenbasierter Softwareentwicklung auf die Entwicklung von Sprachen und eine Untersuchung, welche Techniken und Architekturen sich eignen, Sprachen und Software kompositional zu entwickeln.
- Eine Kategorisierung von Sprachkombinationen in Szenarien sowie die Erstellung bzw. Verbesserung der Mechanismen zur kompositionalen Sprachentwicklung in MontiCore.
- Eine Untersuchung, wie die Szenarien anhand der Kompositionalitätsmechanismen bezüglich der konkreten und abstrakten Syntax realisiert werden können.
- Die Entwicklung eines Frameworks zur Erstellung kompositionaler Symboltabellen und Kontextbedingungen mit anschließender Diskussion der Umsetzung der Szenarien bezüglich dieser Sprachbestandteile.
- Die Erweiterung des unterliegenden Grammatikformates zur Spezifikation sprachspezifischer Werkzeuge sowie die Generierung dieser Werkzeuge in Form von auslieferbaren Eclipse-Plugins, wobei auch diese Werkzeuge die Anforderungen kompositionaler Sprachentwicklung erfüllen.

Insgesamt entsteht ein durchgängiger und kohärenter Ansatz zur kompositionalen Sprachentwicklung auf den Ebenen der konkreten Syntax, der abstrakten Syntax, der Symboltabellen, der Kontextbedingungen und der sprachspezifischen Werkzeuge.

Abstract

The use of domain specific languages (DSLs) for the construction of complex software systems allows for an efficient development, a higher degree of reuse, and reduces recurring and thus error-prone labor. Additionally, DSLs permit to directly involve domain experts in a development process. The drawback is the necessary effort for the creation of the languages including concrete and abstract syntax, symbol tables, context conditions, and supporting language-specific tools. In addition, the specific purpose of DSLs often makes it necessary to combine different languages or even code, starting from sequential tool chains up to the embedment of (parts of) languages into others. The involved languages, the form of combination, and the domain specific adaption however depends on the context of the project, the aim, and the involved users.

In order to address these challenges, this theses introduces an approach to the compositional development of textual domain specific languages. In a first step, languages are developed independently and without anticipation of future combinations. In a second step, these languages can be combined in different ways and under minimum effort. As technical basis the modularity mechanisms of the MontiCore-framework are used and extended.

The main results of this theses can be summarized as follows:

- A transfer of the mechanisms of component-based software engineering to the domain of language construction and an investigation which techniques and architectures support a compositional development of languages and software.
- A categorization of language combinations into scenarios and the development and improvement of modularity mechanisms in MontiCore.
- An investigation how the identified scenarios can be realized on the levels of the concrete and abstract syntaxes using the modularity mechanisms.
- The development of a framework for the creation of compositional symbol tables and context conditions and a discussion how the scenarios can be realized for these aspects.
- An extension of the underlying grammar format for the specification of language-specific tools. This specification is used to generated deployable Eclipse-plugins which respect the aspect of compositionality.

Altogether, an integrated and coherent approach for the compositional development of languages on the levels concrete syntax, abstract syntax, symbol tables, context condition checks, and language-specific tools is presented.

Danksagung

Erster Dank geht natürlich an meinen Doktorvater Prof. Dr. Bernhard Rumpe. Mit seinen vielen Anregungen, Ideen und Verbesserungsvorschlägen sowie der gesamten Anleitung während der Promotionszeit hat er wesentlich zum Erfolg dieser Dissertation beigetragen. Neben dieser fachlichen Komponente möchte ich mich im besonderen - und auch im Namen meiner Frau und meiner Kinder - für die menschliche Komponente bedanken. Denn durch die Möglichkeit, meine Dissertation von Braunschweig aus zu beenden, konnte ich Familie und Beruf in Einklang bringen - ein heutzutage leider oftmals viel zu schweres Unterfangen. Auch bei Frau Prof. Dr. Goltz und Herrn Prof. Dr. Balke bedanke ich mich für die Übernahme des Referats bzw. des Vorsitzes des Prüfungsausschusses.

Innerhalb des Kollegenkreises will ich zunächst einigen Menschen danken, die im Sinne dieser Dissertation besonders hervorzuheben sind. Zuerst ist hier Herr Dr. Holger Krahn zu nennen, auf dessen Vorarbeiten ich in dieser Dissertation aufbauen durfte. Die gemeinsamen Jahre als Bürokollegen, die vielen Diskussionen und Ideenaustäusche, die gemeinsamen Papiere und auch die privaten Aspekte waren mir stets ein besonderes Vergnügen. Daneben möchte ich Martin Schindler und Arne Haber für die Bereitschaft danken, den von mir entwickelten Ansatz selbst in den ersten instabilen Versionen einzusetzen. Auch Galina Volkova und Marita Breuer danke ich für die vielen Ideen, Anregungen und Verbesserungswünsche. Weiterhin sind natürlich die Reviewer dieser Arbeit zu erwähnen: Ingo Weisemöller, Dirk Reiss und Andreas Horst. Eure Kommentare trugen wesentlich zur Steigerung der Qualität dieser Arbeit bei. Allen anderen Mitarbeitern, Ex-Mitarbeitern und Studenten danke ich für die vielen Hinweise und anregenden Fragen.

Auch meiner ganzen Familie gebührt besonderer Dank. Meinen Eltern und meinen Großeltern möchte ich hierbei für die Unterstützung auf meinem gesamten Lebensweg danken. Meinen Schwiegereltern und auch meinen Schwieger-Großeltern danke für alles, was sie in den letzten 16 Jahren - immerhin der Hälfte meines Lebens - für uns getan haben.

Nicht zuletzt gilt mein ganz besonderer Dank jedoch meiner Frau Bettina, meiner Tochter Melina und meinem Sohn Moritz. Bettina - Du hast mir in den letzten Jahren immer den Rücken gestärkt, mich unterstützt und verstanden - selbst wenn die Arbeit mal wieder überhandgenommen hat. Melina und Moritz danke ich für jedes einzelne Lächeln, für jede „Kuschel-Einheit“ und für all die anderen schönen Momente, die ich mit Euch erleben darf. Ihr drei seid diejenigen, die meinem Leben den wirklichen Sinn geben.

Inhaltsverzeichnis

I. Grundlagen **1**

1. Einführung **3**
1.1. Motivation der Arbeit . . . 3
1.2. Wichtigste Ziele und Ergebnisse . . . 5
1.3. Nahestehende Arbeiten . . . 6
1.4. Wichtigste Notationselemente . . . 7
1.5. Aufbau dieser Arbeit . . . 7

2. Methodik der komponentenbasierten Softwareentwicklung **11**
2.1. Charakterisierung kompositionaler Softwaresysteme . . . 11
2.2. Anforderungen an eine Komposition . . . 13
2.2.1. Bestandteile einer Komposition . . . 13
2.2.2. Anforderungen an das Komponentenmodell . . . 13
2.2.3. Anforderungen an die Kompositionstechnik . . . 15
2.2.4. Anforderungen an die Kompositionssprache . . . 16
2.3. Vorteile kompositionaler Softwaresysteme . . . 16
2.4. Zusammenfassung . . . 19

3. Einführung in die Sprachentwicklung mit MontiCore **21**
3.1. DSLs und GPLs . . . 21
3.2. Bestandteile einer Sprache . . . 22
3.3. Sprachentwicklung mit MontiCore . . . 24
3.3.1. Das Eingabeformat . . . 25
3.3.2. Ableitung der abstrakten Syntax . . . 27
3.3.3. Parsertechnologie . . . 29
3.3.4. Das DSLTool-Framework . . . 30
3.4. Zusammenfassung . . . 31

II. Kompositionale Sprachentwicklung **33**

4. Anwendungen und Techniken **35**
4.1. Einsatz multipler Sprachen . . . 35
4.1.1. OCL als DSL in der Modellierung . . . 36
4.1.2. SQL als DSL in der Programmierung . . . 36

4.1.3. Kombination von Modellierung und Programmierung: Statecharts und Java . . . 37
4.1.4. Weitere Beispiele . . . 37
4.2. Kompositionalitätsmechanismen des MontiCore-Frameworks . . . 39
4.2.1. Grammatikvererbung . . . 39
4.2.2. Einbettung . . . 40
4.2.3. Sprachaggregation . . . 41
4.3. Anwendungsszenarien kompositionaler Sprachentwicklung . . . 42
4.3.1. Sprachvereinigung . . . 42
4.3.2. Nichtterminal-Erweiterung . . . 42
4.3.3. Nichtterminal-Ersetzung . . . 42
4.3.4. Einbettung einer einzelnen Sprache . . . 42
4.3.5. Einbettung von Sprachalternativen . . . 43
4.3.6. Entwicklung einer erweiterbaren Sprache . . . 43
4.4. Korrelationen zwischen Kompositionsmechanismen und Anwendungsszenarien 43
4.5. Modell- und Sprachkomposition . . . 44
4.5.1. Modellkomposition . . . 44
4.5.2. Sprachkomposition als spezielle Form der Modellkomposition . . . 46
4.5.3. Unterstützung der kompositionalen Modellierung . . . 46
4.6. Aufbau kompositionaler, sprachverarbeitender Werkzeuge . . . 47
4.6.1. Kombination von Sprachen . . . 48
4.6.2. Umsetzung der Anforderungen an kompositionale Systeme . . . 50
4.7. Techniken zur Entwicklung kompositionaler sprachverarbeitender Werkzeuge . 50
4.7.1. Frameworks und Bibliotheken . . . 51
4.7.2. Vererbung und Delegation . . . 51
4.7.3. Fabriken . . . 53
4.7.4. Adaptoren . . . 54
4.7.5. Visitoren . . . 55
4.7.6. Kompilierungspozess und Projektorganisation . . . 55
4.8. Zusammenfassung . . . 57

5. Kompositionale Entwicklung konkreter Syntax 59
5.1. Kompositionalität im Erkennungsprozess . . . 59
5.1.1. Generelle Arbeitsweise von Lexer und Parser . . . 59
5.1.2. Verwendete Technologie . . . 61
5.1.3. Entscheidungsfindung im Parse-Prozess bei Alternativen . . . 62
5.1.4. Arbeitsweise von Lexer und Parser in ANTLR . . . 63
5.1.5. Wechsel der Parser und Lexer bei Einbettung . . . 63
5.2. Konfiguration und Auswahl der Lexer und Parser . . . 65
5.2.1. Konfiguration der Sprachkombination bei Einbettung . . . 65
5.2.2. Auswahl der Lexer und Parser bei Sprachaggregation . . . 66
5.2.3. Auswahl der Lexer und Parser bei Mehrfachvererbung . . . 67
5.3. Sprachvereinigung . . . 68

5.4. Nichtterminal-Erweiterung . . . 70
5.4.1. Umsetzung durch Mehrfachvererbung . . . 70
5.4.2. Umsetzung durch Einbettung . . . 72
5.4.3. Vergleich der Lösungen . . . 76
5.5. Nichtterminal-Ersetzung . . . 77
5.5.1. Umsetzung durch Mehrfachvererbung . . . 77
5.5.2. Umsetzung durch Einbettung . . . 79
5.5.3. Entscheidungsfindung durch den Parser . . . 79
5.5.4. Vergleich der Lösungen . . . 81
5.6. Einbettung einer einzelnen Sprache . . . 82
5.7. Einbettung von Sprachalternativen . . . 84
5.7.1. Alternative Umsetzung durch Einbettung einer Nichtterminal-Erweiterung 87
5.7.2. Vergleich der Lösungen . . . 88
5.8. Entwicklung einer erweiterbaren Sprache . . . 88
5.9. Verwandte Arbeiten . . . 92
5.10. Zusammenfassung . . . 94

6. Kompositionale Entwicklung abstrakter Syntax 95
6.1. Kompositionalitätsmechanismen und abstrakte Syntax . . . 95
6.2. Visitoren . . . 98
6.2.1. Kombination von Visitoren . . . 100
6.2.2. Informationsaustausch bei Einbettung . . . 100
6.2.3. Erweiterung bei Vererbung . . . 102
6.3. Sprachvereinigung . . . 105
6.4. Nichtterminal-Erweiterung . . . 106
6.4.1. Umsetzung durch Mehrfachvererbung . . . 106
6.4.2. Umsetzung durch Einbettung . . . 108
6.4.3. Verwendung der Visitoren . . . 108
6.5. Nichtterminal-Ersetzung . . . 109
6.6. Einbettung einer einzelnen Sprache . . . 111
6.7. Einbettung von Sprachalternativen . . . 113
6.8. Entwicklung einer erweiterbaren Sprache . . . 113
6.9. Verwandte Arbeiten . . . 114
6.10. Zusammenfassung . . . 115

7. Kompositionale Entwicklung von Symboltabellen 117
7.1. Einführung . . . 117
7.1.1. Entries . . . 118
7.1.2. Scopes und Namespaces . . . 118
7.1.3. Arten von Symboltabellen . . . 120
7.2. Theoretische Grundlagen . . . 122
7.2.1. Attributierte Grammatiken . . . 126
7.2.2. Assoziierte Namespaces . . . 127
7.2.3. Eintragungen in Symboltabellen . . . 128

7.2.4. Namespace-Grammatiken . . . 128
7.2.5. Auflösung und Hiding . . . 130
7.2.6. Sprachkombinationen . . . 131
7.3. Struktur und Arbeitsweise . . . 132
7.3.1. Zeit- und Speichermanagement . . . 132
7.3.2. Strukturen . . . 133
7.3.3. Symboltabellenaufbau . . . 136
7.4. Sprachkombinationen . . . 143
7.4.1. Adaption der Symboltabelleneinträge . . . 143
7.4.2. Einbringen der Adaptoren . . . 144
7.4.3. Einbringen zusätzlicher Auflösungskomponenten . . . 145
7.5. Umsetzung der Szenarien . . . 146
7.5.1. Sprachvereinigung . . . 146
7.5.2. Nichtterminal-Erweiterung . . . 147
7.5.3. Nichtterminal-Ersetzung . . . 147
7.5.4. Einbettung einer einzelnen Sprache und Einbettung von Sprachalternativen 148
7.5.5. Entwicklung einer erweiterbaren Sprache . . . 148
7.6. Verwandte Arbeiten . . . 150
7.7. Zusammenfassung . . . 151

8. Kompositionale Entwicklung von Kontextbedingungen 153
8.1. Prüfung von Kontextbedingungen . . . 153
8.2. Anforderungen an Kontextbedingungsprüfungen . . . 154
8.3. Architektur . . . 158
8.4. Einsatz des MontiCore-Attributgrammatiksystems . . . 161
8.4.1. Das MontiCore-Attributgrammatik-System . . . 162
8.4.2. Informationsaustausch . . . 164
8.4.3. Zugriff auf zentrale Strukturen . . . 167
8.5. Umsetzung der Szenarien . . . 167
8.5.1. Sprachvereinigung . . . 167
8.5.2. Nichtterminal-Erweiterung . . . 168
8.5.3. Nichtterminal-Ersetzung . . . 169
8.5.4. Einbettung einer einzelnen Sprache und Einbettung von Sprachalternativen 170
8.5.5. Entwicklung einer erweiterbaren Sprache . . . 170
8.6. Verwandte Arbeiten . . . 171
8.7. Zusammenfassung . . . 172

III. Weiterführende Konzepte und Anwendungen 175

9. Kompositionale Entwicklung von Werkzeugen 177
9.1. Eclipse und sprachspezifische Plugins . . . 177
9.1.1. Aufbau einer Workbench . . . 178
9.1.2. Innere Architektur . . . 179

9.1.3. Bereitgestellte Funktionalitäten . . . 179
9.2. Erstellung von Werkzeugen . . . 181
9.2.1. Entwicklung eines sprachspezifischen Werkzeugs . . . 182
9.2.2. Struktur eines Plugins . . . 184
9.2.3. Umsetzung der Sprachaggregation und Einordnung in die Sprachhierarchie 186
9.3. Umsetzung der Szenarien . . . 188
9.3.1. Sprachvereinigung . . . 188
9.3.2. Nichtterminal-Erweiterung und Nichtterminal-Ersetzung . . . 189
9.3.3. Einbettung einer einzelnen Sprache und Einbettung von Sprachalternativen 189
9.3.4. Entwicklung einer erweiterbaren Sprache . . . 190
9.4. Verwandte Arbeiten . . . 191
9.5. Zusammenfassung . . . 192

10. Fallstudie **193**
10.1. Beteiligte Sprachen . . . 193
10.1.1. Statecharts . . . 193
10.1.2. OCL . . . 194
10.1.3. Java . . . 195
10.2. Sprachkombinationen . . . 195
10.2.1. SC/JO . . . 195
10.2.2. Java/O . . . 198
10.3. Realisierung . . . 200
10.3.1. Komponentenanordnung . . . 200
10.3.2. Verbund der Grammatiken . . . 201
10.3.3. Symboltabellen, Kontextbedingungen und Editoren . . . 202
10.4. Wichtigste Ergebnisse und Bewertung . . . 205
10.5. Zusammenfassung . . . 207

IV. Epilog **209**

11. Zusammenfassung und Ausblick **211**
11.1. Zusammenfassung . . . 211
11.2. Ausblick . . . 213

Literaturverzeichnis **215**

V. Anhänge **225**

A. Glossar **227**

B. Zeichenerklärungen **235**

C. Abkürzungsverzeichnis **237**

D. Essenzen der gemeinsamen Obergrammatiken der Sprachen der Fallstudien **239**
D.1. Literals 239
D.2. Types 242
D.3. Commons 245

E. Essenz der Statechart-Grammatik **247**

F. Essenz der OCL-Grammatik **251**

G. Essenz der Java-Grammatik **261**

H. Lebenslauf **277**

Teil I.

Grundlagen

Kapitel 1.

Einführung

Die Nutzung domänenspezifischer Sprachen (engl. Domain-Specific Languages - DSLs) bei der Erstellung komplexer Software erlaubt eine effiziente Entwicklung, bessere Wiederverwendung und verringert sich wiederholende und daher fehlerträchtige Entwicklungsarbeit. Zusätzlich sind Domänenexperten anhand einer DSL in der Lage, selbst Modelle zu erstellen und daraus gegebenenfalls Code zu generieren. Das Wissen des Domänenexperten geht somit direkt in den Entwicklungsprozess ein (z.B. [Spi01]).

Die Verwendung von DSLs und die damit entstehenden Vorteile sind jedoch nicht kostenlos. Da die Sprachen auf eine bestimmte Domäne mit all ihren Eigenheiten abzielen, müssen sie oftmals von Grund auf neu entwickelt werden. Die Neuentwicklung beinhaltet dabei eine Vielzahl teilweise sehr aufwändiger Aktivitäten [Kle07a]. Diese umfassen unter anderem die Entwicklung der konkreten Syntax, der abstrakten Syntax, der Symboltabellen und Kontextbedingungen sowie die Erstellung von sprachspezifischen Werkzeugen und Codegeneratoren.

Erschwerend kommt hierbei eine Kerneigenschaft der DSLs hinzu: die Sprachmittel einer Sprache konzentrieren sich oftmals auf die Darstellung eines bestimmten Aspektes der Domäne und abstrahieren somit von der Gesamtkomplexität. Zur Spezifikation aller Domänenaspekte werden somit Instanzen verschiedener Sprachen benötigt [VWH09]. Die konkrete Zusammensetzung der Sprachen variiert jedoch je nach Projektkontext, Zielsetzung oder Nutzerpräferenzen. Diese nötige Flexibilität erschwert die Erstellung neuer Sprachen zusätzlich.

Das Hauptziel dieser Arbeit ist es daher, sowohl den Aufwand bei der Erstellung neuer Sprachen zu reduzieren, als auch die flexible Kombination existierender Sprachen zu ermöglichen. Diese Kombination soll sich dabei nicht nur auf einen Teilaspekt der Sprachentwicklung beschränken, sondern sich vielmehr kohärent auf die Syntaxen, die Symboltabellen, die Kontextbedingungen und die sprachspezifischen Werkzeuge auswirken.

1.1. Motivation der Arbeit

Ein schon seit langer Zeit in der Informatik angewandter Ansatz zur Erstellung komplexer Systeme ist die *komponentenbasierte Softwareentwicklung* [Szy02]. Hierbei werden die Systeme aus vielen Einzelkomponenten zusammengesetzt, wobei bei der Integration lediglich ein relativ geringer Anteil zwischen den Einzelkomponenten vermittelnder Artefakte erstellt wird. Die Einzelkomponenten, die wiederum aus untergeordneten Komponenten zusammengesetzt sein können, werden dabei unabhängig von der späteren Zusammensetzung entwickelt, qualitätsgesichert und ausgeliefert. Durch diese Vorgehensweise entstehen verschiedene Vorteile für den

Entwickler: so wird unter anderem die Wiederverwendung existierender Artefakte ermöglicht, die Erweiterbarkeit der Systeme unterstützt, das Testen erleichtert und insgesamt die Agilität erhöht[1].

Bei der Entwicklung von Sprachen hat sich ein solcher Ansatz jedoch noch nicht bzw. nur in bestimmten Aspekten durchgesetzt. Dies hat unterschiedliche Gründe:

- Insbesondere DSLs werden oftmals nicht von Sprachentwicklungsexperten sondern von Entwicklern, die wenig Erfahrung in der Erstellung von Sprachen haben, erstellt. Hierdurch kommt es zu Lösungen, die ad-hoc entstehen und oftmals wenig strukturiert sind. Die Einzelsprachen haben daher eine sehr unterschiedliche Implementierungsweise, wodurch eine Kombination verschiedener Sprachen erschwert wird. Eine vorgegebene und standardisierte Entwicklungsweise hingegen würde die Kompositionalität im gleichen Maße wie bei der komponentenbasierten Softwareentwicklung fördern [Aßm03].

- Die Entwicklung einer Sprache ist leider umfangreich [Kle07a]. Die verschiedenen Anteile werden oftmals mit speziellen Werkzeugen wie z.B. Parsergeneratoren für die konkrete Syntax, Constraint Languages wie OCL für die Kontextbedingungen oder Eclipse als Grundlage für Editoren erstellt. Diese Werkzeuge bieten dabei unterschiedliche Mechanismen an, die es bei einem kompositionalen Ansatz zu vereinheitlichen gilt.

- Ein dritter Grund für die fehlende Komponentenbasierung bei der Sprachentwicklung ist derselbe wie bei der komponentenbasierten Softwareentwicklung. Die Entwickler der Einzelkomponenten/Einzelsprachen sind oftmals nicht diejenigen, welche die Komponenten bei Bildung eines größeren Systems wiederverwenden oder aber die Entwickler sehen im Voraus für sich selbst keine mögliche Wiederverwendung. Unter diesen Aspekten und da insbesondere noch kein durchgängiger Ansatz für die kompositionale Sprachentwicklung existiert, wird der Aufwand bei deren Implementierung unter Verwendung komponentenorientierter Richtlinien für den Einzelentwickler oftmals als zu hoch angesehen. Eine Unterstützung der Kompositionalität aus werkzeugtechnischer Sicht würde den Aufwand und somit die Hürde senken [Hud98].

- Ein weiterer Grund für die fehlende Übertragung der Komponentenbasierung auf die Sprachentwicklung ist der sich erst in den letzten Jahren abzeichnende neuartige Umgang mit DSLs. Zwar sind spezielle Sprachen für spezifische Aufgaben schon immer in der Informatik entwickelt und angewandt worden, der massive Einsatz multipler DSLs hat sich jedoch erst mit der vermehrten Verwendung von Modellierungstechniken etabliert. Gerade diese Modellierungstechniken benötigen verschiedene Sprachen auf unterschiedlichen Abstraktionsebenen, die je nach Projektsituation frei wählbar und kombinierbar sein müssen [VWH09].

Insbesondere der letzte Punkt - die freie Wählbarkeit und Kombinierbarkeit von Sprachen - ist das Ziel unterschiedlicher wissenschaftlicher Arbeiten (z.B. [BHD+01, Cle07, Ada91, EH07]. In diesen Arbeiten spiegelt sich jedoch auch die Komplexität der Sprachentwicklung wider: es

[1] Für eine detaillierte Diskussion wird auf Abschnitt 2.3 verwiesen.

werden stets nur bestimmte Teilaspekte (z.B. konkrete oder abstrakte Syntax) betrachtet, nur einige wenige beziehen mehr als einen Aspekt mit in die Überlegungen ein. Ein Ansatz, der die gesamte Palette der benötigten Elemente auf kompositionaler Basis betrachtet, existiert jedoch nicht.

Zweifellos erhöht die Einbeziehung aller Teilaspekte den Schwierigkeitsfaktor erheblich. Für jeden Aspekt existieren eigene Realisierungsmethoden mit unterschiedlichen Randbedingungen. Ein integrativer und kohärenter Ansatz muss daher die Randbedingungen aller Teilaspekte mit in die Überlegungen einbeziehen. Demgegenüber ist die Einbeziehung aller Teilaspekte jedoch auch zwingend notwendig: eine Kompositionalität auf nur einer Ebene - beispielsweise der konkreten Syntax - bringt nur geringe Vorteile, denn bei fehlenden Mechanismen auf den anderen Ebenen müssen diese nach einer Komposition neu entwickelt werden. Gleiches gilt, wenn Kompositionalitätsmechanismen auf verschiedenen Ebenen existieren, diese sich jedoch nicht entsprechen oder die unterschiedlichen Randbedingungen eine gleichzeitige Anwendung nicht erlauben.

Die zentrale Fragestellung sollte daher nicht auf der Realisierung oder der Verbesserung der Kompositionalität einzelner Teilaspekte liegen, sondern vielmehr in der Integration. Denn erst hierdurch können komplette Sprachen als Komponenten angesehen und mit all ihren Teilen separat entwickelt, getestet und ausgeliefert werden. Bei der konkreten Zusammensetzung können diese dann bibliotheksartig eingebunden und somit wiederverwendet werden. Wie bereits bei der komponentenbasierten Softwareentwicklung sollte der Aufwand zur Kombination dabei vergleichsweise klein gegenüber dem Aufwand bei der Entwicklung der Einzelsprachen sein.

Insgesamt lässt sich feststellen, dass die Komponentenbasierung in der Softwareentwicklung nicht zuletzt aufgrund der dadurch entstehenden Vorteile bereits seit langem breite Anwendung findet. Auf der anderen Seite existiert ein durchgehender Ansatz für die Domäne der Sprachentwicklung nicht. Insbesondere der flexible Einsatz multipler Sprachen bei der Spezifikation komplexer Systeme verlangt jedoch nach Methodiken und Werkzeugen, die eine Wiederverwendung, eine Evolution und Adaption sowie eine Erweiterung existierender qualitätsgesicherter Sprachen erlauben. Die konsequente Wiederverwendung steigert dabei nicht nur die Qualität der Sprachen und die Flexibilität und Agilität in Bezug auf den Umgang mit ihnen, sondern wirkt sich auch positiv auf die anhand der Sprachen entwickelten Zielsysteme aus.

1.2. Wichtigste Ziele und Ergebnisse

Die wichtigsten Ziele und Ergebnisse dieser Arbeit sind:

1. Übertragung der Prinzipien komponentenbasierter Softwareentwicklung auf die Sprachentwicklung. Hierbei werden die einzelnen Sprachen separat und ohne Antizipation einer konkreten Kombination entwickelt und können anschließend bibliotheksartig mit geringem Aufwand mit anderen Sprachen kombiniert werden.

2. Analyse verschiedener existierender Sprachkombinationen und Erstellung einer Einteilung dieser Kombinationen in Szenarien.

3. Erweiterung der in MontiCore [KRV07b, KRV08b, KRV10, Kra10] entwickelten Möglichkeiten zur Kompositionalität konkreter und abstrakter Syntax. Hierzu erfolgt eine Untersuchung, wie sich die identifizierten Szenarien durch existierende Mechanismen realisieren lassen. Zusätzlich erfolgen Anpassungen oder Erweiterungen aufgrund der Untersuchungsergebnisse.

4. Erstellung eines Frameworks als Basis zur Entwicklung von Symboltabellen und Kontextbedingungen für Sprachen sowie einer Generierung sprachspezifischer Werkzeuge in Form von Eclipse-Plugins.

5. Unterstützung der Szenarien auch auf den Ebenen der Symboltabellen, Kontextbedingungen und der sprachspezifischen Werkzeuge.

6. Kohärente Komposition: insgesamt wird für jedes identifizierte Szenario ein durchgängiger, nahtloser Kompositionsmechanismus auf den Ebenen der konkreten Syntax, der abstrakten Syntax, der Symboltabellen, der Kontextbedingungen und der sprachspezifischen Werkzeuge geschaffen. Erst durch diese Kohärenz ist eine kompositionale Verwendung von Sprachen möglich.

1.3. Nahestehende Arbeiten

An dieser Stelle werden einige Arbeiten vorgestellt, die in engem Verhältnis zur vorliegenden Arbeit stehen, als Grundlage genutzt wurden oder aus denen Sprachimplementierungen als Forschungsobjekte genutzt wurden. Da derzeit kein Ansatz bekannt ist, der die Kompositionalität von Sprachen kohärent auf allen Ebenen unterstützt, werden die verwandten Arbeiten, die jeweils nur einen bestimmten Aspekt betrachten, in den jeweiligen Teilkapiteln vorgestellt.

Zunächst fußt diese Arbeit auf Vorarbeiten vor allem bezüglich des Werkzeuges MontiCore. Hier wurden in der Dissertation „MontiCore: Agile Entwicklung von domänenspezifischen Sprachen“ [Kra10] wesentliche Grundlagen gelegt, die in dieser Arbeit systematisch wiederverwendet und erweitert worden sind. Zu diesen Grundlagen gehört das MontiCore-Grammatikformat zur integrierten Spezifikation von konkreter und abstrakter Syntax, das Framework zur Verarbeitung von Modellen und erste Modularitätsmechanismen Vererbung und Einbettung, wobei insbesondere die Einbettung in Kooperation entwickelt wurde.

Daneben wurden während der Erstellung dieser Arbeit eine Vielzahl von am Institut für Software Systems Engineering der TU Braunschweig bzw. des Lehrstuhls Software Engineering der RWTH Aachen entwickelten Sprachen als begleitende Forschungsobjekte oder als explizite Fallstudien genutzt. Diese sind in verschiedenen Projekten des Lehrstuhls bzw. Instituts entstanden, wurden teilweise in Studien-, Diplom-, Bachelor- oder Masterarbeiten entwickelt oder sind Bestandteile von Dissertationen. Die betreffenden Sprachen sind dabei unter anderem

- eine Sprache zur Darstellung endlicher Automaten [Rum96],
- Message Sequence Charts [GMP03, KRV07a],
- das MontiCore-Grammatikformat [KRV07b, Kra10],

- Java 1.5 [GJS05, FM07],
- C++ [Str00, Wit07],
- die Architekturbeschreibungssprache MontiArc zur Spezifikation interaktiver, verteilter Systeme [HKRR11],
- eine Sprache zur Spezifikation von Constraints für Gebäude- und Energiedaten [NMFR10],
- eine Sprache zur Spezifikation und Generierung von Websystemen incl. Aktivitätsdiagrammen [DRRS09],
- die OCL [OMG10a, PS07, Hel10] und
- die UML/P [Sch11, Rum04b, Rum04a] mit den Sprachen Klassendiagramme, Objektdiagramme, Sequenzdiagramme und Statecharts.

1.4. Wichtigste Notationselemente

Im Verlauf dieser Arbeit werden unterschiedliche Diagramme und Codefragmente zur Darstellung technischer Sachverhalte genutzt. Diese sind bei Bedarf mit Markern versehen, die die Bedeutung der jeweiligen Elemente festlegen. Eine vollständige Auflistung dieser Notationselemente findet sich in Anhang B.

Viele dieser Elemente sind oft selbsterklärend und benötigen keine näheren Erläuterungen. Einige wenige sind jedoch von zentraler Bedeutung und werden deshalb bereits in diesem ersten Kapitel vorgestellt. Diese Notationselemente finden dabei in unterschiedlichen Diagramm- und Codearten Verwendung und deuten auf die Rolle der jeweiligen Artefakte während der kompositionalen Sprachentwicklung auf MontiCore-Basis hin. Insgesamt wird dabei zwischen den in Tabelle 1.1 dargestellten drei Rollen unterschieden.

Zusätzlich wird in vielen Diagrammen und Codefragmenten unterschieden, ob das dargestellte Artefakt vom Entwickler handgeschrieben («`hw`» - handwritten) ist oder generiert («`gen`») wurde. Da generierter Code generell nicht modifiziert wird, ist eine eindeutige Unterscheidung immer möglich.

1.5. Aufbau dieser Arbeit

Diese Arbeit gliedert sich wie folgt:

Teil I - Grundlagen

Kapitel 1 beinhaltet eine Einführung in diese Arbeit, nennt wichtigste Ziele und Ergebnisse und stellt nahestehende Arbeiten vor.

Markierung	Beispiele für markierte Elemente	Bedeutung
«mc»	Klassen in Klassendiagrammen	Basisklassen des MontiCore-Frameworks. Diese sind einmalig vorgegeben und werden für die Erstellung einer neuen Sprache lediglich wiederverwendet.
«component»	Klassen in Klassendiagrammen, Grammatiken, Sprachdateien	Elemente, die für die Entwicklung einer neuen Sprache erstellt bzw. aus einer einzelnen Sprachdefinition generiert wurden. Diese Elemente sind unabhängig von einer späteren Sprachkombination.
«glue»	Klassen in Klassendiagrammen, Grammatiken, Sprachdateien	Elemente, die für die Verbindung verschiedener Sprachen entwickelt bzw. zum Zwecke der Sprachkombination generiert wurden. Diese sind sowohl abhängig von den konkreten Sprachen als auch von der Sprachkombination.

Tabelle 1.1.: Notationselemente zur Unterscheidung der Rollen der Artefakte

Kapitel 2 charakterisiert die komponentenbasierte Softwareentwicklung und diskutiert deren Vorteile gegenüber herkömmlichen Entwicklungsmethodiken.

Kapitel 3 führt in die Sprachentwicklung und in das MontiCore-Framework ein.

Teil II - Kompositionale Sprachentwicklung

Kapitel 4 bespricht die Anwendungen und Techniken kompositionaler Sprachentwicklung und stellt unter anderem die verschiedenen Nutzungsszenarien vor.

Kapitel 5 diskutiert die kompositionale Entwicklung konkreter Syntax und beschreibt hierfür die Umsetzung der Szenarien.

Kapitel 6 stellt die Umsetzung der Szenarien bezüglich der abstrakten Syntax einer Sprache vor.

Kapitel 7 beinhaltet eine theoretische Betrachtung kompositionaler Symboltabellen und diskutiert deren werkzeugtechnische Umsetzung.

Kapitel 8 erklärt die Thematik der Kontextbedingungsprüfungen und die darauf aufbauende Realisierung der Szenarien.

Teil III - Weiterführende Konzepte und Anwendungen

Kapitel 9 zeigt die Möglichkeiten zur Generierung kompositionaler sprachspezifischer Werkzeuge auf.

Kapitel 10 beschreibt die Fallstudie, in der die in dieser Arbeit entwickelten Mechanismen angewandt wurden.

Teil IV - Epilog

Kapitel 11 fasst die Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Erweiterungen.

Teil V - Anhänge

Anhang A - Glossar erläutert die wichtigsten Fachbegriffe dieser Arbeit.
Anhang B - Zeichenerklärungen erklärt die verwendeten Abbildungen.
Anhang C - Abkürzungsverzeichnis listet die verwendeten Abkürzungen auf.
Anhang D - Essenzen der gemeinsamen Obergrammatiken der Sprachen der Fallstudien stellt die grundlegenden Grammatiken der Fallstudie vor.
Anhang E - Essenz der Statechart-Grammatik beinhaltet die in der Fallstudie verwendete Grammatik der Statecharts.
Anhang F - Essenz der OCL-Grammatik enthält die in der Fallstudie eingesetzte Grammatik der OCL.
Anhang G - Essenz der Java-Grammatik stellt die in der Fallstudie verwendete Java-1.5-Grammatik vor.
Anhang H - Lebenslauf enthält einen Lebenslauf.

Kapitel 2.

Methodik der komponentenbasierten Softwareentwicklung

Die zunehmende Komplexität von Softwaresystemen und die damit verbundenen Auswirkungen verlangen nach spezialisierten Entwicklungsmethodiken. Die grundlegende Problemstellung ist dabei keineswegs erst in den vergangenen Jahren forschungsrelevant, vielmehr wurden bereits in der Anfangszeit der Informatik bzw. des Software Engineering Probleme und Lösungsansätze diskutiert. So erläutert bereits McIlroy [McI68], dass die aus anderen Ingenieursdisziplinen bekannte Vorgehensweise der Wiederverwendung von Komponenten als austauschbare Blackboxes mit wohldefinierten Schnittstellen ein wesentliches Kernkonzept zur Beherrschung von Komplexität ist. Auch Parnas beschreibt bereits 1972 [Par72], wie Systeme sinnvoll in Einheiten geteilt werden können, deren Komposition anschließend das Gesamtsystem bildet.

Seit diesen Anfängen ist die Idee der komponentenbasierten Entwicklung immer weiter vorangetrieben worden [Aßm03]. In den Frühstadien waren vor allem einzelne Funktionen und Module Ziele der Wiederverwendung, während die heutige objektorientierte Entwicklung es erlaubt, Objekte bzw. Klassen als eigenständige Komponenten wiederzuverwenden. Ansätze wie CORBA [RNS04] ermöglichen es, auf noch höherem Abstraktionsgrad sehr große Standardkomponenten, die möglicherweise in unterschiedlichen Sprachen geschrieben sind, in einem Gesamtsystem zu integrieren.

Insgesamt lassen sich bei diesen Ansätzen wesentliche Gemeinsamkeiten finden, die die kompositionale Softwareentwicklung bzw. kompositionale Softwaresysteme charakterisieren. Diese Eigenschaften werden im ersten Teilabschnitt dieses Kapitels vorgestellt.

Der zweite Teilabschnitt diskutiert anschließend die Anforderungen an die verschiedenen Bestandteile eines Kompositionssystems und der dritte Abschnitt stellt die Vorteile kompositionaler gegenüber herkömmlicher Softwareentwicklung vor.

2.1. Charakterisierung kompositionaler Softwaresysteme

Die zentrale Idee der komponentenbasierten Entwicklung ist die Erstellung eines Gesamtsystems durch die Komponierung existierender Komponenten. Diese Vorgehensweise steht im starken Gegensatz zur traditionellen Softwareentwicklung, bei der das Gesamtsystem meist von Grund auf neu entwickelt wird. Abbildung 2.1 illustriert die Vorgehensweise komponentenbasierter Entwicklung.

Hierbei ist die hierarchische Struktur komponentenbasierter Systeme zu erkennen: auf jeder Stufe existieren Komponenten, die sich wiederum aus Subkomponenten zusammensetzen.

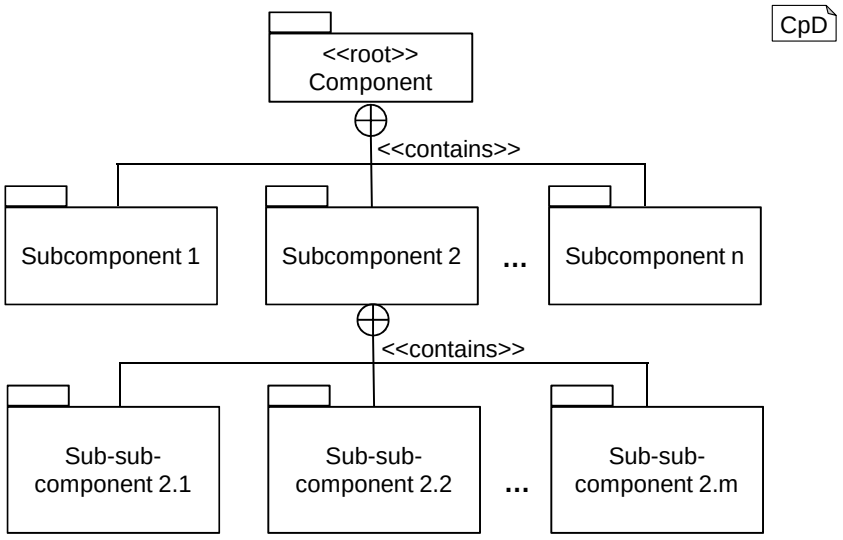


Abbildung 2.1.: Aufbau eines komponentenbasierten Softwaresystems [Gro05]

Selbst die oberste Hierarchieebene - das eigentliche Softwaresystem - wird dabei als Komponente gesehen, da dieses wiederum als Komponente eines anderen Systems eingesetzt werden kann. Explizit zu beachten ist die «contains»-Beziehung. Diese besagt, dass eine übergeordnete Komponente die untergeordneten beinhaltet, nicht aber, dass sie sich nur aus diesen zusammensetzt. Wie in späteren Ausführungen dargelegt wird, enthalten die übergeordneten Komponenten zusätzlich *Glue*, der zwischen den einzelnen Teilen vermittelt.

Die Auswirkungen eines komponentenbasierten Ansatzes auf den Entwicklungsprozess sind dabei vielfältig (siehe z.B. [CCL06, Szy02, HC01]). Herauszuheben ist die Separierung von Komponenten- und Systementwicklung. Da Komponenten eigenständige Artefakte bilden und erst später zu kompletten Applikationen zusammengefasst werden, sind die Komponenten selbst Ziel der Entwicklung, der Qualitätssicherung und Wartung. Die Systementwicklung hingegen fokussiert sich vielmehr auf das Finden und Selektieren adäquater Komponenten und die Entwicklung des Glues.

Auch die Entwicklung von Komponenten an sich unterscheidet sich von der Entwicklung herkömmlicher Softwaresysteme, da der Fokus bei der Komponentenentwicklung viel stärker auf Wiederverwendbarkeit, Generizität und Abstraktion liegt. Wie später detaillierter beschrieben wird, sollen die einzelnen Komponenten in verschiedenen Kontexten eingesetzt werden können, ihr äußeres Verhalten in Form von Schnittstellen beschreiben und gleichzeitig von der inneren Funktionsweise abstrahieren.

Ein weiterer Aspekt ist die Bildung von Bibliotheken adäquater Komponenten, ohne dass bereits ein Gesamtsystem vorhersehbar ist. Hierbei wird eine Sammlung von Komponenten entwickelt und System-Entwicklern als Basis zur Verfügung gestellt, die dann ihr System aus Teilen der Bibliothek zusammensetzen. Diese eigenständige Entwicklung von Komponenten unterliegt anderen Voraussetzungen als die Entwicklung kompletter Systeme. Beispielhaft sei hierbei die Aktivität des Testens aufgeführt: das Testen von Komponenten muss anders als das Testen

vollständiger Applikationen gestaltet werden, da die Komponenten Bedingungen an die externe Umgebung stellen, die vom Testsystem zur Verfügung gestellt werden muss. Außerdem sind die Tests explizit Teil der Komponente, da diese in der letztendlichen Zielumgebung erneut getestet werden sollte [Gro05].

2.2. Anforderungen an eine Komposition

Sowohl bei der Entwicklung der Komponenten, als auch bei der letztendlichen Komposition der Einzelteile sind einige grundlegende Richtlinien zu beachten. Ziel dieser Richtlinien ist es dabei, den Aufwand bei der Wiederverwendung zu minimieren und somit die Entwicklungskosten zu senken. Generell können die Richtlinien im Sinne von Anforderungen, die bei der Verwendung eines komponentenbasierten Ansatzes zu beachten sind, verstanden werden. Diese Anforderungen lassen sich dabei anhand der Bestandteile einer Komposition anordnen.

Im Folgenden werden deshalb zunächst die Bestandteile einer Komposition vorgestellt, um in den weiteren Teilabschnitten Anforderungen an diese Bestandteile zu diskutieren.

2.2.1. Bestandteile einer Komposition

Zur Komposition sind 3 wesentliche Bestandteile notwendig: das *Komponentenmodell*, die *Kompositionstechnik* und die *Kompositionssprache* [Aßm03]. Das *Komponentenmodell* beschreibt, um welche Art von Artefakt (z.B. Software in Form von Code, Modelle) es sich handelt und bestimmt damit die Austauschbarkeit von Artefakten. Weiterhin definiert es die nach außen sichtbare Schnittstelle der Artefakte und damit auch, welche Informationen versteckt werden und somit für eine Komposition irrelevant sind. Dies können z.B. Implementierungsdetails oder gar die verwendete Sprache sein. Die *Kompositionstechnik* beschreibt, wie die Artefakte komponiert werden. Sie beinhaltet so zum Beispiel die Frage des *Bindens* (statisch oder dynamisch) oder kann die Aufgabe des Generierens von Glue Code übernehmen. Die *Kompositionssprache* dient dazu, die Komposition an spezielle Bedürfnisse anzupassen. Mit ihr kann unter anderem beschrieben werden, wie die Artefakte kombiniert werden sollen. Diese Sprache muss dabei nicht zwingend eine eigenständige DSL sein, sondern kann auch in Form einer Programmiersprache mit einer geeigneten API realisiert werden.

2.2.2. Anforderungen an das Komponentenmodell

Logische Einheit

Eine wesentliche Eigenschaft von Softwarekomponenten besteht darin, dass sie eine *logische Einheit* für einen *klar umrissenen Aufgabenbereich* bilden (z.B. [Boo87]). Die Einteilung des Systems in Komponenten ist dabei nicht eindeutig: Systeme können auf relativ natürliche Weise anhand einzelner Funktionen zerlegt werden (z.B. Eingabe, Verarbeitung, Ausgabe). Diese Zerlegung ist jedoch nicht immer vorteilhaft, da auf diese Weise Anforderungsänderungen leicht Auswirkungen auf mehrere Komponenten haben können. Deshalb wird als Alternative in [Par72] vorgeschlagen, dass Komponenten einzelne Design-Entscheidungen, die sich unter Umständen ändern können, reflektieren.

Schnittstellen

Die komponentenbasierte Entwicklung zielt vor allem darauf ab, Wiederverwendung zu erhöhen und Komplexität zu vermindern. Daher ist es unerlässlich, dass einzelne Komponenten über klar definierte Schnittstellen verfügen, die Aussagen darüber treffen, welche Funktionalitäten angeboten und benötigt werden. Diese Schnittstellen sollten in jedem Fall formal definiert sein, damit eine automatisierte Überprüfung stattfinden kann. Die Verwendung von Schnittstellen ermöglicht es, Komponenten unabhängig von ihrer Implementierung zu verwenden und somit eventuell gegen andere auszutauschen. Hier ist jedoch anzumerken, dass die Schnittstellen genug Informationen enthalten müssen, um klar abzugrenzen, wann eine Komponente gegen eine andere austauschbar ist. Dies geht im Allgemeinen über die reine syntaktische Konformität hinaus, vielmehr spielt die Semantik eine tragende Rolle.

Die Einführung von Schnittstellen ist weiterhin eng verwandt mit dem Konzept des *information hiding* [Par72]: Gegenüber einem Nutzer einer Komponente werden gezielt Implementierungsdetails versteckt und damit die Verständlichkeit erhöht, da der Nutzer sich nur auf die Angaben der Schnittstelle konzentrieren muss. Zusätzlich wird durch die Verwendung von Schnittstellen ein höheres Maß an Abstraktion erreicht [Szy02]: Nutzer müssen nichts über die innere Funktionsweise der Komponenten wissen, da die Schnittstellen die Komponente hinreichend genau beschreiben, um sie in verschiedenen Kontexten einsetzen zu können.

Prinzipiell stellt sich die Frage, wie viel Information hinter einer Schnittstelle, d.h. in der eigentlichen Implementierung sichtbar ist. Hier wird von Blackbox bis hin zu Whitebox-Ansätzen unterschieden. Erstere verbergen weitere Informationen komplett, während letztere die Implementierung mitliefern, um ein tieferes Verständnis der Komponente zu ermöglichen. Dies mag zwar auf den ersten Blick sinnvoll sein, birgt jedoch die Gefahr, dass Nutzer sich auf Eigenschaften der Implementierung verlassen, die nicht durch die Schnittstelle garantiert werden.

Unabhängigkeit

Ein wichtiges Kriterium für Softwarekomponenten ist die unabhängige Entwicklung und damit verbunden die separate Auslieferbarkeit (z.B. [DW99]). Hierbei werden die Komponenten unabhängig vom Kontext, d.h. dem Rest des Gesamtsystems, entwickelt. Hierzu hilft wiederum der oben genannte Einsatz von Schnittstellen: diese sind einerseits der Kompilierbarkeit geschuldet, unter anderem um die separate Auslieferbarkeit zu ermöglichen. Andererseits dienen sie später dem Verständnis, welche Anforderungen eine Komponente an die Umgebung stellt und somit von anderen Komponenten geliefert werden muss.

Parametrisierung

Der Begriff der Parametrisierung ist eng verwandt mit dem der Schnittstellen. Komponenten werden hier generisch gestaltet und können so flexibel an ihre Umgebung angepasst werden. Daher sollten Komponenten klar definieren, welche Parameter sie besitzen und welche Parameterwerte in konkreten Anwendungen eingesetzt werden müssen.

Standardisierung

Eine Standardisierung spielt eine wichtige Rolle bei der komponentenbasierten Entwicklung: je weiter die verwendeten Technologien, Vorgehensweisen, das Format der Schnittstellen etc. voneinander entfernt sind, desto schwieriger wird eine Integration der Komponenten. Ein standardisierter Ansatz bei der Komponentenentwicklung kann eine wesentliche Vereinfachung und somit Effizienzsteigerung bedeuten.

2.2.3. Anforderungen an die Kompositionstechnik

Geschwindigkeit

Eine der offensichtlichsten Anforderungen an eine Kompositionstechnik ist Geschwindigkeit bei der Komposition. Durch die stetig wachsende Größe aktueller Systeme wird die Anzahl der beteiligten Komponenten immer weiter erhöht. Insbesondere während der Erstellung der Systeme ist ein schnelles Feedback an den Entwickler nötig, um Agilität einerseits und erhöhte Geschwindigkeit andererseits zu unterstützen. Daher muss bereits beim Entwurf einer Kompositionstechnik auf die Geschwindigkeit geachtet werden.

Separate Kompilierung

Wie bereits in [HKR+08] diskutiert, besteht eine grundlegende Möglichkeit zur Erhöhung der Geschwindigkeit und somit für effizientes Entwickeln in der separaten Kompilierung. Wie schon eingangs dieses Teilkapitels erläutert, müssen Komponenten eigenständige Einheiten sein, d.h. insbesondere auch unabhängig kompiliert werden können. Auch der Aspekt der bedingten Kompilierung spielt eine wesentliche Rolle: nur Dinge, die sich seit der letzten Übersetzung geändert haben, sollten erneut kompiliert werden. Anzumerken ist hier jedoch, dass es insbesondere für eine Korrektheitsprüfung nötig sein kann, dass andere Teilkomponenten zusätzlich geladen und betrachtet werden müssen. Durch Einführung der oben beschriebenen Schnittstellen kann dieses Nachladen jedoch auf ein Minimum reduziert werden.

Verzögerung

Eine wesentliche Fragestellung bei der Erstellung eines Kompositionssystems ist die *Bindezeit*, also der Zeitpunkt, an dem vormals unabhängige Komponenten miteinander verbunden werden. Einige Ansätze aus der modellbasierten Entwicklung (z.B. [PB03, KPP06a]) integrieren die einzelnen Teile zu einem sehr frühen Zeitpunkt, d.h. dass oftmals erst die Modelle komponiert werden, bevor daraus Code generiert wird. Dieser Ansatz ist zumindest bei komplexen Systemen nicht erfolgversprechend, da ein integriertes Gesamtmodell schnell die Kapazitäten der Hardware (z.B. Speicher) übersteigen kann. Des Weiteren wird die Agilität innerhalb des Projektes stark reduziert, da die Bildung eines großen Gesamtmodells und die anschließende komplette Neugenerierung sehr viel Zeit in Anspruch nehmen kann und deshalb dieser Prozess als hinderlich gesehen und vermieden wird. Da jedoch zu irgendeinem Zeitpunkt die verschiedenen Teile integriert werden müssen, kann auf eine Komposition nicht verzichtet werden. Die Lösung besteht vielmehr im Herauszögern des Bindens der Komponenten zu einem möglichst

späten Zeitpunkt. In einer modellbasierten Entwicklung sollte dies zum Beispiel erst bei der Kompilierung des aus den Eingabemodellen generierten Codes erfolgen, da die Compiler auch mit einer sehr großen Anzahl von Eingaben arbeiten können. Die Schwierigkeit bei dieser Vorgehensweise in Verbindung mit separater Kompilierung ist das eigentliche Zusammenfügen der Komponenten. Da der Code zunächst unabhängig generiert wird, müssen zusätzliche Teile existieren, die die Funktionen der einzelnen Komponenten miteinander verbinden (Glue Code). In den weiteren Abschnitten dieser Arbeit werden unterschiedliche Techniken für die Erstellung des Glue Codes verwendet, im Wesentlichen werden hierfür spezielle Softwarearchitekturen und *Design Patterns* benutzt.

2.2.4. Anforderungen an die Kompositionssprache

Adaption

Bei einer komponentenbasierten Entwicklung werden oftmals Teile von Drittparteien verwendet. Hierbei kommt es zu Situationen, in denen die Schnittstellen verschiedener Komponenten nicht kompatibel sind. Eine Änderung der Schnittstellen der Komponenten ist meist nicht oder nur mit sehr hohem Aufwand möglich. Daher muss eine Kompositionssprache Mittel zur Verfügung stellen, die verschiedenen Schnittstellen aufeinander zu übertragen. Dies hat anschließend unmittelbare Auswirkungen auf die Kompositionstechnik: hier muss die eigentliche Adaption stattfinden, was meist durch eine Erstellung von Glue in Form von geeigneten Design Patterns wie Adaptern geschieht.

Konsistenzprüfung

Ein wichtiger Schritt bei der Verwendung eines komponentenbasierten Ansatzes ist die Übertragung der Schnittstellen der verschiedenen Komponenten. Hierbei werden benötigte Funktionalitäten einer Komponente durch angebotene Funktionen einer anderen Komponente erfüllt. Die Kompositionssprache, die es erlaubt, solche Zuordnungen zu definieren, muss hierbei sicherstellen, dass eine Konsistenz zwischen den beteiligten Schnittstellen besteht. Hierzu ist einerseits zu prüfen, ob die Funktionalitäten zueinander konform sind (z.B. Signaturgleichheit), andererseits ist bei der Erstellung einer lauffähigen Software aus Komponenten darauf zu achten, dass alle geforderten Funktionalitäten aller Komponenten erfüllt sind.

Diese Konsistenzprüfung spielt vor allem bei der modellgetriebenen, generativen Softwareentwicklung eine wesentliche Rolle. Hierbei sind die Komponenten in Form von Modellen bzw. Modellsammlungen gegeben und werden zueinander in Bezug gebracht. Die Konsistenzprüfung sollte dann bereits auf der Ebene der Modelle erfolgen und nicht erst im generierten Code, da eventuelle Fehler dann auf die Ursprungsmodelle zurückzurechnen sind und Fehlermeldungen somit durch Anwender nicht oder nur erschwert verstanden werden können.

2.3. Vorteile kompositionaler Softwaresysteme

Die kompositionale Entwicklung von Softwaresystemen bietet unterschiedliche Eigenschaften, die zum einen der Komplexitätsbeherrschung, zum anderen aber auch der Agilität, Änderbarkeit

und Wartbarkeit dienen. Im Folgenden werden diese Eigenschaften und die damit verbundenen Vorteile vorgestellt.

Kompositionalität unterstützt Flexibilität. Die Anpassbarkeit an geänderte Anforderungen ist ein Kernziel der Entwicklung von Software. Durch das hohe Tempo der Neuentwicklungen von Technologien in der heutigen Zeit müssen Systeme in der Lage sein, schnell auf diese Änderungen zu reagieren. Beispiele hierfür sind die ständigen Neuerungen im Bereich der Webapplikationen: in den letzten Jahren haben sich ständig neue Basistechnologien und Frameworks entwickelt, die eine Anpassung existierender Software auf unterschiedlichen Ebenen (z.B. Präsentationsschicht, Datenschicht, Logikschicht) verlangen.

Ein kompositionaler Ansatz der Softwareentwicklung unterstützt dabei die Flexibilität in Bezug auf die Möglichkeiten, solche Änderungen einzuarbeiten. Vorhandene Komponenten können dabei durch Neue ersetzt werden, ohne dass das gesamte System geändert werden muss. Im besten Falle genügt dabei die einfache Ersetzung, in manchen Fällen ist jedoch auch eine Anpassung der Schnittstellen der Komponenten nötig [HC01].

Kompositionalität unterstützt Erweiterbarkeit. Neben der Anpassbarkeit bereits entwickelter Software durch Austausch existierender Komponenten unterstützt Kompositionalität auch die Erweiterbarkeit der Systeme. Dies gilt insbesondere bei Erweiterung einzelner Komponenten um neue Funktionalitäten. Diese Erweiterungen wirken sich unmittelbar auf alle Systeme aus, die die geänderte Komponente verwenden.

Neben der Erweiterung einzelner Komponenten können kompositional entwickelte Systeme jedoch auch durch die Einbindung neuer Komponenten erweitert werden. Anwendungsbeispiel hierfür ist die Ersetzung einer einfachen Reporting-Funktion einer Software (z.B. in Form von Reporten in Dokumentform) durch eine multiple Version (z.B. Dokumentreport, Ablage in Datenbanken, Veröffentlichung auf Webseiten etc.).

Kompositionalität unterstützt Wiederverwendung. Durch die Verwendung eines kompositionalen Entwicklungsansatzes wird die Wiederverwendung bereits entwickelter Funktionalität gestärkt. Vorgefertigte Komponenten können bibliotheksartig abgelegt und in neuen Projekten wiederverwendet werden [KRV08a].

Diese Vorgehensweise hat nicht nur Auswirkungen auf die Entwicklungszeit, sondern auch auf die Qualität neuer Systeme. Wenn die vorgefertigten Komponenten qualitätsgesichert (z.B. durch Tests, Reviews etc.) sind, hat das unmittelbar Auswirkung auf die Qualität des Gesamtsystems. Nichtsdestotrotz kann auf Qualitätssicherungsmaßnahmen auf Systemebene (z.B. durch Integrationstests) nicht verzichtet werden. Der Gesamtaufwand gestaltet sich jedoch meist geringer [Szy02].

Kompositionalität unterstützt Testbarkeit. Die Testbarkeit ist ein zentrales Problem bei der Entwicklung komplexer Softwaresysteme. Vorgehensmodelle wie das V-Modell [RBB06] sehen Tests auf unterschiedlichen Granularitätsebenen vor: von Unit- über Integrations- bis hin zu Systemtests.

Die kompositionale Entwicklung unterstützt diese Granularitätsebenen, da im Gegensatz zu monolithischen Anwendungen sowohl einzelne Komponenten separat, als auch die Integration mehrerer Komponenten bis hin zum Gesamtsystem getestet werden können. Wie bereits erläutert wurde, sind Tests explizit Teil der Komponente, da die Komponenten in ihrer letztendlichen Umgebung getestet werden sollten. Daher sind die Tests auf Unit-Ebene bereits durch die Tests der Komponenten vorgegeben und müssen nicht in jedem System neu erstellt werden. Auch die Verwendung von Mock-Objekten als Platzhalter für andere Komponenten wird vereinfacht.

Kompositionalität unterstützt explizite Abhängigkeiten. Explizite Abhängigkeiten zwischen Teilen eines Softwaresystems in Form von Schnittstellen bieten verschiedene Vorteile. Zum einen besteht durch die Verwendung von Schnittstellen eine Änderungssicherheit, da Modifikationen, welche das Interface nicht ändern, keine Auswirkungen auf das System haben[1].

Zum anderen bieten Schnittstellen die Vorteile loser Koppelung und unterstützen das Konzept des Information Hiding [Par72]. Die lose Koppelung unterstützt dabei unter anderem die Austauschbarkeit von Komponenten, da die einzelnen Teile nicht direkt miteinander verbunden sind, sondern lediglich die Schnittstellen bekannt sind. Damit können die Komponenten durch andere ersetzt werden, welche die gleiche Schnittstelle implementieren. Auch die Wiederverwendung wird unterstützt, da Komponenten verschiedene Schnittstellen implementieren bzw. über Adaptoren eingebunden werden können.

Kompositionalität unterstützt Verständlichkeit. Wie im Kapitel 4.5.1 näher untersucht werden wird, ermöglicht die kompositionale Entwicklung die Analyse des Gesamtsystems aufgrund der Eigenschaften der einzelnen Komponenten und der Komposition. Hierdurch können selbst komplexe Systeme in ihrer Gesamtheit verstanden werden.

Auch die oben genannten expliziten Schnittstellen erlauben ein besseres Verständnis der Wechselwirkungen zwischen den einzelnen Komponenten. Hierdurch wird zusätzlich Lokalisierbarkeit erreicht: Stellen, die für geänderte Anforderungen angepasst werden müssen und die entsprechenden Auswirkungen dieser Anpassungen können leichter lokalisiert und verstanden werden [Szy02].

Kompositionalität unterstützt Agilität. Ein kompositionaler Entwurf erlaubt es, Software in kleine Teile zu zerlegen und diese separat zu entwickeln. Durch diese Zerlegung ist es möglich, die einzelnen Komponenten agil zu ändern und anschließend in die verwendenden Systeme einzubinden.

Ein weiterer Aspekt der Agilität ist die separate Kompilierung. Selbst kleine Änderungen an monolithischen Systemen haben oftmals zur Folge, dass das Gesamtsystem erneut übersetzt werden muss oder zumindest eine aufwendige Prüfung stattfinden muss, welche Teile des Systems neu kompiliert werden müssen. Gerade bei großen Systemen kann eine solche Neuübersetzung bzw. Prüfung erheblich Zeit in Anspruch nehmen. Im Gegensatz dazu sind die Auswirkungen der Änderungen im kompositionalen Entwurf meist auf eine einzelne Komponente beschränkt, die Neukompilierung ist somit wesentlich schneller. Einzig wenn sich die Schnittstellen ändern,

[1] Unter dem Begriff Schnittstelle wird an dieser Stelle nicht nur die explizite syntaktische Schnittstelle verstanden. Vielmehr schließt der Begriff auch das Verhalten, also die Semantik, mit ein.

sind mehrere Komponenten betroffen, jedoch im seltensten Fall das Gesamtsystem. Daher ist die separate und bedingte Kompilierung nicht nur als Anforderung, sondern auch als Vorteil zu sehen.

Kompositionalität unterstützt Teamwork. Die kompositionale und modulare Entwicklung von Systemen unter Verwendung von Schnittstellen zwischen den Komponenten verbessert die Teamfähigkeit. Entwickler können sich auf ihre Komponenten mit deren klar umrissenen Aufgaben konzentrieren, ohne immer das Gesamtsystem betrachten zu müssen. Weiterhin können die Schnittstellen relativ früh im Entwicklungsprozess definiert werden. So kann die Entwicklung der Komponenten parallel erfolgen, ohne dass ein Entwickler zwingend abhängig von anderen ist und auf deren Ergebnisse warten muss [Cre94].

2.4. Zusammenfassung

Ziel dieses Kapitels war die Einführung der komponentenbasierten Softwareentwicklung als Grundlage zur kompositionalen Entwicklung domänenspezifischer Sprachen. Hierzu wurden zunächst die Charakteristiken der kompositionalen Softwareentwicklung vorgestellt. Zentraler Aspekt bildet dabei die Integration eigenständiger Komponenten zu einem Gesamtsystem anstelle einer monolithischen Entwicklung von Grund auf.

Aufbauend darauf wurden die Anforderungen an eine Komposition, speziell an deren Bestandteile Komponentenmodell, Kompositionstechnik und Kompositionssprache diskutiert. Hierbei bilden unter anderem die explizite Verwendung von Schnittstellen als Mittel zur Abstraktion und Unabhängigkeit, die Standardisierung und Adaption als Mittel zur Komponierbarkeit und die separate Kompilierung als Mittel zur Steigerung der Effizienz im Entwicklungsprozess eine zentrale Rolle.

Abschließend wurden die Vorteile kompositionaler Entwicklungsmethodiken vorgestellt. Herauszuheben hierbei sind Flexibilität, Erweiterbarkeit, Anpassbarkeit und Agilität. Durch diese Eigenschaften werden komplexe Systeme verständlicher und besser beherrschbar.

Kapitel 3.

Einführung in die Sprachentwicklung mit MontiCore

Nachdem in den letzten Kapiteln eine Einführung in die kompositionale Entwicklung von Softwaresystemen im Allgemeinen erfolgt ist und anschließend der Kompositionalitätsbegriff unabhängig von der Art der Artefakte dieser Arbeit - der Sprachen - untersucht wurde, wird in diesem Kapitel die Sprachentwicklung an sich vorgestellt. Hierzu werden zunächst domänenspezifische und universell einsetzbare Programmiersprachen erläutert und voneinander abgegrenzt. Anschließend wird diskutiert, aus welchen Bestandteilen eine Sprache besteht. Diese Bestandteile umfassen dabei alle Aspekte der Sprache, von der Syntax über Korrektheitsprüfungen bis hin zu sprachspezifischen Werkzeugen.

Im dritten Teilabschnitt wird das MontiCore-Framework zur Erstellung domänenspezifischer Sprachen eingeführt. Es wird unter anderem diskutiert, wie sich die konkrete und abstrakte Syntax in einem gemeinsamen Format spezifizieren lassen und welche Unterstützung über die reine Syntax hinaus existiert. Hierbei werden also die grundlegenden existierenden Funktionalitäten vorgestellt, die im Wesentlichen in der Dissertation [Kra10], aber auch in verschiedenen Publikationen [GKR+06, KRV07b, KRV07a, GKR+08, KRV08b, KRV10] unter Mitwirkung des Autors dieser Arbeit entwickelt wurden. Das MontiCore-Framework dient in den weiteren Kapiteln als technische Grundlage und wird sukzessive um neue Mechanismen erweitert.

3.1. DSLs und GPLs

In der Literatur [DKV00, KPKP06, JB06] existieren unterschiedliche Definitionen von domänenspezifischen Sprachen (domain specific language - DSL). Gemein ist diesen Definitionen, dass sie (a) auf eine bestimmte Domäne abzielen, (b) nur einen stark eingeschränkten Sprachumfang haben und (c), dass sie in den meisten Fällen nicht berechnungsuniversell sind (falls sie überhaupt ausführbar sind). Hierbei unterscheiden sich DSLs prinzipiell von universell einsetzbaren Programmiersprachen (general purpose language - GPL). Diese bieten sehr allgemeingültige Konzepte mit denen sich die Eigenschaften einer speziellen Domäne nur über spezielle Konstruktionen ausdrücken lassen. Objektorientierte Programmiersprachen bieten so beispielsweise die Bildung von Klassen mit entsprechenden Methoden und Attributen als allgemeines Konzept zur Beschreibung von Domänen an. Die Domänenspezifika sind also nicht Teil der Sprache, sondern können über Sprachmittel formuliert werden, im Gegensatz dazu besitzen DSLs bereits in der Sprache selbst Domänenkonzepte.

Die Abgrenzung zwischen DSL und GPL ist dabei keineswegs scharf. Selbst eine Programmiersprache wie Cobol kann als DSL für Geschäftsapplikationen gesehen werden [MHS05]. Die Fragestellung hierbei ist, inwieweit die Funktionalitäten der Sprache als streng domänenspezifisch oder als universell für viele Anwendungsbereiche gesehen werden. Für diese Arbeit ist diese Frage jedoch nicht von zentraler Bedeutung, sodass eine strikte Separierung nicht notwendig ist. Nichtsdestotrotz sollen an dieser Stelle einige typische DSLs und deren Verwendungszweck vorgestellt werden.

EBNF. Die EBNF [EBN] dient zur Spezifikation der kontextfreien Syntax von Sprachen. EBNF oder auch leicht veränderte Formen der EBNF werden häufig in Parsergeneratoren eingesetzt.

HTML. HTML [HTM] ist eine Sprache zur Spezifikation von Webseiten. Sie erlaubt unter anderem die Einbindung und Strukturierung von Inhalten wie Text, Bildern, Tabellen oder Hyperlinks.

GraphViz. GraphViz [Grab] ist eine textuelle Sprache zur Definition von Graphen. Aus einer textuellen Beschreibung können graphische Versionen generiert werden.

Tex. Tex [Knu79] ist ein Textsatzsystem mit zugehöriger Makrosprache zur Definition von Schriftstücken kleineren Umfangs bis hin zu umfangreichen Büchern.

Sprachen der UML. Die UML [OMG10c, OMG10b] mit ihren verschiedenen Teilsprachen ist eine Sprachfamilie zur Beschreibung von Softwaresystemen. Sie bietet unter anderem Sprachen zur Definition der Struktur, des Verhaltens oder der Verteilung von Software.

SQL. SQL [fS06] wird benutzt, um die Struktur von Daten in Datenbanken zu definieren und anschließend Datensätze einzufügen, abzufragen oder auch zu bearbeiten.

UNIX little languages. Das Betriebssystem Unix bietet eine Menge von kleinen Sprachen, die verschiedene Aufgaben erfüllen. Beispiele sind *grep* zur Suche und Filterung von Zeichenketten aus Dateien, *sed* zur automatisierten Änderung von Texten oder auch *make* zur Steuerung von Übersetzungsprozessen anhand expliziter Abhängigkeiten.

3.2. Bestandteile einer Sprache

Der Kern einer Sprache besteht im Wesentlichen aus 3 Artefakten [Kle07a]: konkrete Syntax, abstrakte Syntax und Semantik. Neben diesen Kernbestandteilen von Sprachen bedarf es jedoch für den praktischen Einsatz weiterer Artefakte. Hierzu zählen unter anderem Typsysteme und Symboltabellen, Kontextbedingungen und auf den Sprachen aufbauende Werkzeuge wie Editoren. An dieser Stelle sollen zunächst übersichtsartig die genannten Bestandteile vorgestellt und

abgegrenzt, sowie einige grundlegende Möglichkeiten zur Erstellung genannt werden. Für weitere, tiefergehende Aspekte und eine Vorstellung konkreter verwandter Arbeiten empfiehlt sich das jeweilige Kapitel dieser Arbeit.

Konkrete Syntax. Die konkrete Syntax entspricht dabei der Darstellungsform einer Sprache. Hier kann zwischen textuellen, graphischen oder Mischformen unterschieden werden [GKR+07]. Für textuelle Sprachen werden häufig Parsergeneratoren wie SableCC [GH98], Antlr [PQ95, Par07] oder ASF+SDF [BHD+01] verwendet. Elemente in graphischen Sprachen werden meist durch Icons oder einfache Graphiken dargestellt, typische Frameworks sind GMF [Graa, Gro09], MetaEdit+ [Met] oder Microsoft DSLTools [CJK07].

Abstrakte Syntax. Die abstrakte Syntax entspricht im Gegensatz zur konkreten Syntax der inneren Struktur von Sprachen unabhängig von der Darstellungsform gegenüber Nutzern. Zusätzlich wird oftmals bei der Übersetzung von der äußeren in die innere Repräsentationsform von semantisch irrelevanten Informationen abstrahiert (z.B. Layout, syntaktischer Zucker). Bekannte Werkzeuge zur Definition abstrakter Syntax in Form von Metamodellen sind EMF [BSM+03] und KM3 [JB06].

Semantik. Die Semantik beschreibt die Bedeutung der Sprache, wobei bekannte Ansätze zur Definition denotationale, operationale und translationale Semantiken sind. Dabei verwendet die denotationale Semantik mathematische Konstrukte und erklärt damit die Bedeutung der Sprachelemente, die operationale Semantik verwendet abstrakte Maschinen und erklärt das Verhalten anhand von Zustandsübergängen. Die translationale Semantik hingegen verwendet Transformationen auf eine wohlbekannte Sprache.

Die Erstellung einer Semantik ist eine der aufwendigsten und komplexesten Aspekte einer Sprachdefinition. Aufgrund dieser Komplexität ist die Erstellung eines Semantikframeworks für domänenspezifische Sprachen eine eigenständige Thematik und wird in einer gesonderten Dissertation erarbeitet [Grö10].

Typsysteme. Typsysteme dienen vor allem der Korrektheitsprüfung von Eingaben [Car97]; Artefakten oder Einheiten einer Sprache (z.B. Variablen) werden Typen zugeordnet, durch Berechnungsregeln werden anschließend die Typen komplexerer Ausdrücke berechnet und auf Kompatibilität bzw. Korrektheit geprüft. Dabei reichen die Eigenschaften von Typsystemen über ein weites Spektrum: von starker zu schwacher, von dynamischer zu statischer oder von expliziter zu impliziter Typisierung.

Zur Berechnung von Typen für Programmiersprachen lassen sich im Wesentlichen zwei Techniken benennen: Symboltabellen als zentrales Element werden benutzt, um Namen von Einheiten einer Sprache aufzulösen und damit verbundene weitergehende Informationen zu speichern und abzurufen. Hier kann so zum Beispiel der Typ einer Variable abgespeichert und unter dem Namen der Variable abrufbar sein. Als zweite Technik eignen sich Attributgrammatiken [Knu68] vor allem zur Berechnung des Typs zusammengesetzter Attribute.

Kontextbedingungen. Wie auch Typsysteme werden Kontextbedingungen vor allem zur Sicherstellung der Korrektheit der Eingaben verwendet. Neben grundlegenden Bedingungen wie der Parsbarkeit einer textuellen Eingabe werden meist bereits durch die Typisierung weitere Kontextbedingungen gestellt. Darüber hinaus können jedoch noch weitere Kontextbedingungen existieren, die oftmals gewünschte Eigenschaften einer Sprache bzw. deren Instanzen erzwingen. So wird zum Beispiel in Java geprüft, ob mögliche *checked exceptions* eines Methodenrumpfes auch in der Signatur einer Methode aufgelistet sind [GJS05].

Die Kontextbedingungen und deren Überprüfung werden oftmals in Form von Sourcecode formuliert. Eine andere Vorgehensweise bietet hier die OCL [OMG10a], in der Bedingungen einfach formuliert und Überprüfungen teilweise automatisch generiert werden können.

Werkzeuge. Sprachspezifische Werkzeuge stellen ein wesentliches Mittel dar, um den Komfort und die Benutzbarkeit einer Sprache und somit die Effizienz der Entwickler zu steigern. Für verbreitete Programmiersprachen und DSLs verfügen bereits einfache Editoren über sprachspezifisches Syntaxhighlighting. Hoch entwickelte IDEs hingegen haben sehr viele zusätzliche Funktionalitäten, von graphischen Outlines, die die Eingabe übersichtsartig darstellen, bis hin zu komplexen Debugging-Werkzeugen zur Fehleranalyse.

Werkzeuge zur Entwicklung von eigenen Sprachen stellen unterschiedliche Möglichkeiten zur Implementierung eigener sprachspezifischer Tools zur Verfügung. Eine reichhaltige Bibliothek hierfür bietet Eclipse [GB03], auch die Microsoft DSLTools [CJK07] bieten Unterstützung.

Codegenerierung. Vor allem beim Einsatz domänenspezifischer Sprachen spielt die Codegenerierung eine wesentliche Rolle. Hierbei wird die abstrakte Syntax eines Modells unter Zuhilfenahme zusätzlicher Infrastruktur (z.B. Symboltabellen) in eine ausführbare Zielsprache übersetzt. Neben der eigentlichen Codegenerierung können dabei durchaus weitere Generatoren existieren, etwa zur Erstellung von Dokumentationen oder zur Testfallgenerierung.

Im Wesentlichen lassen sich zwei verschiedene Ansätze zur Codegenerierung identifizieren: erstens kann eine Erstellung von Strings in einer Programmiersprache verwendet werden. Hier wird das Modell meist mittels spezieller Verfahren wie Visitoren oder Attributflüssen durchlaufen. Zweitens existieren Template-basierte Mechanismen (z.B. [SVB02, Fre]), wobei einige Teile des Templates in der Zielsprache und andere Teile in der Steuerungssprache der Template Engine geschrieben werden.

Wie auch die Semantik ist die Codegenerierung nicht Teil dieser Arbeit, sondern wird in einer gesonderten Dissertation [Sch11] behandelt.

3.3. Sprachentwicklung mit MontiCore

MontiCore [GKRS06, GKR+06, KRV08b, KRV07b, KRV07a, KRV08a, Kra10] ist ein Framework zur grammatikbasierten Erstellung domänenspezifischer Sprachen. Es verwendet ein integriertes Format zur Spezifikation der konkreten und abstrakten Syntax und erweitert traditionelle grammatikbasierte Ansätze um Konzepte aus der Metamodellierung wie Vererbung, Assoziationen und Interfaces.

Im Folgenden werden diese Möglichkeiten anhand einer einfachen Sprache - der erkennenden Automaten - dargelegt. Für eine detaillierte, vollständige Beschreibung der Möglichkeiten wird auf [Kra10, GKR+06] verwiesen.

3.3.1. Das Eingabeformat

Das Eingabeformat für MontiCore orientiert sich an traditionellen Formaten für Parsergeneratoren und ist insbesondere an das Eingabeformat von Antlr [PQ95, Par07] angelehnt, da MontiCore auf diesem Werkzeug aufbaut. Prinzipiell wird zwischen lexikalischen Produktionen, die das Verhalten des Lexers bestimmen, und Parserproduktionen zur Definition des Parsers unterschieden. Lexikalische Produktionen werden durch das Schlüsselwort `token` gefolgt vom Namen des Tokens und der Definition der Struktur in Form eines regulären Ausdrucks bestimmt. Zusätzlich kann einem Token ein Typ zugeordnet und ein Mapping vom Inhalt des Tokens zu einer Instanz dieses Typs zugeordnet werden [KRV08b]. Standarddefinitionen für die Erkennung von Identifiern und Strings sind bereits vordefiniert. Abbildung 3.1 zeigt ein einfaches Beispiel.

MontiCore-Grammatik «hw»

```
1  token Name = ('a' .. 'z' | 'A' ... 'Z')+;
```

Abbildung 3.1.: Beispiel für die Definition von lexikalischen Produktionen

Parserproduktionen werden zusätzlich in drei weitere Kategorien unterteilt: Klassenproduktionen, Schnittstellenproduktionen und abstrakte Produktionen, wobei an dieser Stelle zunächst nur die ersten beiden Varianten vorgestellt werden sollen. Klassenproduktionen werden wie in herkömmlichen Parsergeneratoren zur Definition von Nichtterminalen verwendet. Sie bestehen aus dem Namen des Nichtterminals gefolgt von der Beschreibung der erkannten Struktur. Diese Beschreibung besteht aus regulären Ausdrücken über der Menge der Nichtterminale (Verweise auf andere Produktionen) und Terminale (konstante Zeichenketten in Anführungszeichen). Hierbei können Verweise auf andere Produktionen durch Angabe des Namens gefolgt von einem Doppelpunkt vor der Referenz benannt werden.

Schnittstellenproduktionen definieren genau wie Klassenproduktionen Nichtterminale, wobei ihr Rumpf jedoch leer sein kann. Klassenproduktionen können Schnittstellen implementieren, indem sie das Schlüsselwort `implements` gefolgt vom Namen der Schnittstelle hinter dem Namen der Klassenproduktion verwenden. Eine zweite Möglichkeit besteht in der Auflistung aller die Schnittstelle implementierenden Klassenproduktionen im Rumpf der Schnittstellendefinition. Abbildung 3.2 illustriert ein einfaches Beispiel im MontiCore-Format und der entsprechenden EBNF-Form.

Explizit anzumerken ist hierbei, dass das MontiCore-Format keine Angaben über eine Startregel besitzt. Jede einzelne Parserproduktion kann als Startregel verwendet werden, da für jede Parserregel eigene Parser generiert werden. Hierdurch können auch Teile einer Sprache erkannt werden.

Neben diesen einfachen Definitionsmöglichkeiten existiert das Konzept der Regelvererbung, welches vor allem im Falle einer Grammatikvererbung (siehe Kapitel 4.2.1) zum Einsatz kommt.

MontiCore-Grammatik «hw»

```
1 Automaton = Content* ;
2
3 interface Content;
4
5 State implements Content =
6        "state" Name;
7
8 Transition implements Content =
9        from:Name "->" to:Name;
10
```

EBNF

```
1 Automaton ::= Content* ;
2
3 Content ::= State | Transition;
4
5 State ::= "state" Name;
6
7
8 Transition ::= Name "->" Name;
9
10
```

Abbildung 3.2.: Beispiel für Klassen- und Schnittstellenproduktionen

MontiCore-Grammatik «hw»

```
1 Automaton = Content* ;
2
3 interface Content;
4
5 State implements Content =
6        "state" name:Name;
7
8 Transition implements Content =
9        from:Name "->" to:Name;
10
11 //Regelvererbung
12 FinalState extends State =
13        "finalstate" name:Name;
14
```

EBNF

```
1 Automaton ::= Content* ;
2
3 Content ::= State | Transition;
4
5 State ::= "state" Name |
6           FinalState;
7
8 Transition ::= Name "->" Name;
9
10
11
12 FinalState ::= "finalstate"
13                Name;
14
```

Abbildung 3.3.: Beispiel für Regelvererbung

Hierbei können Klassenproduktionen voneinander erben, es entstehen ähnliche Auswirkungen wie bei Schnittstellenproduktionen. Durch die Verwendung der Regelvererbung können die abgeleiteten Regeln an allen Stellen erkannt werden, an denen auch die ursprüngliche Regel erkannt werden kann. Abbildung 3.3 zeigt ein weiteres Beispiel und die entsprechende EBNF-Form.

Als weitere wichtige Eigenschaft von MontiCore-Grammatiken ist die Namensgebung zu nennen. Eine Grammatik hat grundsätzlich einen Namen und wird in einem Paket, welches wie in Java der Verzeichnisstruktur entspricht, abgelegt. Die Paketinformation wird zusätzlich im Kopf der Grammatik angegeben, der voll qualifizierte Name der Grammatik ergibt sich durch Konkatenation des Paket- und des Grammatiknamens. Abbildung 3.4 zeigt eine Grammatik mit dem voll qualifizierten Namen `mc.Automaton`.

MontiCore-Grammatik «hw»

```
1 package mc;
2
3 grammar Automaton{
4
5    Automaton = Content* ;
6
7    interface Content;
8
9    State implements Content = "state" Name;
10
11   Transition implements Content = from:Name "->" to:Name;
12
13   FinalState extends State = "finalstate" Name;
14 }
```

Abbildung 3.4.: Grammatik mit dem voll qualifizierten Namen `mc.Automaton`

3.3.2. Ableitung der abstrakten Syntax

Ausgehend von der Grammatik wird für jede Klassenproduktion eine separate gleichnamige Klasse und für jede Schnittstellenproduktion ein Interface erstellt. Dabei werden Verweise auf der rechten Regelseite einer Klassenproduktion wie folgt umgesetzt: Ist das Ziel des Verweises eine lexikalische Produktion, entsteht ein Attribut in der Klasse, wobei der Name der Referenz als Attributname verwendet wird. Der Typ des Attributs ist im Standardfall String, wurde bei der Definition der lexikalischen Produktion ein anderer Typ angegeben, wird dieser übernommen. Ist das Ziel jedoch eine andere Parserproduktion, entsteht eine Kompositions-Beziehung zwischen den beteiligten Klassen. Die verwendete Regelvererbung führt zu einer objektorientierten Vererbung und die implements-Beziehung zu einer Schnittstellenproduktion führt zur implements-Beziehung zum entsprechenden Interface. Abbildung 3.5 zeigt die entstehende abstrakte Syntax für das Beispiel aus Abbildung 3.3.

Wie in dieser Abbildung zu erkennen ist, können zusätzlich Kardinalitäten an den Kompositionsbeziehungen entstehen. Hierbei wird ausgewertet, wie oft eine assoziierte Instanz derselben Klasse minimal bzw. maximal vorkommen kann. Im Beispiel existiert eine *-Beziehung, da in der Grammatik der Kleene-Stern verwendet wurde.

Zusätzlich zu den dargestellten Attributen werden verschiedene Hilfsmethoden generiert. So werden für jedes Attribut get- und set-Methoden erstellt, es existieren Methoden zum Durchlauf des AST (Abstract Syntax Tree) mit Hilfe eines Visitors [GHJV95] und Methoden zum Vergleichen von ASTs. Weiterhin besteht die Möglichkeit, selbst Attribute und Methoden anzugeben, die in die entstehenden Syntaxklassen eingewoben werden.

Wie bereits zu Beginn dieses Teilkapitels angesprochen, ist MontiCore in der Lage, zusätzlich Assoziationsbeziehungen zwischen den beteiligten Klassen zu erstellen. Dabei ist der Ansatz zweigeteilt: zunächst wird spezifiziert, welche Assoziationen zwischen welchen Klassen bzw. Interfaces existieren. Nachfolgend muss spezifiziert werden, wie die Links zwischen den assoziierten Objekten nach der Syntaxanalyse aufgebaut werden. Hierbei besteht die Möglichkeit,

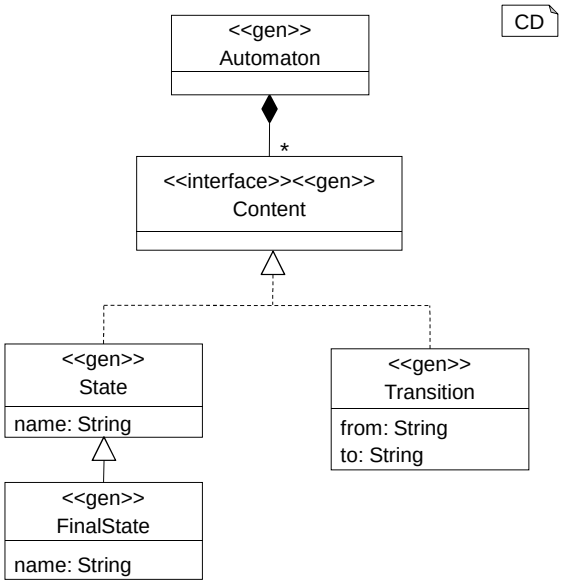


Abbildung 3.5.: Abstrakte Syntax des Beispieles aus Abbildung 3.3

einfache Regeln in einem speziellen Format anzugeben. Ein Beispiel hierfür ist die Identifikation der beteiligten Objekte anhand eindeutiger Namen innerhalb des Modells. Sind kompliziertere Regeln zur Berechnung notwendig, kann der Benutzer diese Regeln in Form von Quellcode hinterlegen. Abbildung 3.6 zeigt ein Beispiel, in dem eindeutige Namen zur Identifikation verwendet werden.

MontiCore-Grammatik «hw»

```
1  association Source
2     State.OutgoingTransitions 1 <-> * Transition.SourceState;
3
4  concept sreference{
5     Source: State.name = Transition.from;
6  }
```

Abbildung 3.6.: Beispiel für die Definition von Assoziationen

Hierbei wird in den ersten beiden Zeilen die Assoziation `Source` definiert, wobei die zweite Zeile besagt, dass eine Transition mit genau einem Zustand assoziiert ist. Dieser Zustand ist durch das Attribut `SourceState` erreichbar. Im Umkehrschluss sind beliebig viele Transitionen mit einem Zustand durch das Attribut `OutgoingTransitions` assoziiert.

Die Zeilen 4 bis 6 spezifizieren anschließend, wie die Objekte miteinander verbunden werden. Hierbei wird der Name der Assoziation wiederholt (`Source:`) und anschließend beschrieben, dass eine Transition mit einem Zustand genau dann verbunden ist, wenn der Name des Zustandes dem Inhalt des Attributes `from` einer Transition entspricht. Aus dieser Spezifikation

werden anschließend einerseits entsprechende Attribute und Zugriffsmethoden in den Klassen der abstrakten Syntax erzeugt, andererseits entstehen Komponenten, die die Links zwischen den beteiligten Objekten automatisch erstellen.

Zusätzlich zu den hier vorgestellten grundlegenden Eigenschaften existieren weitergehende Möglichkeiten, die konkrete und abstrakte Syntax zu spezifizieren. Hierzu gehören die Definition von Enumarationsproduktionen, die Verwendung von Produktionsparametern oder mehrteilige Klassenproduktionen (siehe [Kra10]). Einige dieser Techniken werden bei Notwendigkeit innerhalb dieser Arbeit separat eingeführt.

3.3.3. Parsertechnologie

MontiCore verwendet als unterliegenden Parsergenerator ANTLR [PQ95, Par07], welcher Parser im Recursive-Descent-Stil generiert. Die ASTs werden top-down aufgebaut und Linksableitungen gebildet. Im Gegensatz zur herkömmlichen LL(k)-Eigenschaft [ASU86] wird der feste Lookahead durch die Angabemöglichkeit syntaktischer Prädikate erweitert, die Technologie wird daher als LL(*) bezeichnet. Abbildung 3.7 zeigt die erweiterten Möglichkeiten des LL(*)-Ansatzes.

MontiCore-Grammatik «hw»

```
1  A = (B* C =>) B* C |
2      (B* D =>) B* D;
3  B = "b";
4  C = "c";
5  D = "d";
```

Abbildung 3.7.: Beispiel für den Einsatz syntaktischer Prädikate

In diesem Beispiel sind zwei syntaktische Prädikate jeweils zu Beginn der Alternativen für `A` in Klammern angegeben. Grund hierfür ist, dass die Alternativen für kein k die LL(k)-Eigenschaft erfüllen, da jede der Alternativen mit beliebig vielen „b“ beginnen kann. Im herkömmlichen LL(k)-Verfahren müsste an dieser Stelle eine Linksfaktorisierung des `B*`-Anteiles der Alternativen erfolgen. Durch den Einsatz der Prädikate prüft jedoch der Parser, ob der folgende Text gemäß des im Prädikat angegebenen Inhaltes geparst werden kann. Dieser Inhalt wird dann zunächst in einen Puffer abgelegt und - bei Erfolg der Prädikaterkennung - innerhalb der Alternative abgebaut.

In den meisten Fällen kann jedoch eine Angabe eines syntaktischen Prädikats entfallen, wenn der Lookahead des Parsers genügend groß eingestellt ist. Hierfür existiert die Möglichkeit, im Kopf einer Grammatik die Größe des Lookaheads zu definieren. Neben dem Lookahead für den Parser kann hier zusätzlich ein Lookahead für den Lexer angegeben werden, da dieser nicht wie üblich auf endlichen Automaten basiert, sondern die gleiche Erkennungsstrategie wie der Parser verwendet.

Alternativ zu den syntaktischen Prädikaten können auch semantische Prädikate verwendet werden. Diese Prädikate sind in der Zielsprache des generierten Parsers zu formulieren (ANTLR beherrscht unterschiedliche Zielsprachen, u.a. Java und C++) und müssen boolesche Ausdrücke

sein. Diese Prädikate werden ungeprüft übernommen und innerhalb eines `if`-Konstrukts in den generierten Parser eingewoben. Somit können beliebige Bedingungen zur Auswahl der Alternativen formuliert werden.

Neben dem Einsatz von Code als semantisches Prädikat zur Auswahl der Alternativen können auch Aktionen formuliert werden. Diese können an beliebiger Stelle innerhalb einer Produktion angegeben werden und werden ausgeführt, falls der Erkennungsprozess an die entsprechende Stelle gelangt. Der automatische Aufbau des ASTs ist beispielsweise durch diese Technik umgesetzt worden.

Hervorzuheben ist an dieser Stelle, dass die ANTLR-Technologie keine Modularitätsmechanismen besitzt. Für die Zwecke dieser Arbeit wurde deshalb ANTLR modifiziert. Die einzelnen Modifikationen werden in den speziellen Kapiteln näher erläutert.

3.3.4. Das DSLTool-Framework

Neben der Generierung von Lexern, Parsern und AST-Klassen bietet MontiCore eine Vielzahl an Funktionalitäten, die das Erstellen eines sprachverarbeitenden Werkzeugs unterstützen. Die für diese Arbeit notwendigen Aspekte werden im Folgenden kurz vorgestellt und die grundlegenden Aufgaben umrissen.

Roots. In MontiCore wird jede Eingabedatei durch sogenannte *Roots* repräsentiert. Diese enthalten neben dem eigentlichen Modell in Form des ASTs zusätzliche Informationen wie zum Beispiel Symboltabellen. Außerdem ermöglichen die Roots Zugriff auf weitere Komponenten wie dem Parser, dem PrettyPrinter oder einer Komponente zur Standardisierung von Fehlermeldungen. Typischerweise wird für jede Modellart eine eigene Root-Klasse als Subklasse von `mc.DSLRoot` erstellt, um zusätzliche sprach- und modellspezifische Inhalte aufzunehmen.

RootFactory. Roots werden prinzipiell in speziellen Fabriken [GHJV95], den *RootFactories* erzeugt. Sie können dabei hierarchisch angeordnet werden, wobei die Auswahl der zuständigen Fabrik für ein Modell auf Basis verschiedener Kriterien erfolgen kann (z.B. Dateiendung). Demnach existiert für jede Modellart eine eigene RootFactory.

Workflows. Hat eine RootFactory eine Root erstellt, so wird diese initial mit dem Inhalt der Datei gefüllt. Im Verlaufe der Bearbeitung sind nun verschiedene Algorithmen für unterschiedliche Aufgaben zuständig: zunächst wird der Dateiinhalt geparst, ein AST erstellt und in der Root abgelegt. Anschließend wird typischerweise die Symboltabelle erstellt, dann erfolgen beispielsweise Transformationen bevor letztendlich Code erzeugt wird. Diese Algorithmen werden separat als Dateien abgelegt - den *Workflows*. Diese kapseln also den algorithmischen Anteil der Bearbeitung und sind zustandslos, während die Roots die Datenhaltung darstellen und zustandsbehaftet sind.

Modelloader. Eine wichtige Rolle im Verlaufe dieser Arbeit spielt der *Modelloader*. Dieser ist in der Lage, Modelle anhand des vollqualifizierten Namens nachzuladen und eine entsprechende

Root für dieses Modell zurückzuliefern. Hierbei kann innerhalb des Frameworks angegeben werden, welche Workflows beim Nachladen ausgeführt werden sollen - beispielsweise das Parsen und der Symboltabellenaufbau. Diese Funktionalität ist vor allem bei der Prüfung von Kontextbedingungen notwendig, da die Korrektheit eines Modells meist von den Eigenschaften anderer Modelle abhängt.

Visitor. MontiCore stellt eine speziell auf Kompositionalität abgestimmte Version des *Visitors* [GHJV95] zur Verfügung. Dieser kann ein Modell entlang des ASTs durchlaufen und spezielle Aufgaben wie PrettyPrinting, Symboltabellenaufbau oder das Überprüfen von einfachen Kontextbedingungen durchführen. Die Kompositionalitätsaspekte des speziellen Visitor-Ansatzes und deren Auswirkungen werden detailliert im nächsten Teilkapitel besprochen.

Attributgrammatik-System. Weiterhin stellt das DSLTool-Framework ein Attributgrammatik-System [Knu68, Ste08] zur Verfügung. Hierbei wird die Grammatik um die Deklaration von Attributen erweitert, wobei die Attributberechnungen im Gegensatz zu traditionellen Ansätzen nicht in der Grammatik spezifiziert werden, sondern direkt in Java-Code. Durch diese Vorgehensweise steht die volle Funktionalität einer objektorientierten GPL zur Verfügung. Auch das Attributgrammatiksystem ist auf Kompositionalität ausgelegt und wird in verschiedenen Teilbereichen dieser Arbeit Anwendung finden.

Weitere Komponenten. Neben den vorgestellten Komponenten existiert eine Vielzahl anderer unterstützender Funktionalitäten. Unter diesen befinden sich Komponenten zur einheitlichen Fehlermeldung, eine Unterstützung zur Dateierzeugung oder eine Ablaufsteuerungs-Komponente. Detaillierte Beschreibungen finden sich in [Kra10].

3.4. Zusammenfassung

Das Ziel dieses Kapitels bestand in der Einführung in die Entwicklung von Sprachen. Hierzu wurden zunächst domänenspezifische Sprachen vorgestellt und von universell einsetzbaren Programmiersprachen abgegrenzt. Wesentliche Merkmale der DSLs waren dabei die Domänenspezifität, der eingeschränkte Sprachumfang und - im Falle ausführbarer Sprachen - die meist fehlende Berechnungsuniversalität. Die Auflistung bekannter domänenspezifischer Sprachen zeigt die weite Verbreitung dieser Technologie.

Im folgenden Teilabschnitt wurden dann die Bestandteile einer Sprache vorgestellt. Diese reichen von der äußeren Darstellung - der konkreten Syntax - über Mechanismen zur Korrektheitsprüfung wie Symboltabellen oder Kontextbedingungen hin zu sprachverarbeitenden Werkzeugen und Algorithmen wie der Codegenerierung. Die Breite dieser verschiedenen Artefakte unterstreicht dabei die Komplexität und den hohen Aufwandsbedarf bei der Sprachentwicklung.

Im dritten Teilabschnitt wurde das MontiCore-Framework zur Entwicklung textueller Sprachen vorgestellt. Es ermöglicht die integrierte Definition der konkreten und abstrakten Syntax in einem Format und ist in der Lage, aus dieser Spezifikation unter anderem Lexer, Parser und

die Klassen der abstrakten Syntax zu generieren. Weiterhin bietet MontiCore eine reichhaltige Unterstützung bei der Entwicklung sprachverarbeitender Werkzeuge an.

Teil II.

Kompositionale Sprachentwicklung

Kapitel 4.

Anwendungen und Techniken

In den letzten Kapiteln erfolgten grundlegende Betrachtungen zu verschiedenen Teilaspekten dieser Arbeit. Diese umfassten unter anderem die kompositionale Softwareentwicklung, sowie eine Einführung in die Entwicklung von Sprachen und in das MontiCore-Framework.

Ziel dieses Kapitels ist nun eine weitergehende Diskussion der Kompositionalität im speziellen Anwendungsfall der Sprachentwicklung. Hierzu werden zunächst mehrere Beispiele sowohl aus der Modellierungswelt als auch aus der Welt der Programmiersprachen aufgeführt, welche die Notwendigkeit der kompositionalen Sprachentwicklung unterstreichen. Diese Beispiele zeigen zusätzlich die verschiedenen benötigten Mechanismen auf.

Anschließend werden die drei Kompositionsmechanismen Spracheinbettung, Sprachaggregation und Sprachvererbung eingeführt. Spracheinbettung ermöglicht dabei die Kombination verschiedener Teilsprachen für ein Modell. Sprachaggregation hingegen dient zur Kombination unterschiedlicher Sprachen über Modellgrenzen hinweg und Sprachvererbung zur Spezialisierung bzw. Erweiterung existierender Sprachen.

Im dritten Teil wird der generelle Aufbau sprachverarbeitender Werkzeuge vorgestellt und die Anforderungen kompositionaler Systeme für die vorgeschlagene Architektur diskutiert.

Abschnitt vier führt die verwendeten Techniken zur Entwicklung kompositionaler, sprachverarbeitender Werkzeuge ein. Diese Techniken umfassen dabei sowohl geeignete Design Patterns, als auch generelle Techniken, die Kompositionalität unterstützen und dienen als Grundlage für den weiteren Verlauf dieser Arbeit.

4.1. Einsatz multipler Sprachen

Der Einsatz mehrerer Sprachen ist eine weit verbreitete Technik zur Definition eines Softwaresystems. Hierbei werden verschiedene Sprachen eingesetzt, um die unterschiedlichen Aspekte eines Softwaresystems zu spezifizieren. Die Art der eingesetzten Sprachen variiert dabei stark: Modellierungssprachen können verwendet werden, um Teile des Systems auf sehr hohem Abstraktionsgrad zu beschreiben. Programmiersprachen oder auch Teile davon können dann eingesetzt werden, um Spezifika auf niedrigem Abstraktionsniveau zu definieren. Zusätzlich zu Programmier- und Modellierungssprachen werden oftmals Sprachen eingesetzt, die auf die jeweilige zu entwickelnde Funktion zugeschnitten sind (z.B. SQL für Datenhaltung). Die Kombination der verwendeten Sprachen ist dabei jedoch nicht immer gleich, sondern variiert je nach Projektkontext.

Im Folgenden wird an verschiedenen Beispielen der Einsatz mehrerer Sprachen aufgezeigt.

Diese Beispiele zeigen dabei die angesprochene Vielfalt der eingesetzten Spracharten sowie die Kombinationsmöglichkeiten einer Sprache mit verschiedenen anderen.

4.1.1. OCL als DSL in der Modellierung

Die Object Constraint Language [OMG10a] ist eine textuelle Sprache zur Formulierung von Invarianten und Vor-und Nachbedingungen. Die OCL zeichnet sich dadurch aus, dass Bedingungen prinzipiell an einen Kontext geknüpft sind, etwa an eine Klasse in einem Klassendiagramm oder Objekte in Objektdiagrammen. Abbildung 4.1 zeigt ein Beispiel der Kombination von Klassendiagrammen und OCL.

Klassendiagramm

```
1 classdiagram masterdata{
2
3    class Person{
4      int age;
5    }
6
7 }
```

OCL

```
1 context Person p inv:
2   p.age >=0
3
4
5
6
7
```

Abbildung 4.1.: Beispiel für den Einsatz der OCL mit Klassendiagrammen

Durch die benötigte Angabe des Kontextes wird unmittelbar ersichtlich, dass die OCL nie separat, sondern immer in Verbindung mit einer anderen Hostsprache verwendet werden muss. Hierfür gibt es eine Vielzahl möglicher Kombinationen, allein in [Rum04b] und [Rum04a] wird die OCL mit 5 verschiedenen Sprachen (Klassendiagramme, Objektdiagramme, Statecharts, Sequenzdiagramme und Java) verwendet.

Diese Eigenschaft unterstreicht den Bedarf an der kompositionalen Entwicklung der Sprachen. Allein durch die große Vielfalt an Kombinationen ist es nicht zielführend, eine statische Kopplung der Sprachen zu entwickeln. Ein kompositionaler Ansatz ist hier zu bevorzugen.

4.1.2. SQL als DSL in der Programmierung

SQL ist eine domänenspezifische Sprache zur Definition, Abfrage und Manipulation von relationalen Datenbanken. Sie wird häufig in Verbindung mit Programmiersprachen eingesetzt, um die Persistenz der anfallenden Daten zu realisieren. Bekannte Beispiele für die Verbindung einer GPL mit SQL sind Java, C++ oder PHP. Abbildung 4.2 zeigt ein Beispiel zur Verwendung von SQL innerhalb Javas.

Diese Abbildung unterstreicht die fehlende Integration von Java und SQL: die SQL-Query in Zeile 8 wird als reiner String angegeben und vom Java-Compiler ungeprüft übernommen. Eventuelle Fehler bei der Formulierung der Anfrage werden daher erst zur Laufzeit entdeckt. Wünschenswert wäre hier eine Integration der Sprachen, sodass sowohl von rein syntaktischer Seite als auch auf Ebene der Korrektheitsprüfung bis hin zu IDEs eine vollständige Unterstützung beider Sprachen stattfindet. Zwar existieren für manche Sprachen bereits solche Ansätze (z.B. im .NET-Framework [MBB06]), diese verbinden jedoch die beteiligten Sprachen fest und

Java-Quellcode

```
1 //init database connection
2 Class.forName("com.mysql.jdbc.Driver");
3 Connection con = DriverManager.getConnection(
4         "jdbc:mysql://localhost:3306/SimpleDB", "User", "Password");
5 Statement st = con.createStatement();
6
7 //execute a query
8 ResultSet set = st.executeQuery("SELECT * FROM Person");
```

Abbildung 4.2.: Beispiel für den Einsatz Java und SQL

sind nicht auf andere Sprachen übertragbar. Auch hier würde ein kompositionaler Ansatz Abhilfe schaffen.

4.1.3. Kombination von Modellierung und Programmierung: Statecharts und Java

Statecharts [OMG10c, OMG10b] sind eine zustandsbasierte Modellierungssprache, die unter anderem verwendet werden kann, um das Verhalten von Objekten einer Klasse zu spezifizieren. Hierzu wird ein Statechart einer konkreten Klasse zugeordnet, wobei diese entweder durch ein Klassendiagramm oder auch durch eine Java-Klasse gegeben sein kann. Abbildung 4.3 zeigt ein Beispiel.

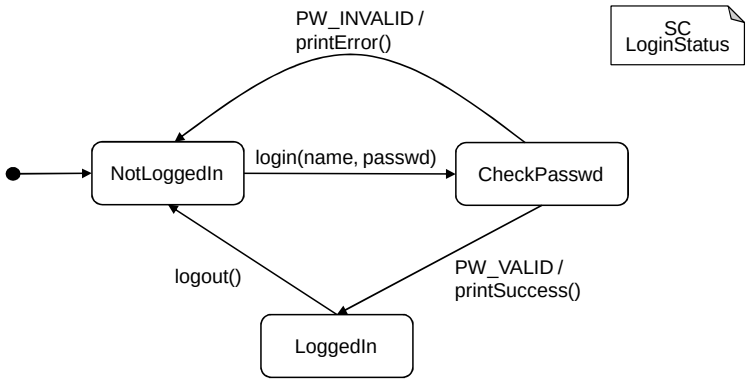
In diesem Beispiel modelliert das Statechart das Verhalten von Objekten der Klasse `LoginStatus`. Nachdem ein Objekt eine Nachricht zum Einloggen mit Benutzername und Passwort erhalten hat, prüft es im Zustand `CheckPasswd` zunächst die Gültigkeit. Ist das Passwort korrekt, wird in den Zustand `Login` gewechselt und eine Erfolgsnachricht ausgegeben, andernfalls erfolgt ein Übergang zum Ausgangszustand und eine Fehlermeldung wird angezeigt. Empfängt das System die Aufforderung zum Ausloggen im Status `LoggedIn`, wird wiederum zum Ausgangszustand übergegangen.

In der zweiten Hälfte der Abbildung ist in Auszügen die eigentliche Klasse `LoginStatus` in Java definiert. Beachtenswert sind die Bedingungen in den `if` bzw. `else if` Anweisungen. Aus Java-Sicht wird hier auf boolesche Variablen zugegriffen, die jedoch nicht in der Klasse selbst definiert sind. Vielmehr sind diese an die Namen der Zustände des Statecharts gekoppelt, wobei sich der Name einer Variablen aus dem Präfix `isIn_` gefolgt vom Zustandsnamen zusammensetzt. Hier werden demnach Zustände als Variablen identifiziert - andere denkbare Szenarien sind Zustände als enum-Werte oder die Verwendung des State-Musters.

Auch in diesem Fall ist es nicht wünschenswert, die Sprachen Java und Statecharts fest miteinander zu verbinden. Eine lose Kopplung im kompositionalen Sinne ist vorzuziehen.

4.1.4. Weitere Beispiele

Neben den aufgeführten Beispielen zur Kombination von DSLs, Modellierungs- und Programmiersprachen existieren eine Vielzahl weiterer aktueller Anwendungsgebiete, in denen mehrere



Java-Quellcode

```
1 public class LoginStatus{
2
3   public void printStatus(){
4      if (isIn_NotLoggedIn){
5         print("Not logged in");
6      }
7      else if (isIn_CheckPasswd){
8         print("Checking password");
9      }
10     else{
11        print("Logged in");
12     }
13  }
14 }
```

Abbildung 4.3.: Kombination von Java und Statecharts

Sprachen eingesetzt werden. Diese sollen an dieser Stelle nur kurz erläutert werden. Für tiefergehende Beschreibungen empfehlen sich die jeweiligen Literaturreferenzen.

Java-Code innerhalb von JSPs. Java Server-Pages [Ber03] sind eine Web-Programmiersprache zur dynamische Erzeugung von HTML/XML-Seiten innerhalb eines Websystems. Die Syntax besteht dabei aus HTML- bzw. XML-Tags, in die Java-Anteile eingebettet werden können. Die JSPs werden von einem Compiler zu reinem Java-Code übersetzt, wobei die Java-Anteile übernommen und die HTML/XML-Anteile umgewandelt werden.

Template-Engines. Template-Engines bieten einen Mechanismus zur einfachen Generierung von Quellcode an. Ein Template beinhaltet dabei typischerweise zwei verschiedene Anteile: zum einen werden statische Teile des erwarteten Generats in der Zielsprache verfasst, zum anderen werden für die dynamischen Anteile spezielle Direktiven benutzt (z.B. Schleifen, if-Bedingungen oder auch Variablenzuweisungen) oder externer Sourcecode aufgerufen. Die Kor-

rektheit der Kombination der verschiedenen Sprachanteile aus der Zielsprache und der Sprache der Template-Engine wird in den meisten Fällen nicht statisch geprüft. Beispiele für Template Engines finden sich in [GC03] oder unter [Fre].

MontiCore/ANTLR. ANTLR [Par07] als Parsergenerator und darauf aufbauend MontiCore bieten die Möglichkeit, Javacode in die Grammatik einzubetten. Wie schon im vorigen Kapitel beschreiben, dient dieser Code als semantisches Prädikat der Auswahl von Alternativen einer Produktion. Weiterhin können Aktionen mittels Javacode definiert werden. Diese Aktionen werden ausgeführt, sobald der Erkennungsprozess die entsprechende Stelle erreicht. Diese Technik wird in MontiCore zum Aufbau des AST genutzt.

Wie diese Beispiele aufzeigen, existiert eine Vielzahl von Anwendungen, in denen mehrere Sprachen eingesetzt werden. Die Form der Kombinationen sind dabei unterschiedlicher Natur. Einerseits werden Anteile verschiedener Sprachen in einem einzigen Modell eingesetzt (JSPs oder MontiCore), andererseits existieren auch Beispiele, in den Sprachen über Modellgrenzen hinweg kooperieren (z.B. Statecharts und Java).

4.2. Kompositionalitätsmechanismen des MontiCore-Frameworks

Zur kompositionalen Entwicklung von Sprachen bietet das MontiCore-Framework drei verschiedene Mechanismen an: Grammatikvererbung zur Erweiterung oder Einschränkung einer existierenden Sprache, Einbettung zur Kombination verschiedener Sprachanteile innerhalb eines Dokumentes und Sprachaggregation zur Verwendung mehrerer Sprachen über Dokumentgrenzen hinweg.

4.2.1. Grammatikvererbung

Erster zentraler Kompositionalitätsmechanismus des MontiCore-Frameworks ist die *Grammatikvererbung*. Hierfür kann im Kopf der Grammatik eine `extends`-Klausel angegeben werden, gefolgt vom voll qualifizierten Namen einer oder mehrerer anderer Grammatiken. Abbildung 4.4 zeigt ein Beispiel.

In dieser Abbildung wurde die Beispielgrammatik der Automaten aus Abbildung 3.4 erweitert, um sie durch eine neue Transitionsart zu ergänzen. Diese ermöglicht es, zusätzlich Aktionen an den Transitionen in Form eines Strings zu definieren (die Lexerregel String ist vom MontiCore-Framework standardmäßig vorgegeben). Anzumerken ist hierbei, dass die Grammatik vollständig ist, d.h. alle Regeln der Supergrammatik sichtbar sind. Somit können diese einfach referenziert oder wie im obigen Beispiel der Transition erweitert werden. Zusätzlich ist es möglich, die Regeln der Supergrammatik zu überschreiben, indem eine Regel unter Verwendung des gleichen Namens redefiniert wird.

Die obige Technik der Vererbung von einem Nichtterminal in der Subgrammatik wird typischerweise angewandt, wenn der Entwickler der Supergrammatik die Erweiterungen in Subgrammatiken nicht voraussehen kann. Sind diese Erweiterungen jedoch vorhersehbar, empfiehlt

MontiCore-Grammatik «hw»

```
1  package mc2;
2
3  grammar AutomatonWithActions extends mc.Automaton{
4
5      TransitionWithActions extends Transition =
6          from:Name "-" action:STRING ">" to:Name;
7
8  }
```

Abbildung 4.4.: Beispiel für die Grammatikvererbung

es sich oftmals, Erweiterungspunkte in der Supergrammatik durch Schnittstellenproduktionen zu setzen. Im Beispiel der Automatengrammatik wurde das Interface `Content` eingeführt, wodurch Subgrammatiken leicht neue Inhalte hinzufügen können, indem Regeln definiert werden, die dieses Interface implementieren.

4.2.2. Einbettung

Zweites Konzept zur Modularisierung von Sprachen ist die *Einbettung*. Hierbei werden in einer Grammatik explizite Löcher in Form externer Nichtterminale ohne Produktionskörper vorgesehen. Abbildung 4.5 zeigt ein Beispiel.

MontiCore-Grammatik «hw»

```
1  package mc3;
2
3  grammar AutomatonWithExternalActions extends mc.Automaton{
4
5      external Action;
6
7      TransitionWithActions extends Transition =
8          from:Name "-" action:Action ">" to:Name;
9
10 }
```

Abbildung 4.5.: Beispiel für externe Nichtterminale

In dieser Abbildung wird in Zeile 5 ein externes Nichtterminal `Action` definiert, welches auf der rechten Regelseite der `TransitionWithAction` referenziert wird. Hintergrund ist hierbei, dass die Automatensprache unabhängig von der verwendeten Aktionssprache definiert wird. Somit kann sie in verschiedenen Kontexten wiederverwendet werden, etwa im Verbund mit Java oder C++.

Um eine solche Kombination zu erreichen, wird der generierte Parser/Lexer der Hostsprache (hier der Automatenparser) mit dem generierten Parser/Lexer der eingebetteten Sprache

kombiniert. Es erfolgt hierfür zunächst eine separate und unabhängige Generierung der Parser und Lexer. Zur Konfigurationszeit eines Werkzeuges wird dann festgelegt, bei welchem externen Nichtterminal ein Übergang zu welcher Zielsprache stattfindet. Dies kann einerseits durch eine separate DSL festgelegt werden, andererseits kann dies durch die Nutzung von API-Funktionalitäten erfolgen. Explizit anzumerken ist die Möglichkeit, verschiedene Zielsprachen für unterschiedliche externe Nichtterminale festzulegen. Da MontiCore-Grammatiken keine explizite Startregel besitzen, können sogar Teile von Sprachen (z.B. Statements oder Expressions aus Java) eingebettet werden.

Konzeptuell kann die Grammatikeinbettung mit der Integration von Grammatikfragmenten aus [Läm01] verglichen werden. Grammatikfragmente enthalten dabei im Gegensatz zu den Standardversionen kontextfreier Grammatiken Referenzen zu undefinierten Nichtterminalen. Demnach kann obiges Beispiel gedanklich durch zwei Grammatiken ersetzt werden, die in einem weiteren Schritt integriert werden, wobei das externe Nichtterminal der ersten Grammatik durch das ersetzende Nichtterminal der zweiten Grammatik ersetzt wird. Hieraus würde eine Gesamtgrammatik ohne undefinierte Nichtterminale entstehen.

4.2.3. Sprachaggregation

Wie bereits im Abschnitt 3.3.4 beschrieben, ermöglicht das DSLTool-Framework die Verarbeitung verschiedener Sprachen innerhalb eines gemeinsamen Werkzeuges. Hierzu werden verschiedene RootFactories eingebunden, die je nach Typ des zu verarbeitenden Modells unterschiedliche Roots produzieren. Zudem ist es möglich, je nach Rootart unterschiedliche Workflows - z.B. verschiedene Workflows zum Parsen - zu definieren.

Durch dieses Konzept wird die Aggregation von Sprachen ermöglicht: das Gesamtwerkzeug wirkt von außen wie die Integration der beteiligten Parser, Lexer etc. Gedanklich kann diese Vorgehensweise durch die Erstellung eines neuen Parser ersetzt werden, der dann je nach Art des Modells den entsprechenden sprachspezifischen Parser aufruft. Insbesondere wird durch die Integration die Basis für das Zusammenspiel zwischen den beteiligten Sprachen ermöglicht. Durch ein Gesamtwerkzeug ist es so möglich, auch sprachübergreifende Symboltabellen, Kontextbedingungen etc. zu entwickeln. Als Beispielanwendung für die Sprachaggregation sei die schon in Abbildung 4.3 beschriebene Kombination von Java und Statecharts mit ihren dort beschriebenen Eigenschaften aufgeführt.

Insgesamt bestehen zwischen Einbettung und Aggregation grundlegende technische Unterschiede in Bezug auf den Erkennungsprozess, die gemeinsame AST-Struktur im Falle der Einbettung und dem weiteren Verarbeitungsprozess. Aus konzeptueller Sicht können beide Versionen jedoch gegeneinander ausgetauscht werden: im obigen Fall der Transitionen mit Aktionen kann die alte Grammatik ohne Aktionen und eine zweite Grammatik speziell für Aktionen eingesetzt werden. Aus Zuordnungszwecken muss zusätzlich ein Konzept existieren, sodass an Aktionen angegeben werden kann, für welche Transition sie erstellt wurden. Im gegebenen Beispiel erscheint dies zwar aus Übersichtsgründen nicht unbedingt sinnvoll, es existieren jedoch Fälle, in denen die Wahl nicht eindeutig ist. Hier spielen die erwartete Größe der Modelle und persönliche Präferenzen der Nutzer eine wesentliche Rolle. Zusätzlich ist es möglich, beide Alternativen anzubieten und den Nutzer je nach Kontext flexibel entscheiden zu lassen.

4.3. Anwendungsszenarien kompositionaler Sprachentwicklung

Die zu Beginn dieses Kapitels aufgezeigten Beispiele zur kompositionalen Verwendung von Sprachen lassen sich in verschiedene Anwendungsszenarien kategorisieren. Dabei ist die Art der Sprache (DSL, GPL, Modellierungssprache etc.) nicht von Bedeutung, sondern lediglich die Methodik der Kombination. Insgesamt ergeben sich hierdurch sechs verschiedene Anwendungsszenarien zur kompositionalen Sprachentwicklung.

4.3.1. Sprachvereinigung

Der offensichtlichste Anwendungsfall von Sprachkombination ist die Sprachvereinigung. Hierbei wird davon ausgegangen, dass mindestens zwei vorhandene Sprachen und die dazugehörige Infrastruktur miteinander kombiniert werden, um anschließend ein Werkzeug zu erhalten, das alle Eingangssprachen unterstützt. Als prominentes Beispiel für dieses Szenario dient die UML. Die verschiedenen Teilsprachen können hier zunächst separat entwickelt und anschließend in ein Gesamtwerkzeug integriert werden. Hierbei müssen meist neue Komponenten entwickelt werden, die beispielsweise sprachübergreifende Kontextbedingungen prüfen.

4.3.2. Nichtterminal-Erweiterung

Bei der Nichtterminal-Erweiterung wird ein Nichtterminal einer existierenden Sprache um den Inhalt eines anderen Nichtterminals erweitert. Dabei dient die Kombination von Java und SQL als Beispiel. Anstatt der bisher existierenden JDBC-Funktionalität, bei der SQL-Anteile als String in Java verwendet wird, wird hier das existierende Nichtterminal für Expressions in Java um SQL-Anweisungen erweitert. Das Resultat ist dabei ein Nichtterminal, das sowohl die alten Java-Anteile als auch zusätzlich die SQL-Syntax beinhaltet.

4.3.3. Nichtterminal-Ersetzung

Im Gegensatz zum vorangegangenen Szenario wird hier ein existierendes Nichtterminal einer Sprache durch ein anderes Nichtterminal einer weiteren Sprache ersetzt. Dieses Szenario tritt häufig auf, wenn Sprachen nichtkompositional entwickelt wurden und eine der beteiligten Sprachen durch eine andere ersetzt werden soll. Beispielhaft kann hier eine Sprache für Automaten mit fest integriertem C++ als Aktionssprache aufgeführt werden. Dieses Szenario fordert hierbei die Ersetzung des C++-Aktions-Anteils beispielsweise durch Java.

4.3.4. Einbettung einer einzelnen Sprache

Bei diesem Szenario wird das Konzept der Spracheinbettung vorausgesetzt. Während der Konfigurationszeit des Werkzeuges wird das beteiligte externe Nichtterminal durch ein Nichtterminal einer anderen Sprache ersetzt. Diese Sprachkombination bleibt während der gesamten Laufzeit des Werkzeuges bestehen. Als illustratives Beispiel dient hierbei wiederum die Kombination von Automaten und Java als Aktionssprache.

4.3.5. Einbettung von Sprachalternativen

Im Gegensatz zur Einbettung einer einzelnen Sprache wird bei der Einbettung von Sprachalternativen zur Konfigurationsszeit lediglich festgelegt, welche Sprachen die Rolle eines externen Nichtterminals einnehmen können. Zur Laufzeit wird dann entschieden, welche der angegebenen Sprachen eingesetzt wird. Diese Festlegung kann dabei entweder einmalig für das gesamte Modell/Programm erfolgen oder auch jedesmal einzeln an jeder Stelle von Sprachübergängen. Angewandt auf das Beispiel der Automaten könnte sich so der Benutzer bei jeder Aktion einzeln entscheiden, ob er Java oder C++ verwendet.

4.3.6. Entwicklung einer erweiterbaren Sprache

Neben den obigen Szenarien besteht ein weiterer Anwendungsfall in der Entwicklung einer erweiterbaren Sprache. Hierbei steht nicht von vornherein fest, mit welcher weiteren Sprache auf welche Art der Sprachkomposition kombiniert werden soll. Vielmehr beschreibt dieses Szenario die Entwicklung einer Sprache, die gezielt Erweiterungspunkte setzt, um eine spätere Kombination zu erleichtern. Bei der Diskussion dieses Szenarios werden in den verschiedenen Kapiteln Möglichkeiten aufgezeigt, diese Erweiterungspunkte unter Antizipation möglicher Kombinationen zu definieren.

4.4. Korrelationen zwischen Kompositionsmechanismen und Anwendungsszenarien

In den beiden vorangegangen Teilkapiteln wurden einerseits die verschiedenen *Kompositionsmechanismen* des MontiCore-Frameworks vorgestellt und andererseits die *Szenarien zur Sprachkomposition* eingeführt. Diese beiden Begrifflichkeiten sind dabei von unterschiedlicher Natur: die Mechanismen bieten die technische Voraussetzung zur Umsetzung der Szenarien. Von Nutzerseite aus stehen jedoch die Szenarien im Vordergrund, da sein Ziel in der Umsetzungsmöglichkeit für ein gegebenes Problem liegt. Daher ist für den Nutzer eine problemorientierte Sichtweise auf die Sprachkomposition unter Anwendung bestimmter Techniken gegenüber einer technischen Diskussion der Mechanismen und deren Anwendungsgebiete von Vorteil. Die problemorientierte Sichtweise spiegelt sich daher in der Struktur der weiteren Kapitel zu den Sprachbestandteilen wider.

Prinzipiell ergeben sich unterschiedliche Korrelationen zwischen Anwendungsszenarien und Mechanismen. Wie in den späteren Kapiteln ausführlich erläutert wird, können die Szenarien teilweise auf unterschiedliche Art und insbesondere unter Verwendung verschiedener oder auch mehrerer Mechanismen umgesetzt werden. Zusätzlich ergeben sich verschiedene Randbedingungen bei der Verwendung unterschiedlicher Mechanismen. Daher ist eine eins-zu-eins Zuordnung zwischen Szenario und Mechanismus nicht möglich. Vielmehr dienen die Mechanismen in einer bestimmten Kombination der Umsetzung eines Szenarios.

Zusätzlich zu den eingeführten Szenarien existieren weitere Anwendungsgebiete in der Kombination der Szenarien. Komplexe Werkzeuge können dabei mehrere verschiedene Szenarien beinhalten. So ist es möglich, dass zwei Sprachen zunächst per fester Einbettung kombiniert

werden und anschließend eine dritte Sprache durch Sprachvereinigung hinzugenommen wird. Das Gesamtszenario ist in diesen Fällen in die einzelnen Teilszenarien aufzuschlüsseln, diese können dann separat behandelt und umgesetzt werden.

4.5. Modell- und Sprachkomposition

Modellkomposition als Mittel zur Beherrschung komplexer, modellbasierter Softwaresysteme hat eine enge Beziehung zur kompositionalen Entwicklung von Sprachen. Diese Beziehung manifestiert sich dabei auf zwei Arten: zum einen kann Sprachkomposition als Instanz der Modellkomposition betrachtet werden und zum anderen ermöglicht Sprachkomposition eine kompositionale Verarbeitung von Modellen. Um diese Beziehungen zwischen Modell- und Sprachkomposition näher vorstellen zu können, werden im nächsten Teilabschnitt zunächst die Eigenschaften der Modellkomposition gemäß [HKR+07] grob umrissen, um anschließend die genaueren Zusammenhänge zu erläutern.

4.5.1. Modellkomposition

Grundlage für die Diskussionen bildet die Erkenntnis, dass der Kompositionalitätsbegriff prinzipiell auf zwei verschiedenen Ebenen betrachtet werden kann [HKR+07]. Einerseits existiert immer eine semantische Ebene - die Bedeutung der einzelnen Modelle. Die eigentliche Komposition findet auf dieser Ebene statt, wobei Nutzer sowohl den einzelnen Modellen als auch der Komposition an sich eine Bedeutung geben. Auf der anderen Seite existiert die syntaktische Ebene sowohl der Modelle als auch des *Kompositionsoperators*. Das wesentliche Problem besteht darin, dass die Benutzer niemals auf semantischer, sondern immer auf syntaktischer Ebene arbeiten [HR04]: sowohl die Modelle als auch der Kompositionsoperator werden mittels einer geeigneten Notation definiert. Anschließend führt ein geeignetes Kompositionsprogramm die Komposition aus und liefert wiederum ein Ergebnis auf syntaktischer Ebene, welches der Benutzer interpretiert und ihm somit eine Semantik zuweist. Entsprechen sich syntaktische und semantische Komposition, entspricht das Ergebnis dem des vom Benutzer Gewollten, andernfalls liegen höchstwahrscheinlich Fehler in der Definition des syntaktischen Kompositionsoperators vor. In diesem Fall verifiziert der Benutzer die syntaktische Ebene gegen die semantische und versucht, Unstimmigkeiten zu entfernen.

Aus formaler Sicht spielt bei diesen Betrachtungen die *semantische Abbildung* eine zentrale Rolle: diese bildet die Modelle der syntaktischen Ebene auf die entsprechende Semantik ab. Wie bereits in [HKR+07] ausführlich diskutiert, ist die Abbildung dabei mengenwertig: jedem Modell werden alle dem Modell entsprechenden Elemente der semantischen Domäne zugeordnet. Die Abbildung erfolgt daher in die Potenzmenge der Elemente der semantischen Domäne.

Definition 4.5.1 (Semantische Abbildung)**.** *Sei U_M das Universum der Modelle und U_S das Universum der Elemente der semantischen Domäne. Die semantische Abbildung sm ordnet jedem Modell $m \in U_M$ potentiell mehrere Elemente $s \in U_S$ zu:*

$$sm : U_M \rightarrow \wp(U_S)$$

Die Komposition auf syntaktischer Ebene kann ausgehend vom Universum der Modelle U_M wie folgt definiert werden.

Definition 4.5.2 (Syntaktischer Kompositionsoperator)**.** *Ein syntaktischer Kompositionsoperator $\otimes$ ist eine Funktion, die aus zwei Ursprungsmodellen ein komponiertes Zielmodell erzeugt: $\otimes : U_M \times U_M \rightarrow U_M$.*

Ebenso kann auch auf semantischer Ebene ein Kompositionsoperator definiert werden. Hierbei werden jedoch nicht einzelne Elemente der semantischen Domäne komponiert, sondern jeweils Mengen von Elementen, da den Betrachtungen eine mengenwertige Semantik unterliegt.

Definition 4.5.3 (Semantischer Kompositionsoperator)**.** *Ein semantischer Kompositionsoperator $\oplus$ ist eine Funktion, die aus zwei Elementen der Potenzmenge über den Elementen der semantischen Domäne D ein integriertes Element erzeugt:*

$$\oplus : \wp(U_S) \times \wp(U_S) \rightarrow \wp(U_S).$$

Ausgehend von diesen Definitionen und den eingangs des Teilabschnitts erfolgten Erläuterungen kann nun ein Zusammenhang zwischen Syntax, Semantik und Komposition von Modellen identifiziert werden. Abbildung 4.6 zeigt diesen Zusammenhang.

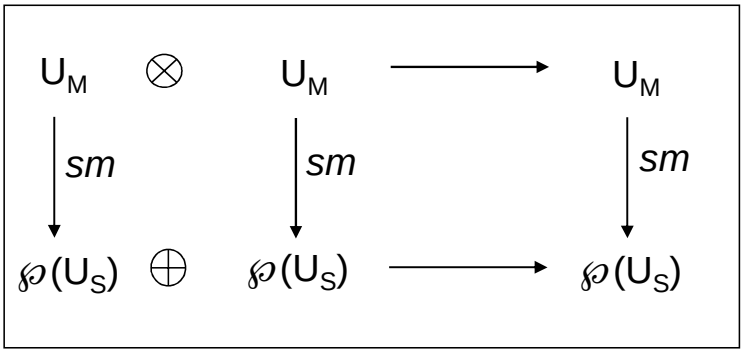


Abbildung 4.6.: Zusammenhang zwischen semantischem und syntaktischem Kompositionsoperator

Auf dieser Abbildung sind sowohl der syntaktische (obere Hälfte, von links nach rechts) als auch der semantische Kompositionsoperator (untere Hälfte, von links nach rechts) gemäß ihrer Definitionen erkennbar. Weiterhin ist an den jeweiligen Stellen die semantische Abbildung sm zwischen Syntax und Semantik aufgeführt. Zusätzlich kann der Abbildung der Zusammenhang der Kompositionsoperatoren entnommen werden: entsprechen sich syntaktische und semantische Komposition bildet die semantische Abbildung einen Homomorphismus zwischen den beteiligten Elementen. Hierdurch kommutiert das Diagramm, d.h. es ist irrelevant ob zunächst auf Ebene der Syntax komponiert und anschließend die Semantik gebildet wird oder ob zuerst die Semantik der Modelle gebildet und anschließend auf semantischer Ebene komponiert wird. Aus mathematischer Sicht gilt in diesem Falle demnach

$$sm(m_1) \oplus sm(m_2) = sm(m_1 \otimes m_2)$$

Diese grundlegenden Betrachtungen und die tiefergehenden Untersuchungen in [HKR+07] haben unmittelbaren Einfluss auf die modellbasierte kompositionale Entwicklung. Hierbei ist es das Ziel, die Modelle so lange wie möglich einzeln zu halten ohne ein integriertes Gesamtmodell bilden zu müssen. Erst durch die Entsprechung von Syntax und Semantik können die Modelle einzeln verstanden und behandelt werden. Insbesondere ist es hierdurch möglich, separat Code zu generieren und die einzelnen Bestandteile des Systems in einem späteren Entwicklungsschritt zu integrieren. Des Weiteren ermöglicht ein kompositionaler Ansatz ein Verständnis des Gesamtsystems aufgrund der Eigenschaften der einzelnen Modelle und des Kompositionsoperators.

4.5.2. Sprachkomposition als spezielle Form der Modellkomposition

Die aufgeführten Ziele der Modellkomposition entsprechen denen der kompositionalen Sprachentwicklung. Auch hierbei werden die Bestandteile - die Teilsprachen - separat entwickelt und erst in einem späteren Schritt integriert. Die Bildung einer Gesamtsprache - z.B. in Form einer vereinigenden Grammatik - als Ausgangspunkt zur Codegenerierung ist nicht nötig.

Neben den Zielen können auch die theoretischen Grundlagen auf Sprachkomposition angewandt werden, wobei jeder Teilsprache eine geeignete Semantik zugeordnet wird. Diese Semantik muss dabei die unterschiedlichen Aspekte der Sprachentwicklung beinhalten: bei der konkreten Syntax eignet sich beispielsweise die Menge der erkannten Worte bzw. alle möglichen Parsebäume und bei der abstrakten Syntax die Menge aller möglichen ASTs, wobei diese durch Kontextbedingungen und Symboltabellen weiter eingeschränkt werden können. Zusätzlich zur eigentlichen Semantik kann dann ein semantischer Kompositionsoperator definiert werden, der diese Semantiken geeignet komponiert. Durch eine solche Vorgehensweise ist es möglich, kompositionale Sprachentwicklung auf die Modellkomposition zurückzuführen und somit die Sprachkomposition als Instanz der Modellkomposition zu betrachten.

4.5.3. Unterstützung der kompositionalen Modellierung

Die kompositionale Modellierung hat ihr Hauptziel in der separaten Verarbeitung von Modellen. In vielen Aspekten kann eine solche Separierung erreicht werden, wobei Fälle existieren, in denen eine vollkommene Loslösung vom Kontext nicht erreicht werden kann. Beispielhaft

ist hier die Korrektheitsprüfung von Modellen zu nennen: die Korrektheit eines Modells hängt oftmals von anderen Modellen ab, etwa wenn Referenzen auf andernorts definierte Elemente existieren. Hierbei kann die Korrektheit nur unter Einbezug des Kontextes geprüft werden. Zusätzlich zur Korrektheitsprüfung existieren weitere Aspekte, in denen Informationen zwischen Modellen bzw. auf Modellen arbeitenden Anwendungen (z.B. Codegeneratoren) ausgetauscht werden müssen. Den Prinzipien der Kapselung und des information hiding folgend, sollte die ausgetauschte Information dabei minimal sein.

Diese Beispiele zeigen insgesamt den Bedarf an spezieller Infrastruktur zur kompositionalen, modellbasierten Entwicklung auf. Diese Infrastruktur sollte Mittel und Wege zur Verfügung stellen, Modelle prinzipiell separat zu verarbeiten und im Falle nötiger Kontextinformationen zusätzliche Unterstützung bieten. Wie im Laufe dieser Arbeit verdeutlicht wird, stellen MontiCore und insbesondere die Teilaspekte der Symboltabellen und Kontextbedingungsprüfungen eben diese Funktionalitäten zur Verfügung. Daher kann der in dieser Arbeit entwickelte Ansatz als Grundlage zur kompositionalen, modellbasierten Entwicklung verwendet werden.

4.6. Aufbau kompositionaler, sprachverarbeitender Werkzeuge

Wesentliches Kernziel dieser Arbeit besteht darin, die Sprachen möglichst separat zu definieren und eine Möglichkeit zur anschließenden Kombination der Sprachen zu schaffen. Daher wird zunächst versucht, soviele Anteile der Sprache wie möglich separat zu generieren bzw. implementieren. Generell ist dabei von einer konkreten Kombination mit anderen Sprachen zu abstrahieren. So sollte zum Beispiel bei der Definition der OCL nicht davon ausgegangen werden, dass diese später einzig mit Klassendiagrammen kombiniert wird. Dies ist vor allem bei der Programmierung zu beachten: im Quellcode sollten keine Referenzierungen auf Klassen oder sonstige Elemente erfolgen, die wiederum aus anderen Sprachen generiert bzw. für diese programmiert sind.

Im Allgemeinen ist eine vollkommene Separierung nicht möglich, da Informationen von einer Sprache in eine andere Sprache übertragen werden müssen. So muss die Information, dass ein bestimmter Typ „A“ im Klassendiagramm definiert ist, in die Sprache OCL übertragen werden, wenn dort eine Invariante für eben diese Klasse spezifiziert ist. Die Information, wie Teile einer Sprache in einer anderen Sprache interpretiert und somit wiederverwendet werden können, ist jedoch immer von der konkreten Sprachkombination und auch von der Interpretation einer Sprache bezüglich der anderen abhängig. Daher ist das Ziel einer 100%-igen Separierung nicht zu erfüllen. Vielmehr ist das Ziel daher eine separate Definition von Sprachen und deren Interfaces mit anschließender Implementierung oder Generierung von Glue Code, der eben diesen Informationsaustausch zwischen festgelegten Sprachen ermöglicht. Der Aufwand für die Erstellung des Glue Codes sollte dabei jedoch wesentlich kleiner sein als der Aufwand des Erstellens eines nichtkompositionalen Werkzeuges für die konkrete Sprachkombination.

Neben den Komponenten für die einzelnen Sprachteile und dem Glue Code, welcher die Komponenten der einzelnen Sprachen kombiniert und somit zwischen ihnen vermittelt, ist ein wesentlicher Bestandteil eines sprachverarbeitenden Werkzeuges unabhängig von den beteiligten Sprachen. Dieser kann in Form von Bibliotheken und/oder Frameworks implementiert werden.

Das Framework ist hier zum einen in der Lage, immer wiederkehrende Aufgaben wie zum Beispiel Filehandling oder Fehlermeldungen zu unterstützen. Zum anderen kann durch ein Framework eine grundlegende und vor allem einheitliche Struktur und Funktionsweise der Komponenten vorgegeben werden. Dies erleichtert die anschließende Integration, denn unterschiedliche Implementierungen ließen sich nur unter erheblichem Aufwand zusammenführen. Abbildung 4.7 fasst den prinzipiellen Aufbau eines kompositionalen, sprachverarbeitenden Werkzeuges zusammen.

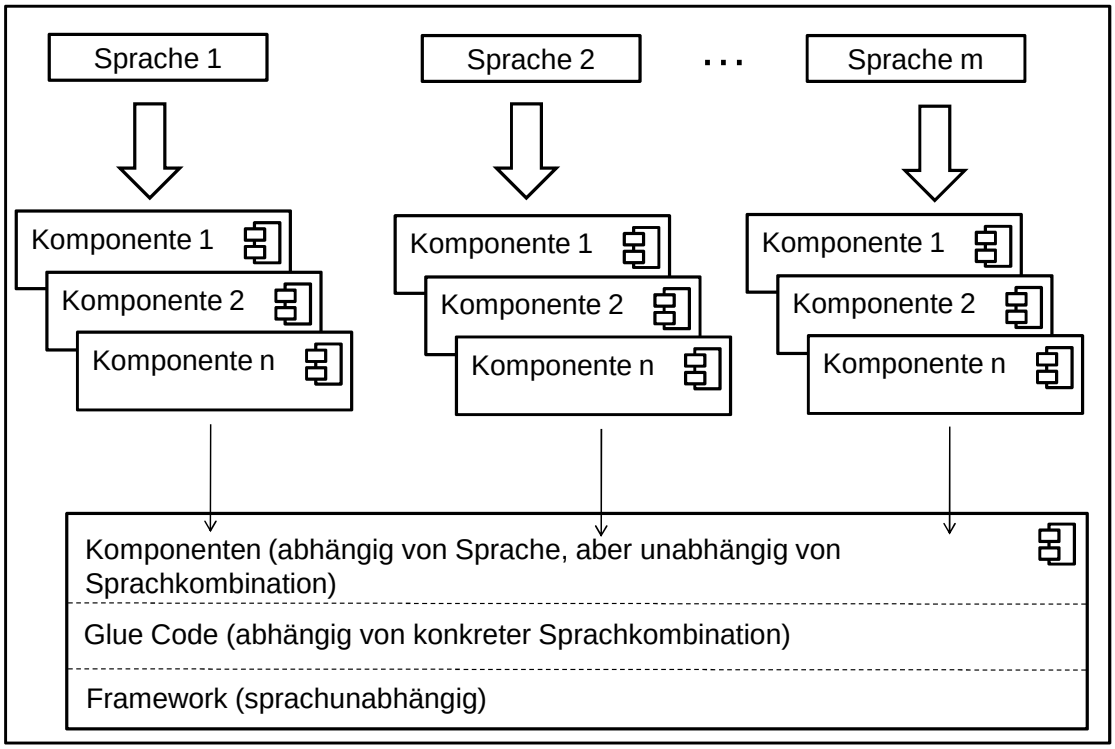


Abbildung 4.7.: Genereller Aufbau eines kompositionalen, sprachverarbeitenden Werkzeuges

4.6.1. Kombination von Sprachen

Prinzipiell lässt sich feststellen, dass Sprachkombinationen hierarchisch aufgebaut sind. Auf oberster Ebene wird dabei Sprachaggregation betrieben: ein Werkzeug muss für jedes Modell entscheiden, in welcher Sprache es geschrieben ist und die entsprechenden Komponenten für diese Sprache zur Ausführung bringen. Innerhalb eines Modells können wiederum verschiedene Teilsprachen durch Einbettung zum Einsatz kommen. Auch diese Einbettung kann Hierarchien aufweisen: so ist es möglich, dass Java-Blöcke innerhalb von Klassendiagrammen eingebettet sind und innerhalb dieser Java-Blöcke - beispielsweise zur Spezifikation von Vor- und Nachbedingungen - OCL verwendet wird. Dieser hierarchische Aufbau lässt sich am einfachsten durch das Composite-Muster [GHJV95] wiedergeben. Abbildung 4.8 zeigt den generellen Aufbau von Sprachen.

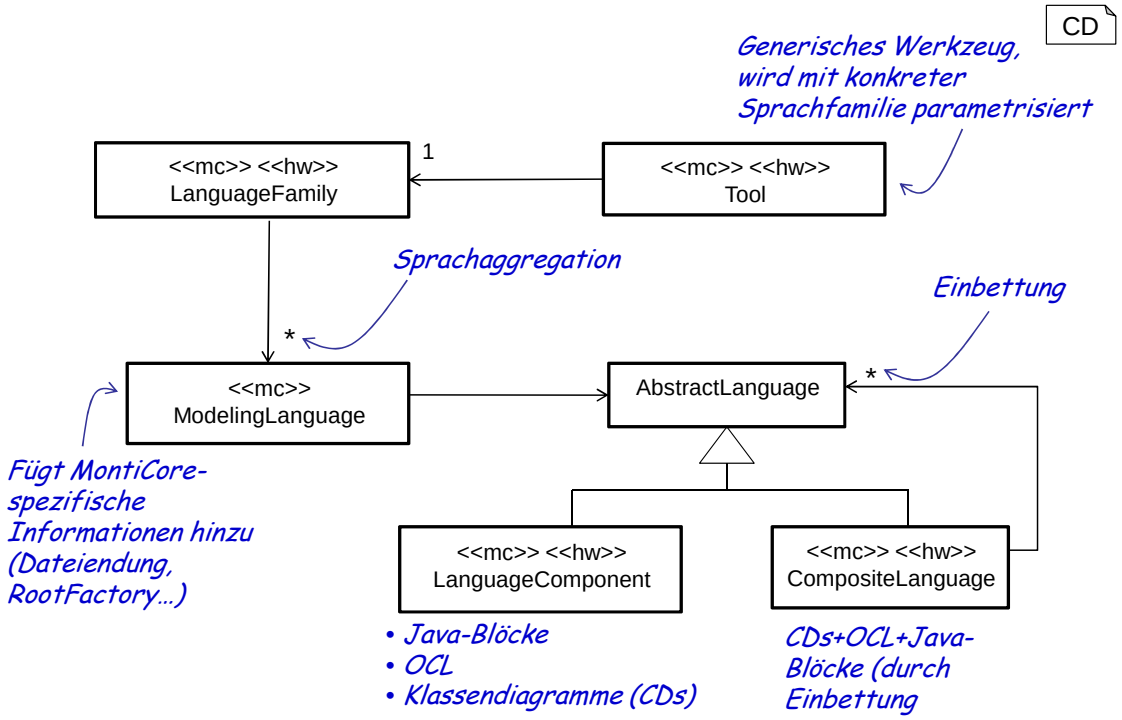


Abbildung 4.8.: Verwendung des Composite-Musters für Sprachen

Wie aus dieser Abbildung ersichtlich ist, wurden die Ebenen, die durch Einbettung und Aggregation entstehen, getrennt. Die Hauptgründe hierfür sind dabei technischer Natur: eine Sprache, die für ein gesamtes Modell zuständig ist, muss spezifische Informationen wie Dateiendung oder die von MontiCore benötigte RootFactory mitliefern. Konzeptuell kann die dargestellte Sprachhierarchie jedoch als reine Anwendung des Composite-Musters gesehen werden.

Zusätzlich ist auf der Abbildung die Möglichkeit zur Wahl der Kombination zu erkennen. Im rechten Teil werden die beteiligten Sprachen OCL, Klassendiagramme und Java-Blöcke zunächst separat erstellt und dann in einem weiteren Schritt kombiniert. Alternativ könnten auch zunächst zwei der drei beteiligten Sprachen kombiniert und anschließend die dritte hinzugefügt werden. Die Wahl ist dabei dem Sprachentwickler überlassen und hängt im Wesentlichen davon ab, ob die Kombination einer Teilmenge der Sprachen sinnvoll erscheint bzw. Wiederverwendungspotential hat. Im angegebenen Beispiel wäre es möglich, zunächst Klassendiagramme mit OCL zu kombinieren, um in diesem Schritt offen zu halten, ob anschließend Java oder C++ hinzugefügt wird. Hierdurch können die Anteile für die Kombination von Klassendiagrammen und OCL in beiden Fällen wiederverwendet werden.

Insgesamt lassen sich die `CompositeLanguage` sowie die `LanguageFamily` als Glue sehen. Durch Verwendung des Composite-Musters auf Einbettungsebene werden die einzelnen Sprachen bzw. auch Sprachkombinationen wiederum zu Komponenten und können wiederverwendet werden.

4.6.2. Umsetzung der Anforderungen an kompositionale Systeme

An dieser Stelle sollen die in Kapitel 2.2 vorgestellten Anforderungen an eine Komposition am vorgeschlagenen Aufbau eines sprachverarbeitenden Werkzeuges diskutiert werden. *Unabhängigkeit* ist vor allem bezüglich der aus den einzelnen Sprachen entstandenen Komponenten vorhanden, da diese sich nicht gegenseitig referenzieren und keinerlei Kenntnis voneinander haben. Lediglich der Glue Code verbindet die Komponenten, dessen Notwendigkeit schon im Punkt *Verzögerung* diskutiert worden ist. Durch die Einführung des Glue Codes anstatt einer direkten Kenntnis der Umgebung und somit der anderen Sprachen sind die Komponenten self-contained und können leicht in andere Umgebungen eingebettet werden. Explizite *Schnittstellen* zwischen den Komponenten werden in den einzelnen Teilkapiteln näher vorgestellt, da diese sich vor allem innerhalb der Definitionen der Teilkomponenten (Syntaxen, Typsysteme, Kontextbedingungen und Werkzeuge) widerspiegeln. Beispiele hierfür sind das Einführen von externen Nichtterminalen bezüglich der Syntax oder die explizite Einführung von Modell-Interfaces. *Standardisierung* findet im Wesentlichen durch das vorgegebene Framework statt. Hierin wird der Aufbau und die Arbeitsweise einzelner Komponenten vorgegeben und somit eine Kopplung erleichtert.

Die letzten an dieser Stelle diskutierten Eigenschaften sind *Konsistenzprüfung* und *separate Kompilierung*. Ersteres wird innerhalb einer Sprache durch entsprechende Komponenten für diese Sprache übernommen, wobei für sprachübergreifende Prüfungen der Glue Code verantwortlich zeichnet. Der zweite Punkt - die separate Kompilierung - ist nur durch den konsequenten Einsatz verschiedener Techniken möglich, die es erlauben, Teile eines Systems separat zu definieren, zu kompilieren und anschließend zu kombinieren. Auf diese Techniken wird speziell im nächsten Teilkapitel eingegangen.

4.7. Techniken zur Entwicklung kompositionaler sprachverarbeitender Werkzeuge

Eine nicht auf das Ziel der Kompositionalität abgestimmte Architektur von Teilkomponenten zur Sprachverarbeitung kann eine Kombinierbarkeit erschweren bzw. verhindern. Daher ist das Verständnis, wie Sourcecode Kompositionalität unterstützen kann von wesentlicher Bedeutung. Hier spielen vor allem Eigenschaften wie Wiederverwendbarkeit, Adaptierbarkeit und Flexibilität eine tragende Rolle. Diese Kriterien sind unter anderen auch Randbedingungen von *Design Patterns* [GHJV95]. Aber nicht nur Design Patterns, sondern auch allgemeinere Entwurfsrichtlinien tragen zur Verbesserung der Kompositionalitätsaspekte von Sourcecode bei. Im Folgenden werden daher sowohl die Richtlinien als auch die für diese Arbeit relevanten Design Patterns vorgestellt. Anzumerken ist hierbei, dass diese nicht nur im erstellten Framework verwendet wurden, sondern auch bei der Erstellung konkreter Komponenten für spezifische Sprachen angewandt werden müssen, da nur eine konsequente Architektur aller Sprachen Kombinierbarkeit garantieren kann. Daher sind sie zusätzlich als Referenz für die Entwicklung neuer Sprachen zu sehen.

4.7.1. Frameworks und Bibliotheken

Eine zentrale Fragestellung beim Entwurf eines Systems zur kompositionalen Sprachverarbeitung besteht darin, ob es in Form eines Frameworks oder in Form von Bibliotheken realisiert werden sollte. Der Hauptunterschied besteht hierbei in der Ablaufkontrolle: während bei Bibliotheken die Ablaufkontrolle beim Benutzerprogramm - im Falle dieser Arbeit bei den zu implementierenden Komponenten - liegt, übernehmen Frameworks die Kontrolle und rufen die vom Benutzer implementierten Funktionen zu den jeweils passenden Momenten auf [JF88].

Es gibt mehrere Vorteile von Frameworks gegenüber Bibliotheken [Hau97]. Erstens ist es möglich, die Aufteilung der Teilaufgaben festzulegen, da ein Framework typischerweise Interfaces oder abstrakte Klassen vorgibt, die jeweils eine bestimmte Funktion erfüllen müssen. Zweitens fließen Erfahrungen unmittelbar ins Framework ein. Ein sehr gut durchdachtes Framework kann so Entwicklern neuer Komponenten die richtige Architektur oder Vorgehensweise vorgeben. Drittens ermöglichen Frameworks einen einheitlichen Aufbau aller Applikationen, was wiederum die Integration erleichtert.

Da diese Gründe den Anforderungen an ein kompositionales Werkzeug entsprechen, ist der Kern des entwickelten Systems als Framework realisiert worden. Nichtsdestotrotz gibt es Funktionen, die das System den Komponenten zur Verfügung stellt. Hierbei handelt es sich oftmals um Querschnittsfunktionen und Funktionen, die von verschiedenen Komponenten an unterschiedlichen Stellen verwendet werden. Diese Funktionen stehen in Bibliotheksform zur Verfügung.

4.7.2. Vererbung und Delegation

Grundsätzlich kann zwischen zwei Formen von Frameworks unterschieden werden [Hau97]: *white-box* und *black-box Frameworks*. White-box Frameworks setzen auf Vererbung als Mittel zur Anpassung/Erweiterung der Framework-Funktionalitäten. Der Benutzer bildet hier Subklassen und überschreibt eventuell zu verändernde Methoden. Anschließend werden Instanzen der neuen Klassen statt ihrer Superklassen im Framework verwendet. Die Terminologie „White-Box“ wird hier verwendet, da die Erweiterungen aufgrund der Vererbung detaillierte Kenntnis über ihre Superklassen haben.

Black-Box Frameworks hingegen setzen Objektkomposition in Form von Delegation als Mittel zur Erweiterung ein. Hier werden explizite Schnittstellen vorgegeben, welche anschließend von den Erweiterungen des Frameworks implementiert werden müssen. Im Gegensatz zum White-Box-Ansatz haben die Erweiterungen keinen detaillierten Einblick in die Framework-Klassen.

In [GHJV95] werden verschiedene Vor- und Nachteile beider Ansätze diskutiert. Auf der einen Seite ist Vererbung sehr einfach zu nutzen, da es direkt als Mittel innerhalb der Programmiersprache zur Verfügung steht. Zusätzlich erlaubt Vererbung die Änderung des Verhaltens des Frameworks, indem Methoden der Superklassen überschrieben werden. Auf der anderen Seite wird Vererbung zur Compile-Zeit definiert, wodurch ein Austausch der Funktionalität zur Laufzeit im Gegensatz zur Delegation nicht möglich ist. Daher sind Black-Box Frameworks flexibler als White-Box-Frameworks. Insgesamt stellen die Autoren von [GHJV95] sogar eine Design-Richtlinie auf, die besagt, dass Delegation der Vererbung zu bevorzugen ist.

Die Besonderheit in dieser Arbeit gegenüber allgemeinen Framework-Betrachtungen besteht darin, dass hier das Framework zunächst von mehreren Komponenten separat genutzt wird. In einem weiteren Schritt werden die Sprachen und deren Komponenten durch Glue Code kombiniert. Dadurch können bei einem White-Box-Ansatz Situationen entstehen, die eine Kombination der Komponenten erschweren oder gar verhindern. Das Hauptproblem liegt dabei in der fehlenden Mehrfachvererbung in Java: Wenn beide Teilkomponenten Subklassen einer vom Framework vorgegeben Klasse haben und sich andere Teilkomponenten der jeweiligen Sprache auf den Typ dieser Subklasse verlassen, können die Komponenten nur erschwert vereint werden.

Beispiel 4.7.1. *Abbildung 4.9 zeigt zwei Designmöglichkeiten für eine Komponente einer Symboltabelle, die Bezeichner auflösen kann (Resolver). Die linke Variante benutzt dabei Vererbung, wobei jede Sprache einen eigenen Resolver hat, der die benötigte Auflösungslogik implementiert. Andere Komponenten benutzen eine Instanz dieser Klasse, um Namen aufzulösen. Wenn beide Sprachen kombiniert werden sollen, müssen die verschiedenen Resolver kombiniert werden. Aufgrund der fehlenden Mehrfachvererbung in Java ist dies jedoch ohne weiteren Aufwand nicht möglich.*

In der rechten Variante wird hingegen die Auflösung an spezielle Resolver delegiert. Diese sind anhand der Methode `getResponsibleKind` *in der Lage, Auskunft darüber zu geben, welche Sorte von Einträgen (z.B. „Variable“ oder „Zustand“) sie auflösen können. Die Komposition der Resolver ist hier sehr einfach möglich, es müssen nur beide vorhandenen Resolver der jeweiligen Sprache registriert werden.*

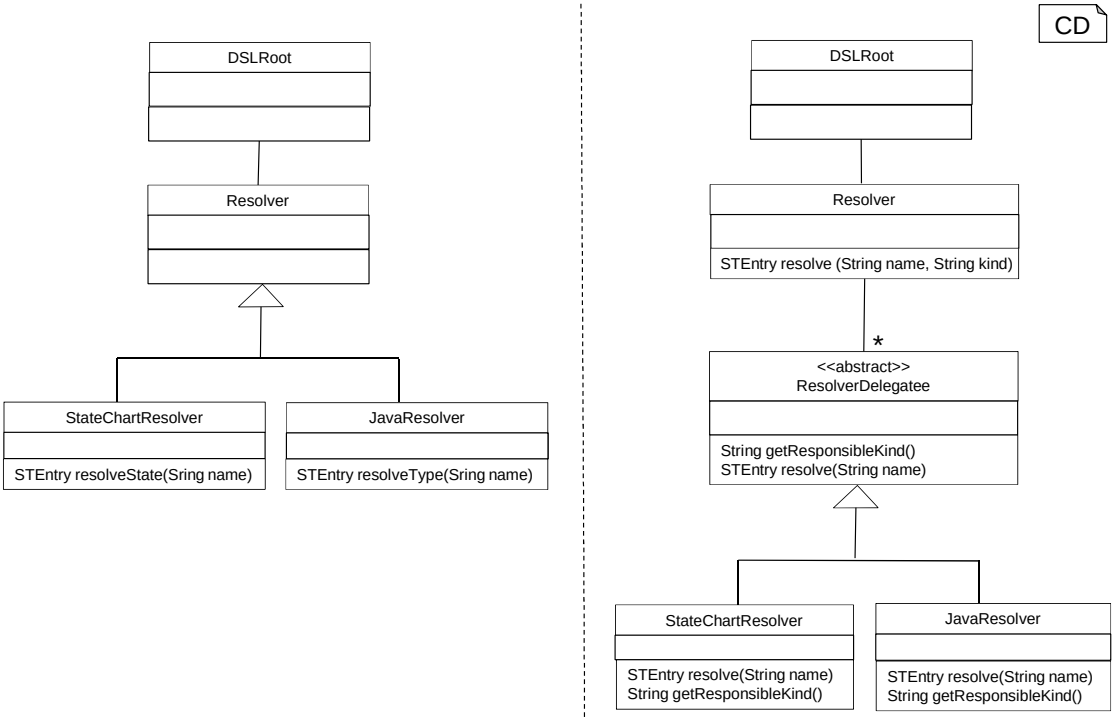


Abbildung 4.9.: Vererbung und Delegation

Aus den genannten Gründen wurde während der Entwicklung des Frameworks und insbesondere bei den Komponenten für Symboltabellen und Kontextbedingungen streng darauf geachtet, dass Vererbung nur in den passenden Situationen angewandt wird. Das Framework erfüllt daher die Black-Box-Eigenschaften.

4.7.3. Fabriken

Wie schon in [Rum04a] beschrieben, erhöhen Fabriken sowohl die Testbarkeit, als auch die Flexibilität eines Systems. Anstatt an vielen Orten im System `new`-Aufrufe zu nutzen, werden hierbei Instanzen einer Klasse grundsätzlich an einer einzigen Stelle im System erzeugt - in der Fabrik-Methode. Alle anderen Teile des Systems rufen dann die Fabrik-Methode auf, um neue Instanzen zu erhalten. Hierdurch ist es sehr einfach möglich, Subklassen des Rückgabetyps der Methode zu erstellen und nur Instanzen dieser Subklasse erzeugen zu lassen.

Auch hier treten wieder die Besonderheiten bei der Kombination mehrerer Sprachen auf: Wenn sich Komponenten unterschiedlicher Sprachen darauf verlassen, dass der Rückgabetyp der Fabrik-Methode eine bestimmte Subklasse der ursprünglichen Fabrik-Methode ist, lassen die beiden sich sehr schwer kombinieren.

Beispiel 4.7.2. *Als Namespace bezeichnet man einen Abschnitt in einem Modell/Programm, in dem Namen und zugehörige Symboltabelleneinträge gemeinsam verwaltet werden. In Java sind dies unter anderem Blöcke, Methoden- oder Klassenrümpfe. Eine Vorgehensweise zur internen Darstellung dieser Abschnitte ist das Erstellen geeigneter Klassen für jeweils einen speziellen Bereich, in unserem Beispiel also eine Klasse für den Namespace eines Blockes, eine Klasse für Methoden- und eine Klasse für Klassenrümpfe. Dabei ist es üblich, eine gemeinsame Oberklasse zu bilden, welche die hierarchische Struktur der Namespaces widerspiegelt und andere gemeinsame Eigenschaften zusammenfasst. Diese gemeinsamen Eigenschaften können jedoch sprachspezifisch sein: so können in allen Namespaces in Java Typen deklariert werden, deren Symboltabelleneinträge dann im entsprechenden Namespace registriert werden. Die gemeinsame Superklasse der Namespaces stellt dann geeignete Methoden zur Registrierung zur Verfügung.*

Die Erstellung von Instanzen kann nun wie beschrieben durch eine Fabrik im Framework übernommen werden. Sicherlich ist es sinnvoll, im sprachunabhängigen Framework eine Fabrik zur Verfügung zu stellen, die Instanzen einer standardisierten Namespace-Klasse produziert. Diese standardisierte Namespace-Klasse kann dann aber nur sprachunabhängige Eigenschaften enthalten. Wenn in einem weiteren Schritt für konkrete Sprachen Subklassen der standardisierten Namespace-Klasse erstellt werden und die Fabrik-Methode so überschrieben wird, dass nur Instanzen dieser Subklasse erzeugt werden und sich vor allem Komponenten darauf verlassen, dass diese Subklassen erzeugt werden, sind die Fabriken verschiedener Sprachen nicht mehr kompatibel zueinander. Grund hierfür besteht darin, dass jede Sprache sich auf ein spezifisches Verhalten seiner speziellen Namespace-Implementierung verlässt und zur Sprachkombination eine neue Klasse erstellt werden muss, die die verschiedenen Eigenschaften der ursprünglichen Klassen vereint. Dies ist jedoch nicht immer bzw. nur erschwert möglich.

Als exemplarisches Beispiel dient Abbildung 4.10. Unter der Annahme, dass eine Sprachkomponente für Statecharts den `SCNameSpace` und die Komponente für Java den `JavaName-`

`Space` instanziiert, ist eine einfache Umsetzung bei der Sprachkombination durch Mehrfachvererbung möglich. Diese ist jedoch in Java nicht vorhanden, sondern muss durch den Einsatz von Interfaces und Delegation umgesetzt werden. Des Weiteren können auf diese Art und Weise auch nur verschiedene Methoden in einer Klasse vereint werden. Widersprechen sich die dahinterstehenden Funktionalitäten, löst auch Mehrfachvererbung dieses Problem nicht. Daher sind Fabriken zwar ein geeignetes Mittel, um beispielsweise Testbarkeit zu unterstützen, Kompositionalität lässt sich jedoch nur bedingt durch deren Einsatz realisieren.

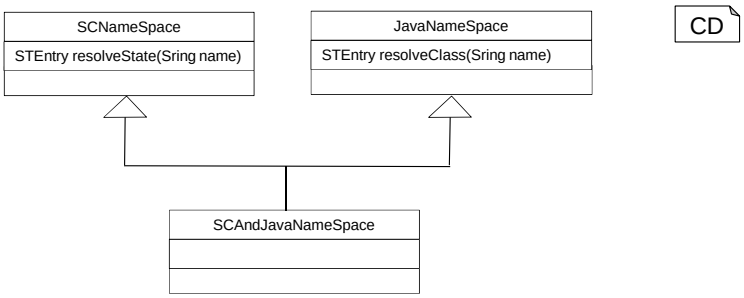


Abbildung 4.10.: Mehrfachvererbung als Lösung zur Vereinigung verschiedener Namespaces.

4.7.4. Adaptoren

Das Adapter-Pattern kann eingesetzt werden, um die Schnittstelle einer Klasse in eine andere Schnittstelle zu übersetzen und somit die Klasse in ursprünglich nicht kompatiblen Kontexten einzusetzen. Hierbei kann zwischen zwei verschiedenen Ansätzen unterschieden werden: dem Objektadapter und dem Klassenadapter. Ersterer benutzt Delegation als Mittel zur Übersetzung, zweiterer verwendet Mehrfachvererbung. Da ein Objektadapter höhere Flexibilität bietet und vor allem eine mehrfache Adaption zur Laufzeit erlaubt, wird im entwickelten Ansatz prinzipiell diese Variante angewendet.

Wichtiges Anwendungsgebiet der Adaptoren in dieser Arbeit ist die Übersetzung von Symboltabelleneinträgen. Diese Einträge stellen spezifische Informationen zu einer Entität einer Sprache dar und haben meist eine sehr einfache JavaBean-ähnliche Form, d.h. sie bestehen im Wesentlichen aus Attributen, Zugriffsmethoden und Verlinkungen untereinander und besitzen keine komplizierte Algorithmik.

Beispiel 4.7.3. *Betrachtet wird das Beispiel der Kombination von Statecharts und Java aus Abbildung 4.3. Im Wesentlichen umfassen hier Symboltabelleneinträge für Zustände im Statechart lediglich den Namen des Zustandes und - bei Verwendung von Hierarchie - Referenzen zu den untergeordneten Zuständen. Die Symboltabelleneinträge für Variablen im Javacode beinhalten hingegen neben dem Namen unter anderem den Typ der Variable.*

Der Prüfmechanismus von Java erwartet bei Auflösung der booleschen Variablen Java-Symboltabelleneinträge für Variablen, im System existieren jedoch nur die Symboltabelleneinträge für Zustände. Durch die Verwendung eines Objektadapters können die Einträge jedoch wie in Abbildung 4.11 ineinander übersetzt werden.

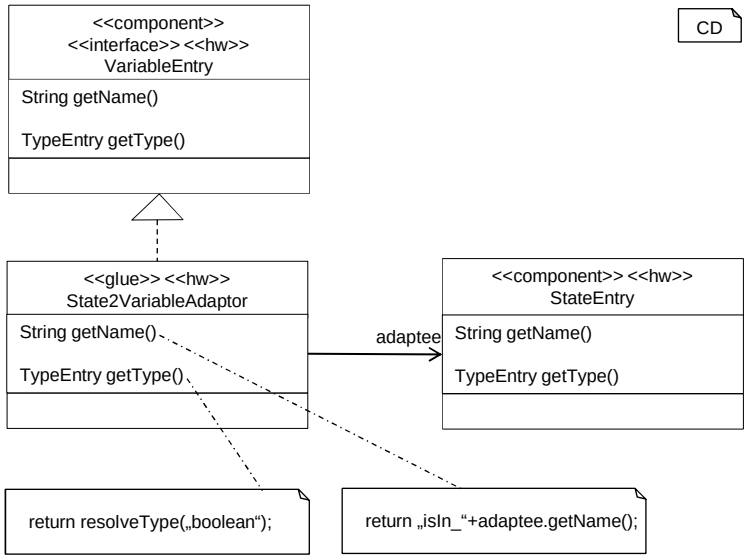


Abbildung 4.11.: Adaptoren für Symboltabelleneinträge.

4.7.5. Visitoren

Das in MontiCore eingesetzte Visitor-Pattern ist an Walkabouts [PJ98] und Runabouts [Gro03] angelehnt. Es unterstützt Kompositionalität in der Form, dass verschiedene Visitoren für unterschiedliche Sprachen zur Laufzeit ohne Neukompilierung kombiniert werden können. Dies ist vor allem bei Spracheinbettung von Vorteil, wobei hier durch die Kombination der Visitoren für die beteiligten Sprachen ein Visitor entsteht, der alle Sprachanteile bearbeiten kann.

Beispiel 4.7.4. *Ein einfaches Anwendungsgebiet der kompositionalen Visitoren sind Pretty-Printer, die in der Lage sind, einen AST wieder in Textform umzuwandeln. Bei Spracheinbettung besteht die Problematik, dass der AST aus Teilen verschiedener Sprachen zusammengesetzt ist und daher auch die beteiligten Pretty-Printer kombiniert werden müssen. Wünschenswert im kompositionalen Sinne ist dabei die eigenständige Entwicklung der Pretty-Printer mit anschließender Kombination durch Glue Code. Abbildung 4.12 zeigt die Umsetzungsmöglichkeiten dieser Anforderung durch den MontiCore-Visitor für Klassendiagramme mit eingebettetem OCL.*

4.7.6. Kompilierungspozess und Projektorganisation

Wie schon im vorigen Teilkapitel erläutert, ist es wichtig, dass die Komponenten unterschiedlicher Sprachen keine Kenntnis voneinander haben, insbesondere sollten zwischen ihnen keine Referenzen existieren. Die sprachübergreifenden Abhängigkeiten sind allein in Glue Code zu organisieren, der die eigentliche Verbindung zwischen Sprachen herstellt. Hauptgrund hierfür besteht in der Lokalisierbarkeit der Auswirkungen von Änderungen an einer Komponente: diese sind stets beschränkt auf die anderen Komponenten der gleichen Sprache und auf die Glue-Code-Anteile, welche die geänderte Sprache verwenden. Ein zweiter Grund ist Geschwindigkeit bei der Kompilierung: da die Auswirkungen auf wenige Orte beschränkt sind, müssen bei Änderun-

Java-Quellcode «hw»

```
1  public String print(ASTNode ast){
2     //the buffer to print into
3     StringBuffer buffer=new StringBuffer();
4
5     //concrete printers for both languages
6     CDPrettyPrintVisitor v1=new CDPrettyPrintVisitor();
7     OCLPrettyPrintVisitor v2=new OCLPrettyPrintVisitor();
8
9     //set target of printing
10    v1.setTarget(buffer);
11    v2.setTarget(buffer);
12
13    //instantiate a visitor and add clients
14    Visitor v = new Visitor();
15    v.addClient(v1);
16    v.addClient(v2);
17
18    //visit the ast
19    v.startVisit(ast);
20
21    //return result
22    return buffer.toString();
23 }
```

Abbildung 4.12.: Beispiel kompositionaler Pretty-Printer

gen auch nur diese neu kompiliert und auf Korrektheit geprüft werden. Dies ist insbesondere bei der Verwendung vieler Sprachen mit hohen Codeanteilen in den jeweiligen Komponenten von Bedeutung, denn die Rekompilierung aller Komponenten würde selbst bei kleinen Änderungen sehr viel Zeit beanspruchen und durch die Wartezeiten den Arbeitsfluss bei der Entwicklung stören.

Anders verhält es sich bezüglich der Abhängigkeit zum sprachunabhängigen Basisframework. Da jede Komponente einer Sprache die hier geforderten Funktionalitäten zur Verfügung stellen muss, besteht eine sehr enge Kopplung. Diese Kopplung ist jedoch nicht als problematisch anzusehen, da hierdurch keine Abhängigkeiten zwischen verschiedenen Sprachen entstehen können.

Diese Richtlinien sind sehr einfach durch Projektorganisation umsetzbar. Hierbei bieten IDEs wie Eclipse und build-Scripte wie make- oder ant-Dateien die Möglichkeit, explizit Abhängigkeiten zu definieren. Diese Abhängigkeiten finden dann im classpath des Java-Compilers Verwendung, sodass unerwünschte Querreferenzen zwischen Komponenten verschiedener Sprachen vom Compiler in Form von Fehlermeldungen berichtet werden. Insgesamt ergibt sich der vorgeschlagene Projektaufbau wie in Abbildung 4.13 dargestellt, wobei die Abhängigkeitsrelation transitiv ist.

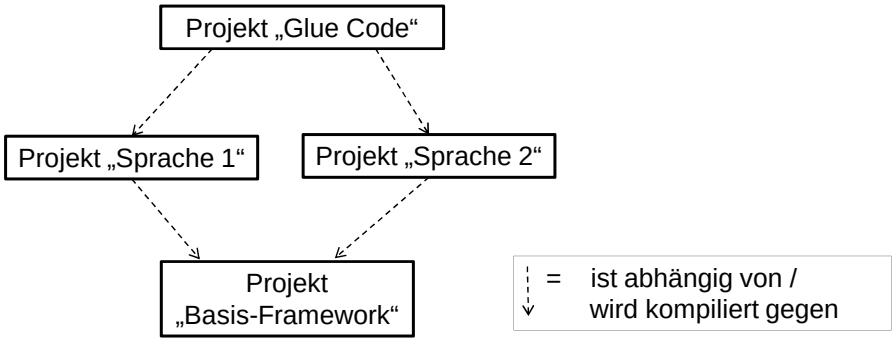


Abbildung 4.13.: Vorgeschlagene Projektabhängigkeiten am Beispiel zweier Sprachen.

4.8. Zusammenfassung

In diesem Kapitel wurde eine grundlegende Einführung in die kompositionale Sprachentwicklung gegeben. Hierzu wurden zunächst einige Beispiele vorgestellt, die aus unterschiedlichen Anwendungsgebieten entstammen. Sie umfassen dabei sowohl Beispiele aus der Programmierung als auch aus der Modellierung. Zusätzliche Beispiele aus unterschiedlichen Domänen wie der Webentwicklung oder der Entwicklung von DSLs unterstreichen die Vielfalt der Anwendungsgebiete.

Aufbauend auf den Beispielen wurden die drei verschiedenen Kompositionalitätsmechanismen Spracheinbettung, Sprachaggregation und Sprachvererbung des MontiCore-Frameworks vorgestellt. Die Vorstellung erfolgte zunächst nur aus allgemeiner Sicht und wird im Verlaufe dieser Arbeit für die unterschiedlichen Artefakte einer Sprache konkretisiert. Dabei dient die in diesem Kapitel eingeführte Architektur sprachverarbeitender Werkzeuge als Grundlage. Diese Architektur orientiert sich dabei streng an den Vorgaben kompositionaler Systeme. Zusätzlich werden die vorgestellten Design Patterns und Techniken zur modularen Entwicklung stetig Anwendung finden.

Kapitel 5.

Kompositionale Entwicklung konkreter Syntax

Die konkrete Syntax einer Sprache beschreibt die äußere Darstellungsform gegenüber dem Benutzer. Hierfür gibt es als wesentliche Ansätze die textuelle und die graphische Syntax sowie Mischformen und Ansätze, die beide Darstellungen als Alternativen nutzen. Bei der textuellen Syntax wird die Sprache meist in Form einer Grammatik definiert, graphische Syntax hingegen besteht in der Regel aus graphischen Elementen wie Rechtecke, Kreise, Ellipsen, spezielle Icons und Verbindungen in Form von Linien zwischen diesen Elementen. Zusätzlich können Farben zur Unterscheidung spezieller Elemente eingesetzt werden. Bei der Mischform wird zusätzlich zu den graphischen Elementen Text eingegeben, meist innerhalb von Textboxen oder speziellen Eingabemasken. In MontiCore und in dieser Arbeit wird die Form der textuellen Syntax verwendet.

Traditionelle Ansätze (z.B. LL(k)- oder LR(k)-Parsen) zur Erkennung von Instanzen einer Sprache verfügen über keine Kompositionalitätsmechanismen. Das Ziel dieses Kapitels besteht daher in der Untersuchung, wie die textuelle konkrete Syntax einer Sprache kompositional entwickelt und erkannt werden kann. Hierzu ist das Kapitel wie folgt aufgebaut:

- Zunächst werden die grundlegende Arbeitsweise des Erkennungsprozesses innerhalb von MontiCore und insbesondere die Erweiterungen zur Unterstützung der Kompositionalität erläutert.
- Im Anschluss erfolgt eine Diskussion, wie die korrekten sprachverarbeitenden Werkzeuge bei der Verwendung mehrerer Sprachen konfiguriert und ausgewählt werden.
- Darauf aufbauend wird diskutiert, wie die verschiedenen Szenarien der kompositionalen Sprachverarbeitung bezüglich der konkreten Syntax umgesetzt werden können.
- Abschließend erfolgt eine Vorstellung verwandter Arbeiten und eine Zusammenfassung des Kapitels.

5.1. Kompositionalität im Erkennungsprozess

5.1.1. Generelle Arbeitsweise von Lexer und Parser

Die Erkennung der textuellen Syntax einer Sprache wird standardmäßig durch zwei Komponenten übernommen: dem *Lexer* und dem *Parser*. Gemeinsam analysieren diese Komponenten

die Eingabe, wobei ihnen unterschiedliche Aufgaben zufallen: der Lexer zerlegt die Eingabe in atomare Bestandteile, während der Parser versucht, eine gültige Ableitung (siehe [ASU86]) zu errechnen. Zunächst sind zur Erläuterung des Zusammenspiels von Lexer und Parser einige Grundbegriffe von zentraler Bedeutung.

Definition 5.1.1 (Tokenklasse, Pattern und Token)**.** *Eine Tokenklasse ist eine Menge von Worten, die durch ein spezifisch für die Tokenklasse gültiges Pattern in Form eines regulären Ausdrucks beschrieben werden. Die Elemente einer Tokenklasse werden Token genannt.*

Tabelle 5.1 zeigt einige Beispiele für Token, Tokenklassen und Patterns auf.

Tokenklasse	Pattern	Token
NAME	[a-zA-Z] [a-zA-Z0-9]*	pi, Person, ...
IF	„if“	if
INTEGER	[0-9]+	13, 42, ...
REAL	[0-9]+ . [0-9]+	2.0, 0.78, ...

Tabelle 5.1.: Beispiele für Tokenklassen, Token und Pattern

Aufbauend auf den Tokenklassen ist der Erkennungsprozess wie folgt gegliedert: der erste Schritt der Erkennung der textuellen Eingabe ist das Zerlegen in Token und wird vom Lexer übernommen. Im zweiten Schritt der Erkennungsphase wird dann die vom Lexer produzierte Tokenreihenfolge analysiert und versucht, eine Ableitung der Grammatik gemäß dieser gegebenen Reihenfolge sowie eine interne Darstellung der Eingabe in Form eines *AST* (abstract syntax tree) zu bilden. Diese Aufgabe wird vom Parser übernommen. Die folgenden Definitionen fassen die Aufgaben der einzelnen Komponenten zusammen, Abbildung 5.2 zeigt die Interaktionen der Komponenten und die involvierten Artefakte.

Definition 5.1.2 (Lexer)**.** *Ein Lexer zerlegt den Zeichenstrom der Eingabe in einen Strom von Token gemäß der ihm vorliegenden Patterns, d.h. der Definitionen der Tokenklassen.*

Definition 5.1.3 (Parser)**.** *Der Parser analysiert die vom Lexer gelieferte Tokenreihenfolge und bildet eine gültige Ableitung gemäß der gegebenen Grammatik. Weiterhin bildet er eine interne Darstellung der Eingabe in Form eines abstract syntax tree (AST).*

Für das Parsen der Eingabe existieren eine Vielzahl unterschiedlicher Ansätze, wobei die bekanntesten das LL- und das LR-parsing (siehe [ASU86]) sind. Der in dieser Arbeit verwendete Ansatz wird im folgenden Teilabschnitt vorgestellt.

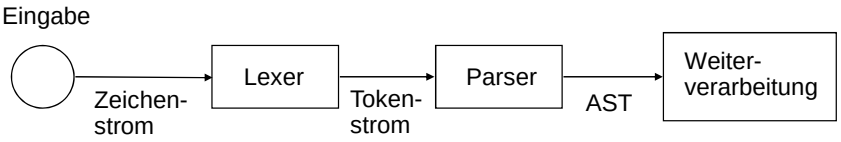


Abbildung 5.2.: Konzeptueller Zusammenhang zwischen den beteiligten Komponenten beim Parsen

5.1.2. Verwendete Technologie

In den von MontiCore bzw. ANTLR generierten Parsern wird der rekursive Abstieg (recursive descent) als spezielle Variante des LL(k)-Parsens verwendet. Hierbei wird für jedes Nichtterminal eine entsprechende Methode generiert, welche den Inhalt dieses Nichtterminals erkennt. Abbildung 5.3 skizziert die Arbeitsweise des rekursiven Abstiegs.

MontiCore-Grammatik «hw»

```
1 Automaton = "automaton" Name "{"
2       (State | Transition)* "}";
3
4 State = "state" Name;
5
6 Transition = from:Name "->"
7              to:Name;
8
9
10
11
12
13
14
15
16
17
18
19
```

Java-Quellcode «gen»

```
1  public void automaton(){
2   consume(TOKEN.AUTOMATON);
3   consume(TOKEN.NAME);
4   consume(TOKEN.LBRACKET);
5
6   while(true){
7     if (LA(1)==TOKEN.STATE){
8        state();
9     }
10    else if (LA(1)==TOKEN.NAME){
11       transition();
12    }
13    else if (LA(1)==TOKEN.RBRACK){
14       break();
15    }
16    else{ //errorhandling }
17  }
18  consume(TOKEN.RBRACKET);
19 }
```

Abbildung 5.3.: Skizze eines Recursive-Descent-Parsers

Auf der linken Seite der Abbildung befindet sich ein Auszug aus der Grammatik für Automaten und auf der rechten Seite die Methode zum Erkennen eines Automaten. Hierbei stellen die `consume`-Anweisungen (Zeilen 2-4) Aufrufe an den Lexer dar, das nächste Token zu erkennen, gegen den gegebenen Parameter zu prüfen und die aktuelle Position auf das Ende des erkannten Tokens zu setzen. Dieser Vorgang wird als *Konsumieren* des Tokens bezeichnet. In der Schleife wird anhand des nächsten Tokens geprüft, ob der folgende Text ein Zustand oder eine Transition ist und anschließend die entsprechende Methode zur Erkennung aufgerufen. Hierfür wird der LA-Aufruf (Lookahead) verwendet, der den Lexer auffordert, das nächste Token zu produzieren, die aktuelle Position in der Eingabe aber beizubehalten. Handelt es sich nicht um einen Zustand oder eine Transition, sondern um eine schließende Klammer, wird die Schleife

abgebrochen und die Klammer erkannt. Anzumerken ist hierbei, dass aus Darstellungsgründen Anteile zur Fehlerbehandlung ausgelassen wurden.

5.1.3. Entscheidungsfindung im Parse-Prozess bei Alternativen

Während des Erkennungsprozesses treten oftmals Stellen auf, an denen der Parser Entscheidungen treffen muss. Dies ist unter anderem dann der Fall, wenn Oder-Alternativen in einer Grammatik auftreten. Im vorangegangenen Beispiel der Automaten befindet sich eine solche Alternative im Produktionskörper der Automaten, wobei hier eine beliebige Folge von Zuständen oder Transitionen erkannt werden kann. Alternativ kann jedoch auch die Erkennung von Zuständen und Transitionen beendet und die abschließende Klammer erkannt werden. Somit ergeben sich drei verschiedene Optionen für den Parser im Erkennungsprozess.

Zur Entscheidungsfindung werden die First-Mengen der Alternativen verwendet [ASU86]. Hierbei handelt es sich um die ersten k Terminale (k wird hierbei als Lookahead bezeichnet), zu denen eine Alternative reduziert werden kann. Im Automaten-Beispiel reicht dabei zur Unterscheidung der Alternativen ein Lookahead von eins aus, da schon anhand des ersten Tokens die Alternativen unterschieden werden können. Diese Unterscheidung wird dann innerhalb der `if`-Anweisungen des generierten Parsers bzw. als Abbruchbedingung der `while`-Schleife verwendet.

Die Berechnung der First-Mengen kann leicht algorithmisch erfolgen [ASU86] und wird innerhalb des MontiCore-Frameworks durch ANTLR übernommen. Hierbei analysiert ANTLR die Alternativen vollautomatisch und generiert die entsprechenden Anweisungen in den entstehenden Parser. Wie jedoch bei der Diskussion der Szenarien deutlich wird, reicht die alleinige Analyse der Alternativen nicht aus. Insbesondere bei der Verwendung externer Nichtterminale kann eine Analyse nicht erfolgen, da zur Generierungszeit des Parsers nicht bekannt ist, welche Sprache eingebettet ist. Somit kann auch die First-Menge nicht berechnet werden.

MontiCore verfolgt hierfür prinzipiell die Strategie der *sicheren Einbettung*. Hierbei wird als First-Menge eines externen Nichtterminals jede mögliche Terminalkombination berechnet. Hierdurch muss der Grammatikentwickler sicherstellen, dass seine Grammatik hinreichend eindeutig ist, um sie mit jeder anderen Grammatik kombinieren zu können. Durch diese Vorgehensweise ist dem Entwickler der Sprachkombination eine Kombinierbarkeit gesichert.

Wie später in den einzelnen Szenarien erläutert wird, entstehen Situationen, in denen eine Kombinierbarkeit mit jeder beliebigen anderen Sprache jedoch gar nicht erwünscht ist. Vielmehr wird hier das Mittel der Einbettung verwendet, um die Parser/Lexer der verschiedenen Sprachen zu entkoppeln. Die eigentliche Sprachkombination ist dennoch feststehend und kann somit zur First-Mengen-Berechnung verwendet werden. Daher wurde diese Berechnung wie folgt verändert: zunächst wird geprüft, ob für eine gegebene Alternative syntaktische Prädikate existieren. Im positiven Fall wird dieses Prädikat als Grundlage zur Berechnung verwendet, andernfalls dient der Körper der Alternative zur Berechnung.

Diese Vorgehensweise hat dabei den Vorteil, dass die Anfänge eingebetteter Sprachen vom Benutzer festgelegt werden können und somit zur First-Mengen-Berechnung verwendet werden. Die Umstellung von der Berechnung aus der Alternative heraus auf die Verwendung der Prädikate ändert die First-Menge bei nicht-externen Nichtterminalen nicht, da die Prädikate typischerweise den gleichen Inhalt haben wie die Alternative selbst. Grund hierfür ist der Zweck

der Prädikate: sie sind lediglich als Mittel gedacht, um weiter als k Elemente vorauszuschauen. Ein Unterschied zwischen syntaktischem Prädikat und eigentlichem Alternativinhalt ist deshalb nicht sinnvoll.

5.1.4. Arbeitsweise von Lexer und Parser in ANTLR

Die Lexer in ANTLR verwenden eine Queue zur Speicherung bereits erkannter Token. Diese Speicherung wird vor allem durch die Lookahead-Aufrufe des Parsers veranlasst. Hierbei wird das Token noch nicht konsumiert, sondern in die Queue eingefügt und der interne Zeiger des Lexers in der Eingabe an das Ende des Tokens gesetzt. Erfolgen erneute Aufrufe seitens des Parsers an den Lexer, wird zunächst die Queue verwendet, d.h. es wird geprüft, ob sich hierin Token befinden und im positiven Falle werden diese zurückgeliefert. Andernfalls wird die Eingabe zur Produktion des nächsten Tokens verwendet und je nach Aufruf (`lookahead` oder `consume`) weiter verfahren. Hauptgrund für diese Vorgehensweise ist dabei Effizienzsteigerung, da so die Eingabe nicht mehrmals in Token zerlegt werden muss, sondern bereits erkannte Anteile zwischengespeichert werden können. Insgesamt ergibt sich der in Abbildung 5.4 dargestellte konzeptuelle Zusammenhang der beteiligten Elemente:

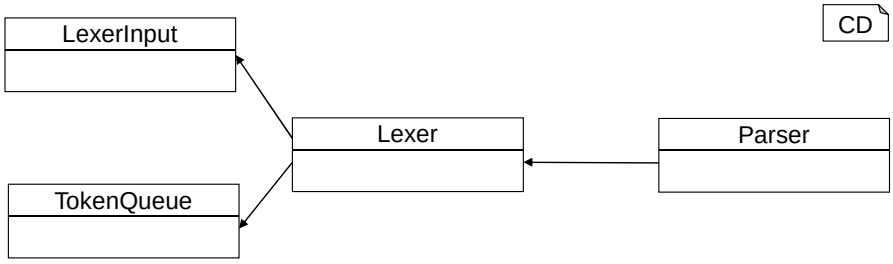


Abbildung 5.4.: Konzeptueller Zusammenhang zwischen den beteiligten Komponenten beim Parsen in ANTLR

5.1.5. Wechsel der Parser und Lexer bei Einbettung

Zur Umsetzung der Einbettung in generierten Parsern musste das Verfahren dahingehend erweitert werden, dass zunächst verschiedene Lexer Zugriff auf die Eingabe erhalten. Hierfür werden verschiedene Lexer in MontiCore prinzipiell mit demselben `LexerInput` verbunden, wodurch alle Lexer stets den gleichen aktuellen Stand der Verarbeitung nutzen. Die gemeinsame Nutzung der Queue ist jedoch nicht ohne weiteres möglich. Der Hauptgrund hierfür besteht in der Möglichkeit, in den verschiedenen Grammatiken unterschiedliche Tokenklassen zu definieren. Dementsprechend unterscheidet sich das Verhalten der Lexer, da gleiche Eingaben von ihnen zu unterschiedlichen Token verarbeitet werden. Wird der Lexer bei Spracheinbettung gewechselt und die möglicherweise gefüllte Queue wiederverwendet, würde der neue Lexer die vom alten Lexer erstellten Token zurück liefern. Die Eingabe ist in diesem Fall falsch in Token eingeteilt worden. Erst wenn die Queue komplett geleert ist, würde der neue Lexer die Einteilung übernehmen.

Um dieses Fehlverhalten zu umgehen, wurde folgender Ansatz verwendet. Erstens verwenden aus Effizienzgründen alle Lexer die gleiche Queue. Zweitens wird die Queue prinzipiell beim Wechsel der Sprache bei Einbettung geleert. Und drittens wird die Position im Eingabestrom auf das Ende des letzten konsumierten Tokens gesetzt. Speziell der dritte Punkt ist dabei von besonderer Wichtigkeit: Es ist nicht möglich, die Position auf den Beginn des ersten Tokens der Queue zu setzen. Hier spielen vor allem sogenannte *Skip*-Tokens eine Rolle: die Skip-Eigenschaft besagt dabei, dass das Token zwar vom Lexer produziert bzw. erkannt, jedoch nicht an den Parser zurückgeliefert wird. Der typische Anwendungsfall solcher Tokens sind Leerzeichen oder Tabulatoren, die aus Sicht des Parsens oftmals irrelevant sind und somit auch vom Lexer nicht in die Queue übernommen werden. Skip-Tokens können in MontiCore mit einem zusätzlichen Flag zur Markierung der Skip-Eigenschaft wie herkömmliche lexikalische Produktionen definiert werden. Das Problem besteht nun darin, dass unterschiedliche Sprachen verschiedene Skip-Tokens definieren können, insbesondere kann es vorkommen, dass die einbettende Sprache Skip-Tokens definiert, die jedoch für die eingebettete Sprache von Bedeutung sind und somit dort keine Skip-Eigenschaft haben. Befindet sich nun ein solches Token genau an Sprachgrenzen, verwirft der äußere Lexer dieses Token. Das anschließende Zurücksetzen der Queue auf das erste Element würde den inneren Lexer nicht dazu veranlassen, den Skip-Anteil der Eingabe erneut einzulesen. Dieses Verhalten führt dann zu Erkennungsfehlern. Dementsprechend muss die Eingabe zwingend auf das Ende des letzten vom äußeren Lexer konsumierte Token gesetzt werden. Um dies zu ermöglichen, wurden die Lexer so erweitert, dass sie prinzipiell die Endposition dieses letzten Tokens speichern.

Insgesamt wurde die Methodik zum Lexerwechsel, d.h. insbesondere die Techniken zum korrekten Zurücksetzen der Eingabe und zur Leerung der Queue innerhalb einer Erweiterung der ANTLR-Klassen gekapselt. Hierdurch war es möglich, die Spracheinbettung in MontiCore durch einfache Aufrufe an die API des erweiterten ANTLRs zu realisieren. Zusätzlich wurde im Rahmen der Arbeit [Kra10] die Architektur dahingehend erweitert, dass oberhalb der Parser/Lexer für eine spezifische Sprache ein übergeordneter Parser existiert, welcher sowohl die einbettenden als auch alle eingebetteten Parser/Lexer verwaltet. Insgesamt ergibt sich deshalb der in Abbildung 5.5 dargestellte Zusammenhang.

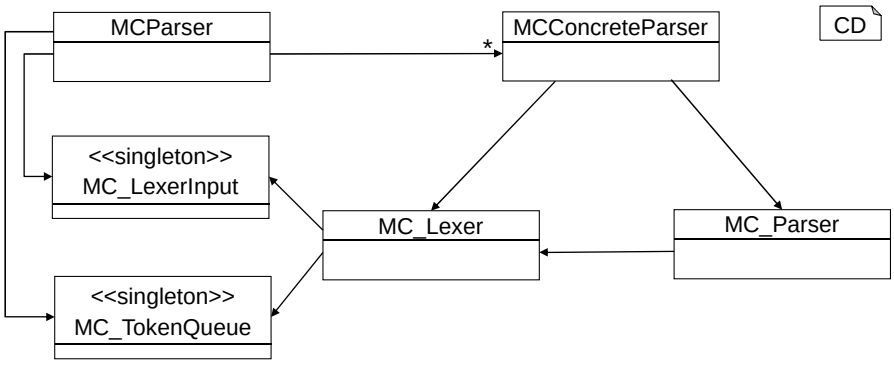


Abbildung 5.5.: Erweiterung von ANTLR und Architektur der MontiCore-Parser

Zu erkennen ist hier der ursprüngliche Zusammenhang der Komponenten aus Abbildung 5.4, wobei von den dort beteiligten Klassen jedoch vererbt wurde, um die beschriebene Funktionalität hinzuzufügen[1]. Zusätzlich existiert ein `MCParser` als Einstiegspunkt zu Instanzerkennung. Dieser verwaltet je beteiligter Sprache bei der Einbettung einen `MCConcreteParser`, welcher einerseits die von ANTLR generierten Lexer und Parser beinhaltet und andererseits einige MontiCore-spezifische Funktionalität kapselt. Für jede beteiligte Sprache wird demnach unabhängig von den anderen Sprachen ein `MCConcreteParser` generiert und bei Sprachkombination lediglich beim `MCParser` registriert. Zusätzlich ist durch die Verwendung des singleton-Stereotyps erkennbar, dass sich alle Instanzen eine gemeinsame Queue und einen gemeinsamen Eingabestrom teilen.

Beim Wechsel der Sprache wird nun wie folgt verfahren: trifft der Erkennungsprozess innerhalb eines Parsers auf ein externes Nichtterminal, ruft er zunächst die erweiterte ANTLR-API zum Zurücksetzen der Queue und der Position im Eingabestrom auf und übergibt anschließend die Kontrolle an den übergeordneten `MCParser`. Je nach Konfiguration der Einbettung wählt dieser den korrekten `MCConcreteParser` zur weiteren Erkennung der Eingabe. Da sowohl die Queue geleert als auch die Position in der Eingabe korrekt ist, kann der beteiligte Parser bzw. Lexer korrekt die Eingabe erkennen.

5.2. Konfiguration und Auswahl der Lexer und Parser

5.2.1. Konfiguration der Sprachkombination bei Einbettung

Die Konfiguration der konkreten Sprachkombination kann auf zwei Arten erfolgen: einerseits ist es möglich, einen `MCParser` (siehe Abbildung 5.5) direkt in Java zu programmieren und dort festzulegen, welche konkreten Parser an der Sprache beteiligt sind und wie die Sprachübergänge erfolgen sollen. Andererseits bietet MontiCore die Möglichkeit, Sprachdateien zu erstellen. In diesen wird festgelegt, welches externe Nichtterminal einer Grammatik durch welches Nichtterminal einer anderen Grammatik ersetzt wird. Aus dieser Sprachdatei wird dann automatisch ein `MCParser` generiert, der die gesamte benötigte Funktionalität beinhaltet. Abbildung 5.6 zeigt eine Sprachdatei zur Verwendung von Java Block-Statements als Aktionssprache in Automaten.

MontiCore-Sprachdatei «hw»

```
1 mc.automaton.AutomatonGrammar.AutomatonRule aut <<start>>;
2 mc.java.Java.BlockStatement block in aut.Action;
```

Glue

Abbildung 5.6.: Beispiel für eine Sprachdatei

In Zeile 1 wird festgelegt, welche Regel welcher Grammatik als Startregel zum Parsen verwendet wird. Hierzu erfolgt zunächst die Angabe der Regel in der Form `Paketname.Grammatikname.Regelname`. Anschließend wird der Grammatik der Kurzname `aut` zugewiesen, der im weiteren Dokument stellvertretend genutzt wird. Der Stereotyp «`start`» definiert

[1] Aus Übersichtsgründen sind die Basisklassen nicht aufgeführt. Subklassen besitzen das Präfix `MC_` zu gleichnamigen Klassen ohne Präfix.

die Regel als Startregel. In Zeile 2 wird zusätzlich festgelegt, dass ein Block-Statement der Java-Grammatik das externe Nichtterminal `Action` der Automatengrammatik ersetzt, wobei der Kurzname `aut` der Automatengrammatik zur Referenzierung wiederverwendet wird. Auch der Grammatik für Block-Statements wird ein Kurzname (`block`) zugewiesen, welcher für eine spätere Referenzierung dienen kann.

5.2.2. Auswahl der Lexer und Parser bei Sprachaggregation

In der in [GKR+06] eingesetzten Architektur für sprachverarbeitende Werkzeuge existiert für jedes Werkzeug eine Einstiegsklasse (Subklasse von `DSLTool`), in der alle von den jeweiligen Sprachen benötigten Teilkomponenten initialisiert wurden. Diese umfassten dabei unter anderem die existierenden Workflows, die zu verwendenden Parser oder auch spezifische Informationen wie Dateiendungen der Modelle einer bestimmten Sprache. Durch diese Vorgehensweise entstanden so verschiedene Subklassen des `DSLTools`, etwa eine Subklasse für Java, eine für Klassendiagramme und eine für Objektdiagramme.

Diese Architektur setzt demnach Vererbung als Mittel zur Spezialisierung/Konfigurierung einer allgemeinen Funktionalität ein. Hierdurch kommt es - wie bereits in Kapitel 4.7.2 diskutiert - zu Problemen bei der Kombination mehrerer Sprachen. Da verschiedene Werkzeuge durch Subklassenbildung parametrisiert werden, können diese Klassen in einem weiteren Schritt sehr schlecht kombiniert werden. Dies ist insbesondere dadurch bedingt, dass in jedem Werkzeug genau ein `DSLTool` als Einstiegspunkt verwendet wird.

Im Sinne der Kompositionalität wurde daher die Architektur des `DSLTool` dahingehend umgestellt, dass Delegation bzw. die Registrierung von Sprachen in einem generischem Tool anstatt des obigen Ansatzes verwendet wird. Hierbei hilft die in Abbildung 4.8 vorgestellte Composite-Architektur für Sprachen. In einem Tool wird hierbei prinzipiell eine Sprachfamilie registriert, die wiederum eine Vielzahl von Modellierungssprachen enthalten kann. Jede dieser Modellierungssprachen kennt dabei die oben angesprochenen Komponenten wie Parser und Workflows. Das generische Tool hingegen sammelt lediglich die Informationen aus allen beteiligten Sprachen zusammen und konfiguriert sich anschließend selbst anhand dieser Informationen.

Durch die in Abbildung 4.8 vorgeschlagene Architektur wird zusätzlich die Auswahl der Parser/Lexer bei Sprachaggregation unterstützt. Wie in dieser Abbildung ersichtlich ist, verfügen Sprachen auf der Ebene der `ModelingLanguage` über spezifische Informationen wie Dateiendung. Unter Nutzung dieser Information werden pro Dateiendung die in [KRV10] beschriebenen `RootFactories` erstellt. Das Framework entscheidet anschließend je nach Dateiendung des aktuellen Modells/Programms, welche der registrierten `RootFactories` beauftragt wird, eine Root zu erstellen. Diese Root enthält dann eine für die gegebene Sprache spezifische Parser/Lexer-Kombination. Durch diese Vorgehensweise wird bei Sprachaggregation stets der korrekte Parser bzw. Lexer verwendet. Zusätzlich bleiben die verwendeten Parser/Lexer interferenzfrei, d.h. die Sprachbausteine bezüglich der konkreten Syntax können sich nicht gegenseitig beeinflussen bzw. verfälschen. Kompositionalität wird zusätzlich unterstützt, da bei Sprachaggregation prinzipiell die ursprünglichen Parser/Lexer verwendet werden und auf Erstellung neuer Artefakte verzichtet wird. Hierdurch wirken sich Änderungen an den beteiligten Sprachen unmittelbar auf Systeme aus, die diese Sprache durch das Kombinationsmittel Aggregation verwenden.

5.2.3. Auswahl der Lexer und Parser bei Mehrfachvererbung

Mehrfachvererbung in der in [KRV08b] beschriebenen Form eignet sich als Mittel zur Sprachkomposition nur im Falle lexikalisch stark ähnlicher Sprachen. Hauptgrund hierfür ist, dass der vorgestellte Mechanismus geordnete Mehrfachvererbung verwendet. Bei Sprachkombination, d.h. speziell bei der Auflösung von Namen für Regeln etc., werden hierdurch die in der Vererbungsliste angegebenen Supergrammatiken von links nach rechts analysiert, wobei bei gleichnamigen Elementen die zuerst gefundenen Versionen als definierend gelten und alle anderen verworfen werden. Dies führt vor allem dann zu Problemen, wenn die beteiligten Sprachen gleichnamige Terminale, Nichtterminale etc. definieren. Die Problematik dieser Vorgehensweise wird in Abbildung 5.7 illustriert.

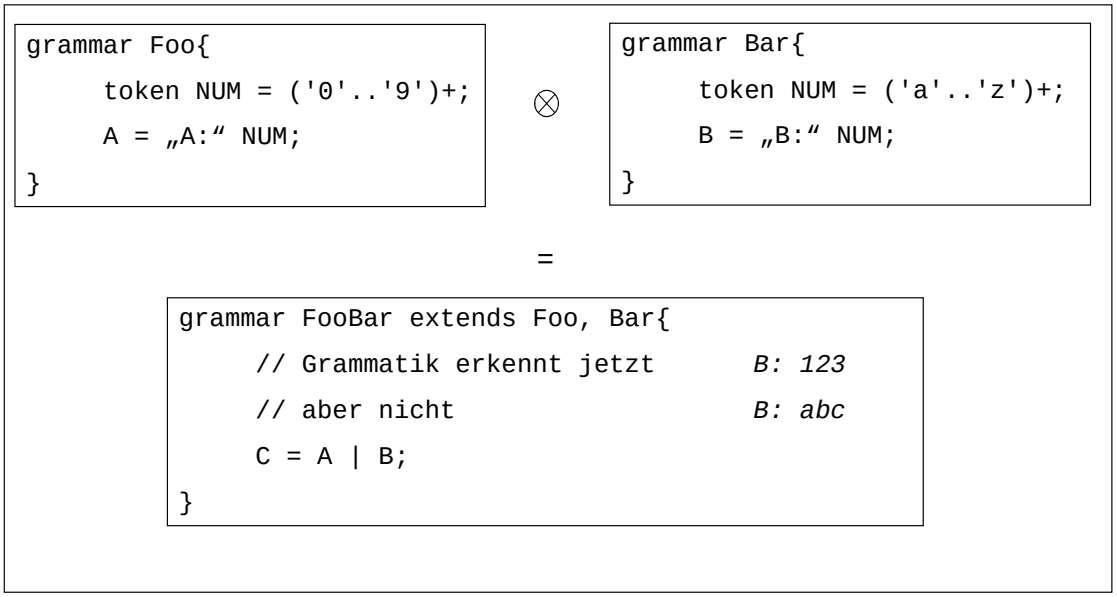


Abbildung 5.7.: Problem gleichnamiger Produktionen bei Grammatikvererbung

Durch die Verwendung des beschriebenen Ansatzes kommt es in diesem Beispiel zu der Situation, dass die durch Mehrfachvererbung entstandenen Lexer/Parser Instanzen erkennen, die weder durch die erste noch durch die zweite Grammatik definiert wurden. Von besonderer Problematik ist dieses Verhalten bei komplexer Vererbungshierarchie. Hier ist dem Benutzer nicht unmittelbar ersichtlich, welche Elemente die Kollision verursachen. Auf der anderen Seite sind Situationen wie die Diamant-Vererbung zu beachten, wobei beide Grammatiken über gleiche Elemente (die der gemeinsamen Supergrammatik) verfügen. In dieser Situation treten die beschriebenen Fehlverhalten nicht auf.

Neben der Namensgleichheit für Sprachelemente besteht eine weitere zu beachtende Randbedingung bei lexikalischen Produktionen. In ANTLR und somit auch in MontiCore müssen die Definitionen für lexikalische Produktionen stets paarweise disjunkte Sprachen ergeben, d.h. für jedes Paar durch reguläre Ausdrücke definierter lexikalischer Produktionen $l_1 \neq l_2$ muss zwingend $L(l_1) \cap L(l_2) = \emptyset$ gelten. Diese Eigenschaft kann jedoch durch den beschriebenen Ansatz

der Grammatikvererbung verletzt werden, da sich die lexikalischen Produktionen verschiedener Sprachen überschneiden können. Hierbei sind zusätzlich zu den eigentlichen lexikalischen Produktionen die Schlüsselwörter zu beachten, da diese ein separates Terminal definieren. Somit existieren fast immer Überschneidungen - beispielsweise zu Terminalen, die Namen definieren. Die Auswirkungen dieser Schlüsselwörter sind dabei jedoch nur bedingt problematisch zu sehen, da lediglich die einzelnen Wörter und nicht komplexe Tokenklassen mit einer potentiell unendlichen Anzahl von Wörtern Überschneidungen liefern.

Zusammenfassend lassen sich folgende Randbedingungen formulieren, unter denen die Mehrfachvererbung zu gewünschten Ergebnissen führt:

1. Die Namen von Nichtterminalen, Interfaces, Terminalen und externen Nichtterminalen müssen verschieden sein.

2. Für je zwei lexikalische Produktionen $l_1 \neq l_2$ muss $L(l_1) \cap L(l_2) = \emptyset$ gelten. Dies muss für alle lexikalischen Produktionen aller beteiligten Grammatiken gelten, insbesondere auch die der Supergrammatiken etc.

Durch Refactoring der beteiligten Grammatiken kann insbesondere der erste Punkt gelöst werden. Hierbei werden gleichnamige Elemente umbenannt, etwa durch Voranstellen des Namens der zugehörigen Grammatik als Präfix. Anzumerken ist hier jedoch, dass die Umbenennung mitunter nicht erwünscht oder im Extremfall - wenn die Grammatiken nicht vorliegen - nicht durchführbar ist. Des Weiteren widerspricht eine solche Vorgehensweise den Zielen der Kompositionalität, da hier verlangt wird, dass die Eingabeartefakte ungeändert übernommen werden. Der zweite Punkt hingegen ist durch Refactoring nicht lösbar, da die Änderung des Inhaltes lexikalischer Produktionen zwangsläufig zur Änderung der Sprache führt.

Dennoch eignet sich Mehrfachvererbung als Mittel zur Sprachkomposition, wenn die gegebenen Randbedingungen erfüllt sind. Insbesondere die Verschiedenheit lexikalischer Produktionen ist bei neu entwickelten DSLs leicht sicherzustellen, indem Grammatiken auf die von MontiCore standardmäßig vorgegebenen Produktionen (z.B. für Namen oder Strings) zurückgegriffen wird. Bei der Erstellung einer Grammatik für existierende Sprachen (z.B. SQL oder Java) müssen hingegen die von der jeweiligen Sprache vorgegebenen Definitionen verwendet werden. In diesen Fällen muss auf die Mehrfachvererbung verzichtet und andere Techniken zur Umsetzung des gegebenen Szenarios verwendet werden. Diese Techniken werden in den Diskussionen zu den einzelnen Szenarien vorgestellt.

5.3. Sprachvereinigung

Die Vereinigung vormals selbständiger Sprachen in ein Gesamtwerkzeug erfolgt über die Mechanismen der Sprachfamilienbildung wie in Abbildung 4.8 dargestellt. Jede Sprache ist dabei eine Instanz der Klasse `ModelingLanguage` und kann durch einen Aufruf der entsprechenden `add`-Methode der Sprachfamilie hinzugefügt werden. Das Hinzufügen mehrerer Sprachen ist durch das Hinzufügen einer `LanguageFamily` möglich. Angewandt auf die Sprachfamilie UML ergibt sich dabei der in Abbildung 5.8 dargestellte Sourcecode.

Java-Quellcode «hw»

Glue

```
1 LanguageFamily family=new LanguageFamily();
2 family.add(JavaLanguageFactory.createLanguage());
3 family.add(UMLFamilyFactory.createFamily());
```

Abbildung 5.8.: Beispiel für die Vereinigung von Sprachen

In dieser Abbildung sind verschiedene Merkmale und Besonderheiten der Sprachvereinigung zu erkennen. Zum einen können einer Familie nicht nur einzelne Sprachen hinzugefügt werden (Zeile 2), sondern auch Sprachfamilien (Zeile 3). Diese Vorgehensweise erlaubt nicht nur die Wiederverwendung einzelner Sprachen, vielmehr wird auch die Wiederverwendung sinnvoller Sprachkombinationen ermöglicht, da nicht nur einzelne Sprachen, sondern auch Sprachkombinationen bibliotheksartig abgelegt werden können. Zum anderen werden sowohl die Sprachen als auch die Familien durch den Einsatz von Fabriken erzeugt. Dadurch ergibt sich der Vorteil, dass die eigentliche Erzeugung an einer einzelnen Stelle erfolgt und somit leicht änderbar ist. Vor allem bei eingebundenen Sprachfamilien wirken sich so Änderungen direkt auf einbindende Familien aus. Im Beispiel kann die UML durch weitere Sprachen ergänzt werden, ohne dass der gegebene Code verändert werden muss. Die Einbindung einer neuen Sprache in die UML wirkt sich stattdessen direkt und automatisch auf die dargestellte Sprachfamilie aus. Insgesamt ergibt sich die in Tabelle 5.9 dargestellte Lösung.

Szenario	Sprachvereinigung
Name der Lösung	Sprachfamilienbildung
Ausgangslage	• Zwei oder mehrere existierende Sprachen (`ModelingLanguage`) oder Sprachfamilien (`LanguageFamily`).
Lösungsstrategie	• Instanziieren einer neuen Sprachfamilie innerhalb einer Fabrik • Hinzufügen der existierenden Sprachen/Sprachfamilien
Anmerkungen	Keine Überlappungen bezüglich der konkreten Syntax.

Tabelle 5.9.: Lösungsansatz für Sprachvereinigung

Auswahl der Lexer und Parser

Wie bereits zu Beginn dieses Kapitels beschrieben, wählt MontiCore die für ein gegebenes Modell/Programm zuständige `RootFactory` anhand der Dateiendung aus. Um diesen Mechanismus zu unterstützen, verfügen `ModelingLanguages` über zwei wesentliche Attribute bezüglich der konkreten Syntax. Zum einen liefert jede Sprache die ihr zugehörige Dateiendung mit und zum anderen die `RootFactory` zum Erstellen neuer Roots. Beim Hinzufügen mehrerer

Sprachen zu einem Gesamtwerkzeug werden diese Informationen gesammelt, um anschließend effizient die richtige `RootFactory` auswählen zu können. Im Falle gleicher Dateiendungen werden entsprechende Fehlermeldungen bereits bei der Initialisierung ausgegeben.

Registrierung der Lexer und Parser

Die für eine Sprache aus einer Grammatik generierten Parser und Lexer werden nach der Instanziierung der Root direkt als Attribute der Root gesetzt. Zusätzlich wird für jede Sprache ein Workflow generiert, der den entsprechenden Parser/Lexer aus der Root ausliest und den parse-Prozess startet. Da sowohl die `RootFactory` als auch die Root und der Workflow generiert werden, ist keine Interaktion seitens des Benutzers notwendig.

5.4. Nichtterminal-Erweiterung

Ziel dieses Szenarios ist die Erweiterung eines Nichtterminals einer Sprache um ein Nichtterminal einer weiteren Sprache. Hierfür eignen sich unterschiedliche Vorgehensweisen, die im Folgenden vorgestellt werden. Als exemplarisches Beispiel dient dabei die Verbindung von Java und SQL gemäß [KRV10]. Hierbei besteht das Ziel darin, Java-Expressions um SQL-Select-Statements zu erweitern um somit SQL direkt in Java verwenden zu können. Vorausgesetzt wird demnach eine Java-Grammatik mit dem Nichtterminal `Expression` und eine SQL-Grammatik mit dem Nichtterminal `SQLSelect`.

5.4.1. Umsetzung durch Mehrfachvererbung

Die Umsetzung dieses Szenarios durch Mehrfachvererbung erfolgt unter Beachtung der in 5.2.3 beschriebenen Randbedingungen. Die beteiligten Grammatiken müssen demnach verschiedennamige Produktionen, Interfaces, Terminale etc. aufweisen. Zusätzlich müssen alle Terminale paarweise disjunkte Sprachen definieren. Unter diesen Randbedingungen kann die Erweiterung der Java-Grammatik wie in Abbildung 5.10 erfolgen.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javasql;
2
3 grammar JavaSQL extends mc.Java, mc.SQL{
4
5   SQLSelect implements Expression;
6
7 }
```

Abbildung 5.10.: Erweiterung von Expressions um SQL-Select-Statements

In dieser Abbildung erfolgt zunächst eine Mehrfachvererbung zu den gegebenen Grammatiken Java und SQL (Zeile 3). Anschließend wird in Zeile 5 definiert, dass das Nichtterminal `SQLSelect` das Interface `Expression` der Java-Grammatik implementiert. Demnach kann

an jeder Stelle, in der ein Java-Ausdruck erkannt werden kann, auch ein SQL-Select-Statement stehen. Hiermit ist zunächst das Ziel der Nichtterminal-Erweiterung erreicht, jedoch sind einige Randbedingungen zu erläutern.

Wie im Beispiel aufgezeigt, besteht eine implements-Beziehung zwischen den beteiligten Nichtterminalen. Hieraus wird ersichtlich, dass `Expression` in der Java-Grammatik als Interface definiert wurde. Alternativ könnte hier auch eine `extends`-Beziehung erfolgen, falls das Nichtterminal `Expression` eine Klassenproduktion wäre. Aus Sicht der konkreten Syntax würden sich dadurch keine Auswirkungen zeigen, aus Sicht der abstrakten Syntax und speziell aus Sicht der in MontiCore gewählten Java-Codegenerierung ist die `extends`-Beziehung hier jedoch nicht möglich[2]. Daher muss für diesen Fall eine geänderte Version der Subgrammatik gemäß Abbildung 5.11 verwendet werden.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javasql;
2
3 grammar JavaSQL extends mc.Java, mc.SQL{
4
5    SQLExpression extends Expression = SQLSelect;
6
7 }
```

Abbildung 5.11.: Erweiterung von Expressions um SQL-Select-Statements im Falle einer Klassenproduktion

In dieser Abbildung wird eine neues Nichtterminal `SQLExpression` eingeführt, welches in einer `extends`-Beziehung zu `Expression` steht. In diesem Falle handelt es sich bei `Expression` demnach nicht um eine Schnittstellenproduktion, sondern um eine Klassenproduktion. Zusätzlich wird `SQLExpression` zu `SQLSelect` der SQL-Grammatik abgeleitet, wodurch die Verwendung einer SQL-Select-Anweisung innerhalb von Java ermöglicht wird.

Eine zentrale Fragestellung für beide Umsetzungen besteht in der Entscheidung für eine bestimmte Alternative der `Expression` durch den Parser. In Java existiert dabei eine Vielzahl von Alternativen: von einfachen Namen über Prä-/Postfix-Operatoren bis hin zu binären Operationen wie der Addition. Zur Unterscheidung wird entweder der gegebene Lookahead verwendet oder es werden syntaktische Prädikate angegeben. Im gegebenen Beispiel wurde nun eine zusätzliche Alternative für Expressions hinzugefügt, welche vom Parse-Mechanismus in die Entscheidungsfindung mit einbezogen werden muss. Im einfachen Fall ist dabei der gegebene Lookahead ausreichend oder ein größerer Lookahead ermöglicht eine Auswahl der Alternative. Diese Erhöhung des Lookaheads kann im Kopf der Grammatik durch Angabe des entsprechenden k erfolgen. Falls jedoch kein k für die zweifelsfreie Unterscheidung ausreichend ist, müssen syntaktische Prädikate wie in Abbildung 5.12 eingesetzt werden.

Hier wird im Unterschied zur vorangegangenen Version das syntaktische Prädikat in Zeile 5 definiert. Dieses Prädikat besagt dabei, dass ein SQL-Select prinzipiell dem Aufbau `SELECT`,

[2] Der Umgang mit dem Überschreiben von Regeln und der Regelvererbung bezüglich der abstrakten Syntax wird in Abschnitt 6.1 detailliert erläutert.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javasql;
2
3 grammar JavaSQL extends mc.Java, mc.SQL{
4
5   SQLSelect implements ("SELECT" NAME ("," NAME)* "FROM")=> Expression;
6
7 }
```

Abbildung 5.12.: Angabe von syntaktischen Prädikaten in JavaSQL

einer Komma-separierten Liste von Spaltennamen gefolgt von `FROM` entspricht. Dieses syntaktische Prädikat wird anschließend vom generierten Parser zur Entscheidungsfindung für die Alternativen der Expression verwendet. Bei der Verwendung der `extends`-Klausel wird das syntaktische Prädikat in gleicher Weise angegeben.

Wie bereits angemerkt, kann durch die Mehrfachvererbung das Ziel der Nichtterminal-Erweiterung umgesetzt werden. Im gegebenen Beispiel wurden Java-Expressions um SQL-Statements erweitert. Dies erfolgte unter der Randbedingung der verschiedennamigen Grammatikelemente und unter Beachtung der Terminalproduktionen. Speziell die zweite Bedingung zeigt dabei eine Auswirkung dieses Ansatzes: die Schlüsselwörter einer der beteiligten Grammatiken können nicht mehr als Namen verwendet werden. Somit kann beispielsweise keine Variable mit dem Namen `SELECT` definiert werden. Die Auswirkungen sind demnach vergleichbar mit der Einführung des `assert`-Statements in Java 1.4. Insgesamt ergibt sich die Lösung durch Mehrfachvererbung daher wie in den Tabellen 5.13 und 5.14 dargestellt.

5.4.2. Umsetzung durch Einbettung

Die Kombination aus einfacher Vererbung und Einbettung als Alternative zur Mehrfachvererbung wurde bereits in [Kra10] erwähnt. Aufgrund des unterschiedlichen Fokus von [Kra10] erfolgte eine ausführliche Beschreibung der Vorgehensweise, der Problematiken und zugehörigen Lösungsstrategien jedoch nicht. Daher wird das Konzept an dieser Stelle detailliert vorgestellt und diskutiert.

Externes Nichtterminal als zusätzliche Alternative

Eine Möglichkeit zur Umsetzung der Nichtterminal-Erweiterung ist das Hinzufügen einer Alternative zu einem gegebenen Nichtterminal, wobei die Alternative zu einem externen Nichtterminal abgeleitet wird. Abbildung 5.15 zeigt den beschriebenen Ansatz.

In dieser Abbildung erfolgt zunächst eine Vererbung zu Java in Zeile 3 und eine zusätzliche Einführung des externen Nichtterminals `Extension` in Zeile 5. Anschließend wird in Zeile 7 das Nichtterminal `Expression` unter Hinzunahme einer Alternative redefiniert. Hierdurch wird das Nichtterminal `Expression` erweitert und kann durch ein SQL-Select-Statement ersetzt werden.

Szenario	Nichtterminal-Erweiterung
Name der Lösung	Mehrfachvererbung und Interface-Erweiterung
Ausgangslage	• Zwei existierende Grammatiken G_1 und G_2 • Zu erweiterndes Nichtterminal N_1 in G_1 als Schnittstellenproduktion • Erweiterndes Nichtterminal N_2 in G_2
Lösungsstrategie	• Erstellen einer neuen Grammatik G_3 mit Vererbungsbeziehung zu G_1 und G_2 • Erstellung einer `implements`-Beziehung zwischen N_2 und N_1 unter Hinzunahme syntaktischer Prädikate
Anmerkungen	Randbedingungen sind die Verschiedenheit für Namen der Elemente (Nichtterminale, Terminale etc.) und die Disjunktheit lexikalischer Produktionen. Es entstehen Überlappungen bezüglich der Schlüsselwörter.

Tabelle 5.13.: Lösungsansatz durch Mehrfachvererbung für Nichtterminal-Erweiterung bei Schnittstellenproduktionen

Wie in dieser Abbildung ersichtlich ist, besteht keine Beziehung zur SQL-Grammatik. Vielmehr ist es möglich, eine beliebige Sprache an das externe Nichtterminal zu binden. Durch die Beliebigkeit der eingesetzten Sprache bei der Einbettung entstehen jedoch Randbedingungen, insbesondere in Bezug auf die Erkennung der Instanzen durch den Parser. Dabei spielt die bereits in Abschnitt 5.1.3 erörterte sichere Einbettung eine Schlüsselrolle: da eine Alternative zu einem externen Nichtterminal reduziert wird, wird die First-Menge dieses Nichtterminal zu jeder beliebigen Terminalabfolge berechnet. Der Prüfmechanismus verwirft die Grammatik als fehlerhaft bezüglich der LL(k)-Eigenschaft.

Zur Lösung dieses Problems existieren zwei Strategien. In der ersten Strategie werden die Anfänge der Alternativen disjunkt gemacht, indem beispielsweise der Alternative ein Zeichen vorangesetzt wird, mit welchem keine andere Alternative beginnen kann. Hierdurch kann der Parser bereits beim ersten Token entscheiden, ob der Zweig des externen Nichtterminals valide ist. Diese Vorgehensweise ist jedoch nicht immer wünschenswert, da sich die erkannte Sprache verändert. In der zweiten Strategie werden explizite syntaktische Prädikate zur Unterstützung des Entscheidungsprozesses durch den Parser definiert.

Durch die Umstellung der First-Mengen-Berechnung auf syntaktische Prädikate kann die zweite Strategie den Anfang der eingebetteten Sprache innerhalb eines Prädikats verwenden. Abbildung 5.16 zeigt den Ansatz und insbesondere die Verwendung des syntaktischen Prädikates in Zeile 8.

Der wesentliche Unterschied zwischen beiden Strategien ist die Kopplung der Sprachen: in der ersten Alternative wird lediglich ein Zeichen eingeführt, welches die Entscheidung ermög-

Szenario	Nichtterminal-Erweiterung
Name der Lösung	Mehrfachvererbung und Regelerweiterung
Ausgangslage	• Zwei existierende Grammatiken G_1 und G_2 • Zu erweiterndes Nichtterminal N_1 in G_1 als Klassenproduktion • Erweiterndes Nichtterminal N_2 in G_2
Lösungsstrategie	• Erstellen einer neuen Grammatik G_3 mit Vererbungsbeziehung zu G_1 und G_2 • Erstellung eines neuen Nichtterminals N_3 mit `extends`-Beziehung zu N_1 • Ableitung von N_3 zu N_2 unter Hinzunahme syntaktischer Prädikate
Anmerkungen	Randbedingungen sind die Verschiedenheit für Namen der Elemente (Nichtterminale, Terminale etc.) und die Disjunktheit lexikalischer Produktionen. Es entstehen Überlappungen bezüglich der Schlüsselwörter.

Tabelle 5.14.: Lösungsansatz durch Mehrfachvererbung für Nichtterminal-Erweiterung

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javasql;
2
3 grammar JavaSQL extends mc.Java{
4
5   external Extension;
6
7   Expression = ...  //Wiederholung alte Definition
8                | Extension;
9 }
```

Abbildung 5.15.: Umsetzung durch externes Nichtterminal als Alternative

licht. Dieses Zeichen ist nicht abhängig von der eingebetteten Sprache, verändert aber die resultierende Sprache. In der zweiten Strategie wird durch das syntaktische Prädikat die Sprache nicht verändert. Dafür wird jedoch festgelegt, dass eingebettete Sprachen mit diesem Prädikat beginnen müssen. Diese Einschränkung ist jedoch für das Szenario der Nichtterminal-Erweiterung unbedeutend, da das Ziel darin bestand, zwei Sprachen fest miteinander zu koppeln. Insgesamt ergibt sich die Lösung wie in Tabelle 5.17 dargestellt.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javasql;
2
3 grammar JavaSQL extends mc.Java{
4
5   external Extension;
6
7   Expression = ...  //Wiederholung alte Definition
8                | ("SELECT" NAME ("," NAME)* "FROM")=> Extension;
9 }
```

Abbildung 5.16.: Verwendung eines syntaktischen Prädikats zur Alternativwahl

Szenario	Nichtterminal-Erweiterung
Name der Lösung	Externes Nichtterminal als zusätzliche Alternative
Ausgangslage	• Zwei existierende Grammatiken G_1 und G_2 • Zu erweiterndes Nichtterminal N_1 in G_1 • Erweiterndes Nichtterminal N_2 in G_2
Lösungsstrategie	• Erstellen einer neuen Grammatik G_3 mit Vererbungsbeziehung zu G_1 • Erstellung eines externen Nichtterminals E in G_3 und Wiederholung der Definition von N_1 unter Hinzunahme einer Alternative für E • Die First-Menge der Alternative muss entweder durch Voranstellen eines Zeichens oder durch die Verwendung eines syntaktischen Prädikates disjunkt von den First-Mengen der anderen Alternativen gemacht werden • Die Sprachkombination erfolgt durch die Sprachdateien gemäß Abschnitt 5.2.1
Anmerkungen	Bei der Verwendung eines zusätzlichen Zeichens wird die Sprache verändert, es kann jedoch jede beliebige Sprache eingesetzt werden. Bei der Verwendung von Prädikaten erfolgt keine Sprachveränderung, der Anfang der eingebetteten Sprachen ist jedoch festgelegt.

Tabelle 5.17.: Lösungsansatz durch externe Nichtterminale als zusätzliche Alternative

Einführung zusätzlicher Regelvererbung

Aus Sicht der konkreten Syntax ist die Einführung einer zusätzlichen Alternative gleichbedeutend mit einer Vererbung vom ursprünglichen Nichtterminal. Abbildung 5.18 zeigt diese Form.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javasql;
2
3 grammar JavaSQL extends mc.Java{
4
5   external Extension;
6
7   SQLExpression extends Expression = Extension;
8 }
```

Abbildung 5.18.: Umsetzung durch Regelvererbung

In dieser Abbildung wird im Gegensatz zu den vorigen Versionen ein zusätzliches Nichtterminal `SQLExpression` eingeführt, welches von der ursprünglichen `Expression` erbt[3]. Wie bereits beim Hinzufügen einer zusätzlichen Alternative im vorigen Teilkapitel, ist es jetzt möglich, eine beliebige Sprache einzubetten. Dadurch entstehen jedoch auch die gleichen Effekte: entweder müssen die First-Mengen der Alternativen durch Angabe eines speziellen Schlüsselwortes disjunkt gemacht werden oder es muss ein syntaktisches Prädikat angegeben werden. Erstere Variante verändert wiederum die Sprache, während die zweite Variante die Sprachen fest aneinander bindet, was allerdings im gegebenen Szenario erwünscht ist. Daher empfiehlt sich die zweite Variante wie in Abbildung 5.19 dargestellt, insgesamt ergibt sich die in Tabelle 5.20 dargestellte Lösung.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javasql;
2
3 grammar JavaSQL extends mc.Java{
4
5   external Extension;
6
7   SQLExpression extends
8             ("SELECT" NAME ("," NAME)* "FROM")=> Expression = Extension;
9 }
```

Abbildung 5.19.: Umsetzung durch Regelvererbung und syntaktisches Prädikat

5.4.3. Vergleich der Lösungen

Die vorgeschlagenen Lösungsmöglichkeiten unterscheiden sich bezüglich der Umsetzung und eventueller Auswirkungen bzw. Randbedingungen. Tabelle 5.21 fasst die wesentlichen Eigenschaften der Lösungsmöglichkeiten zusammen.

[3] Hierbei wird davon ausgegangen, dass `Expression` eine Klassenproduktion ist. Falls `Expression` ein Interface ist, müsste die implements-Beziehung verwendet werden.

Szenario	Nichtterminal-Erweiterung
Name der Lösung	Umsetzung durch Regelvererbung
Ausgangslage	• Zwei existierende Grammatiken G_1 und G_2 • Zu erweiterndes Nichtterminal N_1 in G_1 • Erweiterndes Nichtterminal N_2 in G_2
Lösungsstrategie	• Erstellen einer neuen Grammatik G_3 mit Vererbungsbeziehung zu G_1 • Erstellung eines externen Nichtterminals E und eines Nichtterminals N in G_3 • Einführung einer Regelvererbung zwischen N und N_1 und Ableitung von N zu E • Die First-Menge von N muss entweder durch Voranstellen eines Zeichens vor E oder durch die Verwendung eines syntaktischen Prädikates disjunkt von den First-Mengen der anderen Alternativen gemacht werden • Die Sprachkombination erfolgt durch die Sprachdateien gemäß Abschnitt 5.2.1.
Anmerkungen	Bei der Verwendung eines zusätzlichen Zeichens wird die Sprache verändert, es kann jedoch jede beliebige Sprache eingesetzt werden. Bei der Verwendung von Prädikaten erfolgt keine Sprachveränderung, der Anfang der eingebetteten Sprachen ist jedoch festgelegt.

Tabelle 5.20.: Lösungsansatz durch Regelvererbung

5.5. Nichtterminal-Ersetzung

Ziel dieses Szenarios ist die Ersetzung eines Nichtterminals einer Sprache durch ein Nichtterminal einer weiteren Sprache. Als Anwendungsfall dient hierfür eine Automatensprache mit C++ Anteilen als Aktionssprache, wobei dieser C++ Anteil durch Java-Blockstatements ersetzt werden soll. Wie bereits im vorangegangenen Szenario existieren auch hier verschiedene Lösungsstrategien, die im Folgenden vorgestellt werden.

5.5.1. Umsetzung durch Mehrfachvererbung

Prinzipiell kann das gegebene Szenario durch Mehrfachvererbung der Grammatiken umgesetzt werden. Hierbei wird eine neue Grammatik erstellt, die von den gegebenen Grammatiken erbt

Lösungsansatz	Mehrfachvererbung und Interface-Erweiterung gemäß Tabelle 5.13
Anmerkungen	Randbedingungen sind die Verschiedenheit für Namen der Elemente (Nichtterminale, Terminale etc.) und die Disjunktheit lexikalischer Produktionen. Es entstehen Überlappungen bezüglich der Schlüsselwörter.
Lösungsansatz	Mehrfachvererbung und Regelerweiterung gemäß Tabelle 5.14
Anmerkungen	Randbedingungen sind die Verschiedenheit für Namen der Elemente (Nichtterminale, Terminale etc.) und die Disjunktheit lexikalischer Produktionen. Es entstehen Überlappungen bezüglich der Schlüsselwörter.
Lösungsansatz	Externe Nichtterminale als zusätzliche Alternative gemäß Tabelle 5.17
Anmerkungen	Bei der Verwendung eines zusätzlichen Zeichens wird die Sprache verändert, es kann jedoch jede beliebige Sprache eingesetzt werden. Bei der Verwendung von Prädikaten erfolgt keine Sprachveränderung, der Anfang der eingebetteten Sprachen ist jedoch festgelegt.
Lösungsansatz	Umsetzung durch Regelvererbung gemäß Tabelle 5.20
Anmerkungen	Bei der Verwendung eines zusätzlichen Zeichens wird die Sprache verändert, es kann jedoch jede beliebige Sprache eingesetzt werden. Bei der Verwendung von Prädikaten erfolgt keine Sprachveränderung, der Anfang der eingebetteten Sprachen ist jedoch festgelegt.

Tabelle 5.21.: Lösungsansätze für Nichtterminal-Erweiterung

und das existierende Nichtterminal für Aktionen der Automaten überschreibt. Anzumerken ist, dass die in Abschnitt 5.2.3 erläuterten Randbedingungen für Mehrfachvererbung gelten müssen, da im anderen Falle die Umsetzung durch Mehrfachvererbung zu ungewünschten Ergebnissen führt. Abbildung 5.22 zeigt den Ansatz.

In dieser Abbildung wird zunächst in Zeile 3 eine Mehrfachvererbung zu den gegebenen Grammatiken eingeführt. Anschließend wird in Zeile 5 das existierende Nichtterminal `Action` der Automatensprache überschrieben, indem eine gleichnamige Regel definiert wird. Diese Regel wird auf das Nichtterminal `JavaBlockStatement` der Java-Grammatik abgeleitet, wodurch Java die einzige Aktionssprache darstellt.

Durch die Verwendung der Mehrfachvererbung entsteht die bereits erläuterte Schlüsselwort-Problematik. Schlüsselwörter einer Sprache sind in der kombinierten Sprache nicht mehr Elemente anderer Tokenklassen. Dadurch kann beispielsweise kein Automat mit dem Namen `class` definiert werden. Zusätzlich zu den beschriebenen Effekten können Probleme aufgrund der LL(k)-Eigenschaft der Grammatiken entstehen. Da diese Probleme allen Lösungsstrategien ge-

MontiCore-Grammatik «hw»

Glue

```
1 package mc.automatonWithJava;
2
3 grammar AutomatonWithJava extends mc.Automaton, mc.Java{
4
5   Action = JavaBlockStatement;
6
7 }
```

Abbildung 5.22.: Umsetzung durch Mehrfachvererbung

mein sind, werden diese separat in Abschnitt 5.5.3 vorgestellt.

5.5.2. Umsetzung durch Einbettung

Bei der Umsetzung durch Einbettung wird zunächst eine Subgrammatik der Automaten erstellt und anschließend wie in der vorangegangenen Version das Nichtterminal der Aktion überschrieben. Zusätzlich wird ein externes Nichtterminal eingeführt, auf welches das Nichtterminal für die Aktion reduziert wird. Abbildung 5.23 zeigt die Vorgehensweise.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.automatonWithJava;
2
3 grammar AutomatonWithJava extends mc.Automaton{
4
5   external Extension;
6
7   Action = Extension;
8
9 }
```

Abbildung 5.23.: Umsetzung durch Einbettung

Wie schon im Szenario der Nichtterminal-Erweiterung besteht auch hier keine Verbindung zu Java, vielmehr wird diese Verbindung zur Konfigurationszeit des Werkzeugs hergestellt. Dadurch kommt es bei der Analyse der LL(k)-Eigenschaft zu Fehlermeldungen, insbesondere aufgrund der sicheren Einbettung. Die Problematik der Entscheidungsfindung ist jedoch auch bei der Variante der Mehrfachvererbung vorhanden, weshalb mögliche Lösungsstrategien im Folgenden Teilabschnitt separat vorgestellt werden.

5.5.3. Entscheidungsfindung durch den Parser

Die Hauptproblematik bei der Ersetzung der Nichtterminale besteht darin, dass die ersetzten Nichtterminale auf den rechten Seiten anderer Produktionen verwendet werden. Abbildung 5.24 zeigt einen möglichen Ausschnitt der Automatensprache.

MontiCore-Grammatik «hw»

```
1 package mc;
2
3 grammar Automaton{
4
5   ...
6
7   Transition = from:Name "-" (Action | Stimulus)* ">" to:Name ";";
8
9 }
```

Abbildung 5.24.: Auszug aus der Automatensprache

In dieser Abbildung ist erkenntlich, dass Transitionen über beliebig viele Aktionen oder Stimuli verfügen können. Dementsprechend muss der Parser anhand der First-Mengen entscheiden können, welche dieser beiden Alternativen geparst werden soll, oder aber auch, ob der innere Block abgebrochen wird. Daher müssen die First-Mengen dieser Alternativen disjunkt sein. Diese Eigenschaft wird typischerweise vom Entwickler der Automatengrammatik sichergestellt, kann jedoch durch die Änderung des Inhaltes der Aktion verletzt worden sein. Im letzteren Falle gibt es zwei wesentliche Lösungsstrategien: einerseits kann in der Subgrammatik der Lookahead vergrößert werden und andererseits besteht die Möglichkeit der Angabe syntaktischer Prädikate. Die Lookaheaderhöhung kann bei der Variante der Mehrfachvererbung ausreichend sein, bei der Variante der Einbettung reicht dies jedoch nicht aus. Grund hierfür ist wie schon im vorigen Teil die sichere Einbettung: die First-Menge eines externen Nichtterminals ist beliebig, da gegen jede mögliche Einbettung geprüft werden muss. Daher sind bei der Strategie der Einbettung die First-Mengen der Aktion, des Stimulus und dem Abbruch der Schleife nicht disjunkt. Durch die Umstellung der First-Mengen-Berechnung kann auch hier ein syntaktisches Prädikat wie in Abbildung 5.25 eingesetzt werden, um eine Entscheidungsfindung durch den Parser zu ermöglichen.

MontiCore-Grammatik «hw»

Glue

```
1 package mc;
2
3 grammar Automaton{
4
5   external Extension;
6
7   Action = Extension;
8
9   Transition = from:Name "-" ( ("{") => Action | Stimulus)*
10                ">" to:Name ";";
11
12 }
```

Abbildung 5.25.: Umsetzung durch Einbettung mit syntaktischem Prädikat

Diese Abbildung ist eine Erweiterung der Vorgängerversion aus Abbildung 5.24. Geändert wurde die Definition der Transition durch zusätzliche Angabe eines syntaktischen Prädikates für die Aktion. Durch die Hinzunahme des Prädikates kann der Parser die verschiedenen Alternativen unterscheiden und es erfolgen keine weiteren Fehlermeldungen[4]. Als Anforderung an die eingebettete Sprache bleibt nur, dass diese mit einer geschweiften Klammer beginnen müssen, was im beschriebenen Beispiel der Java-Blockstatements der Fall ist. Weiterhin stellt die Anforderung an die eingebettete Sprache keinen Bruch mit dem Prinzip der Beliebigkeit der eingebetteten Sprache dar, da das Szenario der Nichtterminalersetzung die feste Verbindung zweier Sprachen zum Ziel hatte.

Insgesamt ergeben sich für das Szenario der Nichtterminal-Ersetzung die in den Tabellen 5.26 und 5.27 dargestellten Ansätze.

Szenario	Nichtterminal-Ersetzung
Name der Lösung	Mehrfachvererbung
Ausgangslage	• Zwei existierende Grammatiken G_1 und G_2 • Zu ersetzendes Nichtterminal N_1 in G_1 • Ersetzendes Nichtterminal N_2 in G_2
Lösungsstrategie	• Erstellen einer neuen Grammatik G_3 mit Vererbungsbeziehung zu G_1 und G_2 • Überschreiben von N_1 und Ableitung zu N_2 • Gegebenenfalls Erhöhung des Lookaheads oder Verwendung syntaktischer Prädikate bei Regeln, die N_1 auf der rechten Seite verwenden
Anmerkungen	Randbedingungen sind die Verschiedenheit für Namen der Elemente (Nichtterminale, Terminale etc) und die Disjunktheit lexikalischer Produktionen. Es entstehen Überlappungen bezüglich der Schlüsselwörter.

Tabelle 5.26.: Lösungsansatz durch Mehrfachvererbung für Nichtterminal-Ersetzung

5.5.4. Vergleich der Lösungen

Auch bei der Nichtterminal-Ersetzung unterscheiden sich die vorgeschlagenen Lösungsmöglichkeiten bezüglich der Umsetzung und eventueller Auswirkungen bzw. Randbedingungen. Tabelle 5.28 fasst die wesentlichen Eigenschaften der Lösungsmöglichkeiten zusammen.

[4] Hierbei wird davon ausgegangen, dass Stimuli nicht mit einer geschweiften Klammer beginnen können.

Szenario	Nichtterminal-Ersetzung
Name der Lösung	Einbettung
Ausgangslage	• Zwei existierende Grammatiken G_1 und G_2 • Zu ersetzendes Nichtterminal N_1 in G_1 • Ersetzendes Nichtterminal N_2 in G_2
Lösungsstrategie	• Erstellen einer neuen Grammatik G_3 mit Vererbungsbeziehung zu G_1 • Erstellen eines externen Nichtterminals E und Ableitung von N_1 zu E • Verwendung syntaktischer Prädikate bei Regeln, die N_1 auf der rechten Seite verwenden oder Einführung eines Zeichens vor E in der Regel für N_1, welches die LL(k)-Eigenschaft sichert.
Anmerkungen	Bei der Verwendung eines zusätzlichen Zeichens wird die Sprache verändert, es kann jedoch jede beliebige Sprache eingesetzt werden. Bei der Verwendung von Prädikaten erfolgt keine Sprachveränderung, der Anfang der eingebetteten Sprachen ist jedoch festgelegt.

Tabelle 5.27.: Lösungsansatz für Nichtterminal-Ersetzung durch Einbettung

Lösungsansatz	Mehrfachvererbung gemäß Tabelle 5.26
Anmerkungen	Randbedingungen sind die Verschiedenheit für Namen der Elemente (Nichtterminale, Terminale etc) und die Disjunktheit lexikalischer Produktionen. Es entstehen Überlappungen bezüglich der Schlüsselwörter.
Lösungsansatz	Einbettung gemäß Tabelle 5.27
Anmerkungen	Bei der Verwendung eines zusätzlichen Zeichens wird die Sprache verändert, es kann jedoch jede beliebige Sprache eingesetzt werden. Bei der Verwendung von Prädikaten erfolgt keine Sprachveränderung, der Anfang der eingebetteten Sprachen ist jedoch festgelegt.

Tabelle 5.28.: Lösungsansätze für Nichtterminal-Ersetzung

5.6. Einbettung einer einzelnen Sprache

Bei der Einbettung einer einzelnen Sprache werden zwei existierende Sprachen miteinander kombiniert, indem ein externes Nichtterminal der ersten Sprache durch ein Nichtterminal der

zweiten Sprache ersetzt wird. Beispiel hierfür sind Automaten mit einem externen Nichtterminal für eine Aktion und Java zur Verwendung als Aktionssprache. Die Umsetzung dieses Szenarios bezüglich der konkreten Syntax wurde bereits im Teilkapitel 5.2.1 vorgestellt, weshalb an dieser Stelle lediglich die Grenzen des Ansatzes diskutiert werden sollen.

Zunächst sind beide Grammatiken separat voneinander entwickelt worden, wobei sich insbesondere für die Grammatik der Automaten Bedingungen ergeben. Diese sind vor allem auf das bereits vorgestellte Konzept der sicheren Einbettung zurückzuführen. Der Parser der Automatengrammatik muss an allen Stellen entscheiden können, welche Alternative gewählt wird. Durch die sichere Einbettung wird dabei dem externen Nichtterminal jede beliebige First-Menge zugeordnet, weshalb hier besonderes Augenmerk seitens des Entwicklers liegen muss. Insbesondere die Optionalität der Einbettung stellt hierbei ein Problem dar, wie Abbildung 5.29 zeigt.

MontiCore-Grammatik «hw»

```
1 package mc.automaton;
2
3 grammar Automaton{
4
5   external Action;
6
7   //Definition der States etc.
8
9   Transition = from:Name "-" Action? ">" to:Name ";";
10
11 }
```

Abbildung 5.29.: Problem bei optionaler Einbettung

Bei der Definition der Transition wird hierbei die Aktion optional verwendet. Der Prüfmechanismus von MontiCore verwirft jedoch diese Grammatik, da im Falle der Einbettung einer Sprache, deren First-Menge Elemente beinhaltet, die in obiger Definition auf die Aktion folgen können (sog. Follow-Menge [ASU86]) keine eindeutige Entscheidung für oder gegen die eingebettete Sprache erfolgen kann. Um dem Parser die Einbettung dennoch zu ermöglichen, sollte die Entscheidung explizit gemacht werden. Abbildung 5.30 zeigt den Ansatz.

Hier wurde der optionale Block zusätzlich um die geschweiften Klammern erweitert um dadurch entscheiden zu können, ob der Teil der Aktion geparst werden soll. Beim Beispiel der Einbettung von Java-Blöcken müssten nun in den Instanzen je zwei öffnende bzw. schließende Klammern stehen - eine Klammer für aus der Definition der Automaten für Aktionen und eine für die Java-Blöcke. Dies kann vermieden werden, indem nicht Java-Blöcke selbst, sondern lediglich der Inhalt eingebettet wird. Falls die rechte Regelseite des Nichtterminals jedoch aus mehr als einem weiteren Nichtterminal besteht, kann eine Subgrammatik erstellt und die rechte Regelseite der Java-Blöcke zusammengefasst werden. Abbildung 5.31 zeigt das Vorgehen.

In dieser Abbildung wird deutlich, dass lediglich die geschweiften Klammern weggelassen wurden, ansonsten ist exakt der gleiche Produktionskörper vorhanden. In der finalen Version wird demnach statt des Nichtterminals für Blöcke der Java-Grammatik das Nichtterminal `EmbeddedBlockStatement` der Java-Subgrammatik verwendet. Daher sind auch nur je-

MontiCore-Grammatik «hw»

```
1 package mc.automaton;
2
3 grammar Automaton{
4
5   external Action;
6
7   //Definition der States etc.
8
9   Transition = from:Name "-" ("{" Action "}") ? ">" to:Name ";";
10
11 }
```

Abbildung 5.30.: Lösung der optionalen Einbettung

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javaForEmbedding;
2
3 grammar JavaForEmbedding extends mc.Java{
4
5   //Definition des BlockStatement in der Supergrammatik war
6   //BlockStatement = "{"  <<komplexBody>> "}";
7
8   //selbe Definition, jedoch ohne Klammern
9   EmbeddedBlockStatement = <<komplexBody>>;
10
11 }
```

Abbildung 5.31.: Erweiterung der Java-Grammatik zur Einbettung

weils eine öffnende bzw. schließende Klammer notwendig. Zusammenfassend ergibt sich der in Tabelle 5.32 aufgelistete Lösungsansatz zur Einbettung einer einzelnen Sprache.

Insgesamt bietet die Einbettung einer einzelnen Sprache eine Möglichkeit, die eingebettete Sprache während der Konfigurationszeit festzulegen, wodurch der Programmierer/Modellierer später keinen Einfluss auf die verwendetet Sprache hat. Diese Möglichkeit bietet hingegen die Einbettung von Sprachalternativen.

5.7. Einbettung von Sprachalternativen

Im Gegensatz zu den vorangegangen Szenarien sind bei der Einbettung von Sprachalternativen mindestens drei Sprachen beteiligt. Eine Sprache definiert hierbei ein externes Nichtterminal, mindestens zwei andere Sprachen sind für die Ersetzung dieses Nichtterminals angedacht. Als Beispiel dienen hierfür Klassendiagramme mit einem externen Nichtterminal für Vorbedingungen von Methoden, wobei hierfür alternativ OCL oder Java-assert-Statements verwendet werden können.

Szenario	Einbettung einer einzelnen Sprache
Name der Lösung	Umsetzung durch externe Nichtterminale
Ausgangslage	• Zwei existierende Grammatiken G_1 und G_2 • Externes Nichtterminal E in G_1 • Einzusetzendes Nichtterminal N in G_2
Lösungsstrategie	• Erstellung einer Sprachdatei bzw. Erstellung eines übergeordneten Parsers gemäß Abschnitt 5.2.1 • Hierin Definition der Ableitung von E zu N
Anmerkungen	Eventuell Erstellung einer Subgrammatik von G_2 nötig, um einzubettende Teile in einer separaten Regel zu definieren.

Tabelle 5.32.: Lösungsansatz für die Einbettung einer einzelnen Sprache

Die Festlegung, welche Sprache an einer bestimmten Stelle verwendet wird, erfolgt dabei dynamisch durch den Benutzer. Dieser kann innerhalb des Klassendiagramms festlegen, welche Sprache für das gesamte Diagramm verwendet wird. Alternativ kann an jeder Einbettungsstelle separat eine Sprache festgelegt werden. Abbildung 5.33 zeigt die Idee der dynamischen Festlegung.

Klassendiagramm

```
1 default OCL;
2 classdiagram MasterData{
3
4    class Person{
5      int age=0;
6
7      pre:  !(exists Person p: p.age < 0);
8      public void method1();
9
10     pre (Java): assert age<100;
11     public void method2();
12
13     pre: !(exists Person p: p.age < 0 || p.age >100);
14     public void method3();
15   }
16 }
```

Abbildung 5.33.: Beispiel zur Einbettung von Sprachalternativen

In dieser Abbildung wird in Zeile 1 festgelegt, dass zur Einbettung im Standardfall OCL verwendet wird. In der ersten Methode in Zeile 7 wird dann auch OCL zur Definition der Vorbe-

dingung verwendet. In der zweiten Methode wird hingegen Java verwendet, wobei hier in Klammern die neue Sprache zur Einbettung festgelegt wird. Die dritte Methode verwendet wiederum den Standardfall OCL. Anzumerken ist hierbei, dass die Standardsprache pro Modell/Datei separat festgelegt werden kann.

Die Umsetzung dieses Szenarios beruht auf den in [Kra10] vorgestellten Aktionen innerhalb von MontiCore-Grammatiken. Diese Aktionen sind Java-Anweisungen, die ausgeführt werden, falls der Parser an eine bestimmte Stelle im Erkennungsprozess kommt. Zusätzlich stellt MontiCore vordefinierte Aktionen zur Verfügung, die über eine abgekürzte Syntax gegenüber Java verfügen. Die Verwendung eines globalen Stacks ist dabei eine dieser vordefinierten Aktionen und wird zur Umsetzung des Szenarios verwendet.

Hierbei kann der Sprachentwickler globale Stacks definieren, die abhängig von Werten der späteren Instanz gefüllt werden. Abbildung 5.34 zeigt einen Auszug aus der Klassendiagramm-Grammatik.

MontiCore-Grammatik «hw»

```
1 grammar CD{
2
3   external PreCondition;
4
5   CompilationUnit = DefaultLanguage ClassDiagram;
6
7   DefaultLanguage = "default" language:Name
8                     astscript {push(embed,language);};
9
10  //Definitionen für Klassendiagramme, Klassen etc.
11  ...
12
13  Method = MethodWithDefaultLanguage | MethodWithSpecificLanguage;
14
15  MethodWithDefaultLanguage =
16            "pre"
17            Precondition<global embed>
18            MethodDeclaration;
19
20  MethodWithSpecificLanguage =
21            "pre" "(" language:Name astscript {push(embed,language);} ")"
22            Precondition<global embed>
23            MethodDeclaration;
24            astscript {pop(embed);};
25 }
```

Abbildung 5.34.: Grammatik zur Einbettung von Sprachalternativen

In dieser Abbildung wird ein globaler Stack namens `embed` verwendet, der jeweils in den Zeilen 8, 21 und 24 befüllt bzw. das oberste Element entfernt wird. In Zeile 8 wird dazu abhängig von der Eingabe - im speziellen vom Wert des Attributes `language` - der Stack initial gefüllt. Ist in einer Methodendefinition eine spezielle Sprache angegeben, wird in Zeile 21 dieser

Wert auf den Stack gelegt. Nach Beendigung der Erkennung der Methode wird dieser Wert in Zeile 24 wieder vom Stack entfernt. Bei Methoden ohne spezielle Sprachangabe erfolgt keine Manipulation des Stacks (siehe Zeilen 15-18).

Die Verwendung des externen Nichtterminals `PreCondition` in den Zeilen 17 und 22 erfolgt mit zusätzlichem Parameter in spitzen Klammern. Hierzu wird der Wert des obersten Elementes des globalen Stacks `embed` verwendet. Diese Angabe führt dazu, dass dem MontiCore-Parser nicht nur der Sprachwechsel angezeigt wird, sondern zusätzlich der Wert des Stacks übergeben wird. Anschließend kann der MontiCore-Parser abhängig von diesem Wert den entsprechenden Parser wählen. Die Definition der an der Einbettung beteiligten Sprachen erfolgt dabei wie bereits in Abschnitt 5.2.1 vorgestellt, zusätzlich wird jedoch angegeben, welcher Parser bei welchem Wert ausgewählt werden soll. Abbildung 5.35 zeigt die Definition für das gegebene Beispiel.

MontiCore-Sprachdatei «hw»

Glue

```
1 mc.cd.CD.CompilationUnit cd <<start>>;
2
3 mc.java.Java.AssertStatment assert in cd.PreCondition(Java);
4 mc.ocl.OCL.OCLCondition ocl in cd.PreCondition(OCL);
5
```

Abbildung 5.35.: Beispiel für eine Sprachdatei mit Einbettung von Sprachalternativen

Diese Abbildung zeigt dabei die gleiche Struktur wie Abbildung 5.6, wobei zusätzlich bei jeder eingebetteten Sprache am Ende in Klammern angegeben wird, bei welchem Parameter-Wert diese Sprache ausgewählt werden soll. Hierdurch wird - wie im Szenario vorgegeben - abhängig von der Eingabe die eingebettete Sprache gewählt. Insgesamt ergibt sich die in Tabelle 5.36 zusammengefasste Lösungsstrategie.

5.7.1. Alternative Umsetzung durch Einbettung einer Nichtterminal-Erweiterung

Alternativ zur vorgestellten Umsetzung durch parametrisierte Einbettung kann die Einbettung einer Nichtterminal-Erweiterung verwendet werden. Hierzu werden die eingebetteten Sprachen zunächst gemäß Abschnitt 5.4 zu einer Sprache vereinigt. Im vorliegenden Beispiel wird so eine Sprache gebildet, die Java-assert-Statements um OCL-Ausdrücke erweitert. Anschließend wird diese Sprache durch die Einbettung einer einzelnen Sprache gemäß Abschnitt 5.6 in die Klassendiagramm-Grammatik als Vorbedingung für Methoden eingebettet.

Diese Version der Umsetzung unterscheidet sich von der vorangegangen Version in verschiedenen Punkten. Zum einen muss der Modell/Programm-Schreiber während der Eingabe nicht selbst festlegen, welche Sprache er als Standard bzw. für eine konkrete Vorbedingung verwendet. Dies ist zwar prinzipiell vorteilhaft, die explizite Sprachwahl kann aber durchaus gewollt sein - beispielsweise im Falle einer bestimmten Art der Codegenerierung, die von der Auswahl der Sprache abhängig ist. Zum anderen kann jedoch schon die Nichtterminal-Erweiterung selbst zu Problemen führen, speziell im Falle von Ambiguitäten bei den beteiligten Grammatiken.

Szenario	Einbettung von Sprachalternativen
Name der Lösung	Umsetzung durch externe Nichtterminale
Ausgangslage	• Drei existierende Grammatiken G_1, G_2 und G_3 • Externes Nichtterminal E in G_1 • Einzusetzende Nichtterminale N_2 in G_2 und N_3 in G_3
Lösungsstrategie	• Verwendung des `astscript`-Konstrukts zur Befüllung eines Stacks abhängig vom Dateiinhalt • Erstellung einer Sprachdatei bzw. Erstellung eines übergeordneten Parsers gemäß Abschnitt 5.2.1 • Hierin Definition der Ableitung von E zu N_2 oder N_3 abhängig vom Inhalt des Stacks
Anmerkungen	Wie bei der Einbettung einer einzelnen Sprache ist eventuell die Erstellung einer Subgrammatik von G_2 oder G_3 nötig, um einzubettende Teile in einer separaten Regel zu definieren.

Tabelle 5.36.: Lösungsansatz für die Einbettung von Sprachalternativen

Würden beispielsweise nicht Assert-Statements aus Java verwendet, sondern lediglich boolesche Ausdrücke, wären diese nicht von der OCL unterscheidbar. Allein durch die explizite Festlegung der jeweiligen Sprache werden die Probleme der Ambiguitäten durch minimalen Aufwand seitens des Instanzentwicklers vermieden. Zusammenfassend ergibt sich die alternative Umsetzung durch Nichtterminal-Erweiterung gemäß Tabelle 5.37.

5.7.2. Vergleich der Lösungen

Die vorgeschlagenen Lösungen unterscheiden sich bezüglich der Umsetzung und eventueller Auswirkungen bzw. Randbedingungen. Tabelle 5.38 fasst die wesentlichen Eigenschaften der Lösungsmöglichkeiten zusammen.

5.8. Entwicklung einer erweiterbaren Sprache

Das übergeordnete Ziel dieses Szenarios ist die Verwendung von zur Komposition geeigneten Mechanismen zur Entwicklung einer erweiterbaren Sprache. Wie bereits aus den Ausführungen zu den unterschiedlichen Szenarien deutlich wurde, verfügt MontiCore über ein reichhaltiges Angebot an Mitteln zur Definition der konkreten Syntax. Welche konkreten Mittel für die Definition einer Sprache verwendet werden, unterliegt oftmals einer Designentscheidung seitens des Sprachentwicklers. Nichtsdestotrotz existieren Elemente, deren Verwendung sich hin-

Szenario	Einbettung von Sprachalternativen
Name der Lösung	Umsetzung durch externe Nichtterminale und Nichtterminal-Erweiterung
Ausgangslage	• Drei existierende Grammatiken G_1, G_2 und G_3 • Externes Nichtterminal E in G_1 • Einzusetzende Nichtterminale N_2 in G_2 und N_3 in G_3
Lösungsstrategie	• Vereinigung der Sprachen von G_2 und G_3 durch Nichtterminal-Erweiterung • Einbettung der vereinigten Grammatik
Anmerkungen	Durch fehlende explizite Festlegung durch Instanzentwickler müssen Ambiguitäten vom Sprachentwickler aufgelöst werden.

Tabelle 5.37.: Lösungsansatz für die Einbettung von Sprachalternativen durch Nichtterminal-Erweiterung.

Lösungsansatz	Umsetzung durch externe Nichtterminale gemäß Tabelle 5.36
Anmerkungen	Wie bei der Einbettung einer einzelnen Sprache ist eventuell die Erstellung einer Subgrammatik von G_2 oder G_3 nötig, um einzubettende Teile in einer separaten Regel zu definieren.
Lösungsansatz	Umsetzung durch externe Nichtterminale und Nichtterminal-Erweiterung gemäß Tabelle 5.37
Anmerkungen	Durch fehlende explizite Festlegung durch Instanzentwickler müssen Ambiguitäten vom Sprachentwickler aufgelöst werden.

Tabelle 5.38.: Lösungsansätze für die Einbettung von Sprachalternativen

sichtlich der Kompositionalität als mehr oder weniger vorteilhaft erweisen. Zusätzlich können Design-Richtlinien aufgestellt werden, die die Komponierbarkeit einer Sprache unterstützen. Im Folgenden werden sowohl diese Richtlinien als auch kompositionalitätsunterstützende Elemente vorgestellt. Anzumerken ist hierbei, dass der Sprachentwickler in manchen Fällen mögliche Erweiterungs- bzw. Änderungspunkte für seine Sprache antizipieren muss. Zwar kann generell nicht davon ausgegangen werden, dass alle möglichen Erweiterungen der Sprache vorhergesehen werden können, dennoch sind in vielen Fällen bereits bei der Entwicklung einer Sprache einige Erweiterungspunkte denkbar.

Richtlinie 1: Verwendung standardisierter lexikalischer Produktionen

Wie in Abschnitt 5.2.3 diskutiert wurde, kann es bei der Verwendung von Mehrfachvererbung zur Überschneidung lexikalischer Produktionen kommen. Als Resultat entstehen Sprachen, die nicht der Intention des Sprachentwicklers entsprechen. Daher ist es bei der Entwicklung einer neuen DSL empfehlenswert, die von MontiCore standardmäßig vorgegebenen lexikalischen Produktionen (z.B. für Namen, Strings, Kommentare, Leerzeichen etc. [Kra10]) zu verwenden. Zusätzlich können Standardproduktionen eingeführt werden, indem eine Basisgrammatik mit diesen Produktionen definiert wird und anschließend alle Grammatiken von dieser Basisgrammatik erben.

Wird eine so entwickelte Grammatik unter diesen Bedingungen mit einer weiteren Grammatik durch Mehrfachvererbung vereint und nutzt diese weitere Grammatik ebenfalls die Standardproduktionen, kommt es seitens der lexikalischen Produktionen zu keinen ungewollten Überschneidungen. Lediglich die Schlüsselwörter einer Sprache können nicht mehr als Namen in einer anderen Sprache verwendet werden. Da die Anzahl der Schlüsselwörter jedoch stark begrenzt ist, ist der Effekt wie bereits bei der Einführung des `assert`-Schlüsselwortes in Java 1.4 in vielen Fällen von geringer Bedeutung. Die Verwendung standardisierter lexikalischer Produktionen kann somit die Mehrfachvererbung als technisches Mittel zur Sprachkombination unterstützen.

Richtlinie 2: Namensgebung

Neben den Überschneidungen lexikalischer Produktionen entstehen bei der Mehrfachvererbung Probleme, falls Elemente wie Klassenproduktionen, Interfaces oder externe Nichtterminale der beteiligten Grammatiken gleichnamig sind. Durch die in MontiCore verwendete geordnete Mehrfachvererbung werden stets die zuerst gefundenen Elemente verwendet und alle anderen verworfen (siehe Abschnitt 5.2.3).

Zur Vermeidung dieser Problematik empfiehlt es sich, die Elemente einer Grammatik gegenüber Elementen anderer Grammatiken eindeutig zu benennen. Hierbei kann ein abkürzender Name der Grammatik allen Elementnamen vorangestellt werden. So kann beispielsweise eine Klassendefinition in Klassendiagrammen durch eine Regel `CDClassDefinition` definiert werden, in Java wird hingegen der Name `JavaClassDefinition` verwendet. Bei einer Mehrfachvererbung zu beiden Grammatiken kommt es hierdurch zu keinen Namensüberschneidungen und somit nicht zu ungewollten Ergebnissen.

Richtlinie 3: Reduktion der Regelgrößen

Im Falle der Erweiterung einer Sprache durch Vererbung ist es gegebenenfalls nötig, Teile der Supergrammatik zu wiederholen. So kann beispielsweise eine Grammatik für Automaten ohne Aktionen zu einer Grammatik für Automaten mit Aktionen erweitert werden, indem das Nichtterminal der Transition überschrieben wird und lediglich eine Referenz auf ein Nichtterminal für die Aktion hinzugefügt wird. Um diese Wiederholung so einfach wie möglich zu halten, sollten die Nichtterminalkörper relativ kompakt gehalten werden. Dies wird vor allem dadurch erreicht, dass beispielsweise Blöcke in separate Nichtterminale ausgelagert werden.

Zusätzlich ermöglichen kleine Regeln einen gewissen Grad an Reflektion der Änderungen in Subgrammatiken. Wird die Obergrammatik an einer Stelle angepasst, reflektiert sich die Änderung nur dann in Subgrammatiken, falls die entsprechende Regel nicht überschrieben wird. Durch die Verwendung kleiner Regeln wird die Wahrscheinlichkeit, dass eine Grammatik genau die Änderungsstelle überschreibt, verringert.

Richtlinie 4: Verwendung der objektorientierten Alternativschreibweise

MontiCore bietet zwei verschiedene Möglichkeiten zur Beschreibung von Alternativen an: die EBNF-Schreibweise und die objektorientierte Schreibweise. Abbildung 5.39 illustriert die Unterschiede.

MontiCore-Grammatik «hw»

```
1
2 interface A = B | C | D | E;
3
4
5
6
7
8
```

MontiCore-Grammatik «hw»

```
1
2 interface A;
3
4 B implements A = ...;
5 C implements A = ...;
6 D implements A = ...;
7 E implements A = ...;
8
```

Abbildung 5.39.: Unterschied zwischen EBNF- und objektorientierter Schreibweise

Im linken Teil der Abbildung wird die Regel `A` definiert und hierbei schon alle Alternativen für die Reduktion festgelegt. In der rechten Seite der Abbildung wird das Nichtterminal `A` ohne Produktionskörper definiert, wobei die Reduktionsmöglichkeit zu den verschiedenen anderen Alternativen separat bei der Definition der Nichtterminale erfolgt. Dies entspricht der objektorientierten Sichtweise, bei der in der Subklasse definiert wird, von welcher Oberklasse die Vererbung erfolgt und nicht umgekehrt. Von syntaktischer Seite sind beide Grammatiken gleich, d.h. die erkannte Sprache ändert sich nicht.

Besonders für die Verwendung syntaktischer Prädikate eignet sich die objektorientierte Schreibweise besser. Sind zur Unterscheidung der Alternativen Prädikate notwendig, müssen diese für alle Alternativen in der ersten EBNF-Variante bei Nichtterminal `A` angegeben werden. In der objektorientierten Variante werden diese separat bei den entsprechenden Alternativen angegeben. Im Sinne der Kompositionalität wirkt sich dieses Vorgehen bei der Vererbung aus: wird eines der Nichtterminale überschrieben, kann an gleicher Stelle das Prädikat neu definiert werden, während bei der EBNF-Schreibweise zusätzlich die Definition des Nichtterminals `A` überschrieben werden müsste, um das syntaktische Prädikat anzupassen.

Richtlinie 5: Identifikation von Teilsprachen

Ein wesentlicher Mechanismus zur Erstellung einer erweiterbaren Sprache ist die Identifikation sinnvoller Teilsprachen. Diese Teilsprachen sollten dabei typischerweise bestimmte Aspekte der Sprache (z.B. Expressions in Java) kapseln. Ein weiteres Separierungskriterium besteht, wenn

sich der Sprachentwickler vorstellen kann, dass bestimmte Aspekte der Sprache auch in einer anderen Sprache ausgedrückt werden können. Beispiel hierfür sind die Automaten mit einer Aktionssprache: die Automaten selbst beschreiben den Aspekt der Zustände und Zustandsübergänge während die Aktionssprache den Aspekt der auszuführenden Aktionen beschreibt. Die Aktionssprache selbst sollte nicht in der Automatensprache definiert sein, da es vorstellbar ist, unterschiedliche Aktionssprachen zu verwenden. Anzumerken ist hierbei, dass die Einteilung in verschiedene Sprachen nicht nur in Bezug auf das Mittel der Einbettung zu sehen ist. Vielmehr kann die Sprache auch in verschiedene separate Teile aufgeteilt werden und das Mittel der Sprachaggregation verwendet werden.

Prinzipiell ist die Zerlegung in Teilsprachen mit Aufwand verbunden. Es müssen mehrere Grammatiken geschrieben werden, die Einbettung bzw. Aggregation muss definiert werden usw. Vorteile entstehen jedoch erst bei der Wiederverwendung, welche oftmals nicht durch den Entwickler der Ursprungssprache, sondern durch andere Entwickler erfolgt. Nichtsdestotrotz sollte die Aufteilung aus Gründen der Wiederverwendbarkeit und im Hinblick auf eine langfristige Ersparnis erfolgen.

Richtlinie 6: Zusammenfassung von Sprachteilen

Neben der Aufteilung in verschiedene Teilsprachen trägt auch die Zusammenfassung von Sprachen zur Wiederverwendbarkeit bei. Hierbei werden die einzelnen Sprachen zunächst separat entwickelt und zusätzlich die konkreten Kombinationen abgelegt. Hierdurch ist es möglich, dass die Wiederverwendung nicht nur auf Basis einzelner Sprachen, sondern auch auf Basis sinnvoller Sprachkombinationen erfolgt. Die Ablage von Sprachkombinationen erfolgt dabei durch die Sprachdatei bei Einbettung (siehe Abschnitt 5.2.1) oder durch die Verwendung von Fabriken (siehe Abschnitt 5.3).

Zusammenfassend listet Tabelle 5.40 die vorgeschlagenen Maßnahmen zur Entwicklung einer erweiterbaren Sprache auf.

5.9. Verwandte Arbeiten

Die kompositionale Entwicklung der konkreten Syntax ist ein in der Literatur weit verbreitetes Forschungsgebiet. Existierende Ansätze konzentrieren sich dabei meist auf ein bestimmtes Anwendungsszenario und unterstützen nicht die volle Bandbreite der möglichen Kompositionsformen. Am meisten untersucht sind dabei die Szenarien der Nichtterminal-Erweiterung und der Nichtterminal-Ersetzung, wobei vor allem im Fachgebiet des Compilerbaus hierfür Forschungsansätze existieren. Dabei wurden spezielle Parsealgorithmen entwickelt, die in der Lage sind, Grammatiken zu kombinieren und somit die angesprochenen Szenarien umzusetzen. Diese Parsealgorithmen beruhen im Gegensatz zu den Standardklassen LL(k) und LR(k) meist auf der allgemeineren Gesamtklasse der kontextfreien Grammatiken, da sowohl LL(k)- als auch LR(k)-Sprachen unter Vereinigung nicht abgeschlossen sind (z.B. [BV07]).

Ein Hauptproblem bei der Verwendung von Standardalgorithmen zum Parsen in Bezug auf Kompositionalität ist die auch in MontiCore eingesetzte strikte Trennung zwischen Lexer und Parser. Einige der entwickelten Algorithmen heben diese Trennung auf, indem sie den Lexer

Szenario	Entwicklung einer erweiterbaren Sprache
Name der Lösung	Verwendung unterstützender Konstrukte
Ausgangslage	Entwicklung einer neuen Grammatik.
Lösungsstrategie	Verwendung unterstützender Maßnahmen: • Verwendung standardisierter lexikalischer Produktionen • Eindeutige Namensgebung • Reduktion der Regelgrößen • Verwendung der objektorientierten Alternativschreibweise • Identifikation von Teilsprachen • Zusammenfassung von Sprachteilen
Anmerkungen	Mögliche Erweiterungen müssen teilweise durch den Sprachentwickler antizipiert werden.

Tabelle 5.40.: Vorschläge zur Entwicklung einer erweiterbaren Sprache.

in den Parser integrieren (scannerless parsing [SC89]) oder zumindest den Lexer vom Parser aus steuern (context-aware scanning [WS07]). Speziell letzterer Ansatz ist zurzeit noch wenig verbreitet und es ist unklar, wo die Grenzen dieses Verfahrens liegen.

Ein weiterer Ansatz zur Umsetzung kompositionaler Parsealgorithmen besteht in der Verwaltung mehrerer Parsebäume, die zu einem bestimmten Zeitpunkt in der Erkennungsphase valide für den bisher erkannten Teiltext sind (z.B. [Ear70, Vis97]). Diese Parsebäume werden ständig anhand des analysierten Textes reduziert bzw. erweitert, wobei Prioritäten oder Typanalysen zusätzlich verwendet werden können, um die Menge der validen Bäume einzuschränken [BVVV05]. Wie jedoch korrekte Fehlermeldungen auf Basis mehrerer Parsebäume produziert werden können, ist zur Zeit nicht geklärt. Auch der Umgang mit mehreren Parsebäumen am Ende der Erkennungsphase ist Gegenstand der aktuellen Forschung (z.B.[KdJNNV09, dJNNKV09]).

In der Arbeit [Ada91] werden Grammatikmodule eingeführt, die über einen Import-Mechanismus verfügen. Dieser Mechanismus ähnelt der Vererbung in MontiCore und kann zur Umsetzung der Nichtterminal-Erweiterung und der Nichtterminal-Ersetzung verwendet werden. Zusätzlich können Grammatikmodule über Parameter verfügen, welche in anderen Modulen an konkrete Nichtterminale gebunden werden. Hierdurch kann zwar das Szenario der Einbettung einer einzelnen Sprache umgesetzt werden, insgesamt ist der Ansatz jedoch nichtkompositional, da aus den einzelnen Modulen nicht separat Parser und Lexer generiert werden.

Eine theoretische Grundlage für eine kompositionale Entwicklung der konkreten Syntax einer

Sprache in Bezug auf Einbettung wird in [Läm01] vorgestellt. Im Gegensatz zu Standardversionen von kontextfreien Grammatiken existieren hier sogenannte *bottom* Nichtterminale, deren Produktionskörper innerhalb der Grammatik nicht definiert sind. Eine Komposition mit einer Grammatik definiert in einer späteren Phase den eigentlichen Inhalt des Nichtterminals. Auf andere Kompositionsformen wie Aggregation oder Vererbung wird nicht eingegangen.

Eine kompositionale Semantik für modulare kontextfreie Grammatiken wird in [Win02] vorgestellt. Hier werden in einer Grammatik explizit importierte und exportierte Nichtterminale definiert, anhand derer sich Grammatiken kombinieren lassen. Sowohl [Win02] als auch [Läm01] konzentrieren sich auf die Erweiterung der Definition eines Nichtterminals durch andere Grammatiken. Mit welchen Parsertechnologien und unter welchen Randbedingungen diese Komposition realisierbar ist, wird nicht untersucht. Die Realisierbarkeit ist im Gegensatz zu [Läm01] und [Win02] ein zentraler Aspekt dieser Arbeit.

5.10. Zusammenfassung

In diesem Kapitel wurde die kompositionale Entwicklung konkreter textueller Syntax vorgestellt. Hierzu erfolgte zunächst eine Einführung in die generelle Arbeitsweise des Erkennungsprozesses innerhalb der generierten Lexer/Parser. Da die Komponenten von ANTLR keine Kompositionalität unterstützen, erfolgten Modifikationen und Erweiterungen. Hierbei verwenden verschiedene Lexer/Parser einen gemeinsamen Eingabestrom und eine gemeinsame Queue zur Speicherung bereits erstellter, aber noch nicht konsumierter Token. Durch die Funktionalität des Zurücksetzens der Eingabe und der Leerung der Queue konnten Lexer und Parser bei der Spracheinbettung während des Erkennungsprozesses ausgetauscht werden. Weiterhin wurde der Mechanismus der First-Mengen-Berechnung auf die Einbeziehung syntaktischer Prädikate umgestellt.

Nach der Vorstellung der Konfigurationsmöglichkeiten der Kompositionalitätsmechanismen Einbettung, Vererbung und Aggregation und der Diskussion des Lexer/Parser-Wechsels erfolgte eine detaillierte Analyse der Anwendungsszenarien. Um das gewünschte Verhalten zu erreichen, wurden die vorhandenen Kompositionalitätsmechanismen teilweise kombiniert. Zusätzlich wurden in einigen Szenarien unterschiedliche Lösungsstrategien und ihre Vor- und Nachteile vorgestellt, vor allem in Bezug auf die Verwendung von Mehrfachvererbung gegenüber Einfachvererbung und Einbettung.

Insgesamt werden alle Szenarien durch die vorhandenen Kompositionalitätsmechanismen abgedeckt. Zusätzlich beschränkt sich der Aufwand der Komposition auf wenige Zeilen, entweder in Form der Sprachdateien bei Einbettung, in Form der `extends`-Klausel bei Mehrfachvererbung oder in Form weniger Codezeilen bei Sprachaggregation.

Kapitel 6.

Kompositionale Entwicklung abstrakter Syntax

MontiCore verfügt über ein integriertes Format zur Definition der konkreten und abstrakten Syntax einer Sprache. Ausgehend von der Spezifikation in Form einer Grammatik werden hierbei sowohl Komponenten zur Erkennung der konkreten Syntax, als auch Komponenten zur inneren Darstellung der eingelesenen Modelle - der abstrakten Syntax - generiert. Der aus einer Grammatik gewonnene Parser erstellt dann während der Erkennungsphase eine Instanz der abstrakten Syntax - den abstrakten Syntaxbaum (engl. abstract syntax tree - AST).

Anschließend wird ein AST typischerweise durch Komponenten weiterverarbeitet, die stark von den Klassen der abstrakten Syntax abhängen. Beispiele hierfür sind etwa Symboltabellen, Überprüfungen von Kontextbedingungen, semantische Analysen und Codegeneratoren. Diese Algorithmen arbeiten dabei meist auf Basis des AST, indem sie dessen hierarchische Struktur traversieren. Durch die starke Kopplung der abstrakten Syntax und der darauf basierenden Algorithmik werden innerhalb dieses Kapitels beide Elemente vorgestellt, insbesondere die von MontiCore bereitgestellte Version des Visitor-Musters [GHJV95] und das Zusammenspiel mit den Kompositionalitätsmechanismen auf der Ebene der abstrakten Syntax.

Insgesamt beinhaltet dieses Kapitel

- eine Vorstellung der Kompositionalitätsmechanismen der abstrakten Syntax,
- eine Darstellung des Visitor-Konzepts und dessen Anwendung bei Sprachkomposition, insbesondere Möglichkeiten zum Informationsaustausch zwischen unabhängig voneinander entwickelten Visitoren,
- die Untersuchung der Anwendungsszenarien bezüglich der abstrakten Syntax,
- die Vorstellung verwandter Arbeiten und eine abschließende Zusammenfassung.

6.1. Kompositionalitätsmechanismen und abstrakte Syntax

Die verschiedenen Kompositionalitätsmechanismen werden in MontiCore auf unterschiedliche Weise in der abstrakten Syntax reflektiert. Prinzipiell wird dabei versucht, die besonderen Eigenschaften der Mechanismen bezüglich der konkreten Syntax kohärent auf die abstrakte Syntax und gleichermaßen auf die darauf basierende Algorithmik zu übertragen. Im Folgenden werden die wesentlichen Merkmale zur Unterstützung der Kompositionalität detailliert beschrieben.

Sprachaggregation

Bezüglich der Sprachaggregation besteht die wesentliche Unterstützung in der Erstellung verschiedener ASTs für verschiedene Modellarten. Aus jeder Grammatik werden zunächst separat AST-Klassen generiert, die anschließend vom entsprechenden Parser instanziiert werden. Durch die Kombination der Sprachen gemäß Abbildung 4.8 bleiben die generierten AST-Klassen unabhängig voneinander nutzbar.

Bezüglich der Algorithmik spielen vor allem die Workflows bei der Kombination durch Sprachaggregation eine Rolle. Workflows sind prinzipiell mit der Art der Root parametrisiert, für welche sie Anwendung finden sollen. Da für jede Sprache spezielle Roots existieren, kommt es bei Sprachaggregation zu keinen Überschneidungen. Hierdurch können die Algorithmen innerhalb der Workflows unverändert übernommen werden.

Einbettung

Die für die Einbettung verwendeten externen Nichtterminale werden in der abstrakten Syntax durch Attribute des Basistyps aller AST-Klassen `ASTNode` abgebildet. Durch diese Vorgehensweise ist es möglich, bei konkreter Kombination jeden AST einer eingebetteten Sprache zur Besetzung des entsprechenden Attributs zu verwenden. Zusätzlich wird hierdurch vermieden, dass bei jeder Sprachkombination erneut AST-Klassen mit spezifisch getypten Attributen für externe Nichtterminale generiert werden müssen.

In einigen Fällen empfiehlt es sich jedoch, den Typ des AST-Attributs für externe Nichtterminale anzupassen. Hierdurch können etwa Anforderungen an eingebettete Sprachen in Form von zu implementierenden Interfaces des Wurzel-Knotens formuliert werden. Diese Interfaces können dabei direkt in Java formuliert werden, die Typisierung des Attributs erfolgt jedoch direkt in der MontiCore-Grammatik. Abbildung 6.1 zeigt die Grammatik und die entsprechende abstrakte Syntax exemplarisch auf.

Die beiden eingeführten externen Nichtterminale unterscheiden sich dabei nur insoweit, dass bei `Action2` in Zeile 5 zusätzlich ein Typ hinter dem Schrägstrich angegeben wurde. Hierdurch wird dieser Typ in der abstrakten Syntax übernommen. Wird wie bei `Action1` in Zeile 4 kein expliziter Typ angegeben, wird `ASTNode` verwendet.

Neben der Abbildung der externen Nichtterminale in der abstrakten Syntax ist im Falle der Einbettung der Umgang mit Algorithmen in Form von Visitoren von wesentlicher Bedeutung. Diese unterstützen die Kompositionalität, indem sie bei konkreter Sprachkombination unverändert und ohne erneute Kompilierung wiederverwendet werden können. Die konkrete Funktionsweise der Visitoren wird im Abschnitt 6.2 vorgestellt.

Vererbung

Bei Sprachvererbung wird in MontiCore prinzipiell nur der betroffene Anteil der abstrakten Syntax neu generiert. Hierzu zählen neue AST-Klassen für neu definierte Regeln oder auch überschriebene und somit erweiterte Regeln der Supergrammatik. Beim Überschreiben von Regeln wird zusätzlich aus Wiederverwendungsgründen eine Vererbungsbeziehung zur AST-Klasse der

MontiCore-Grammatik «hw»

```
1
2 grammar AutomatonWithActions{
3
4    external Action1;
5    external Action2 / mc.automaton.IAction;
6
7    TransitionWithAction1 = from:Name "-" action:Action1 ">" to:Name;
8    TransitionWithAction2 = from:Name "-" action:Action2 ">" to:Name;
9
10 }
```

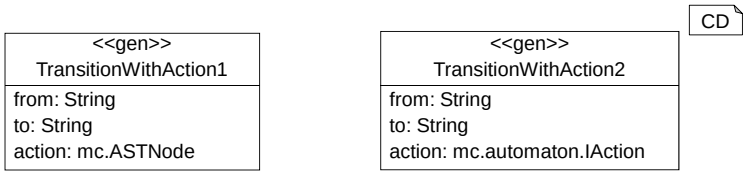


Abbildung 6.1.: Typisierung der Attribute für externe Nichtterminale

Supergrammatik erstellt. Abbildung 6.2 zeigt ein Beispiel für Grammatikvererbung und die entstehende abstrakte Syntax.

In dieser Abbildung existieren im oberen Teil zwei Grammatiken, wobei die rechte Grammatik von der linken erbt. Hierbei werden drei verschiedene Mechanismen angewandt: erstens wird eine Regel (`Automaton`) durch Regelvererbung erweitert, zweitens wird eine Regel (`State`) überschrieben und drittens wird eine Regel (`Transition`) übernommen. Die Auswirkungen sind dabei wie folgt:

- Bei Regelvererbung zwischen `ExtendedAutomaton` und `Automaton` entsteht eine Vererbungsbeziehung. Da die Intention der Regelvererbung darin besteht, ein Nichtterminal zu erweitern, kann der Parser Instanzen beider Klassen abhängig von erkannten Text produzieren.
- Auch beim Überschreiben der Regel `State` entsteht eine Vererbungsbeziehung. Der Parser der Subgrammatik produziert jedoch nur Instanzen des Subtyps.
- Da für die Regel `Transition` keine Änderungen erfolgten, wird nichts neu generiert. Der Parser produziert Instanzen der Klasse, die aus der Supergrammatik erzeugt wurde.

Die Vorgehensweise bietet unterschiedliche Vorteile im Bezug auf Kompositionalität. Zum Ersten entsteht durch den Verzicht der Regenerierung unveränderter AST-Klassen ein Geschwindigkeitsvorteil, da der Generierungsprozess auf ein Minimum beschränkt ist. Zum Zweiten können Visitoren wiederverwendet werden, da keine neuen Klassen entstehen. Diese Eigenschaft beschränkt sich jedoch nicht nur auf die Teile, die nicht generiert sondern von der Supergrammatik übernommen wurden, sondern vielmehr auch auf überschriebene und erweiterte Nichtterminale. Hauptgrund hierfür ist die eingeführte Vererbung zwischen den neu generierten Klassen

MontiCore-Grammatik «hw»

```
1 package mc.automaton;
2
3 grammar AutomatonGrammar{
4
5
6
7   Automaton = ... ;
8
9
10
11   State = ... ;
12
13   Transition = ... ;
14 }
```

MontiCore-Grammatik «hw»

```
1 package mc.extendedAutomaton;
2
3 grammar AutomatonGrammar2 extends
4   mc.automaton.AutomatonGrammar{
5
6   //extend Automaton
7   ExtendedAutomaton
8      extends Automaton = ...;
9
10  //override State
11  State = ... ;
12
13  //Do nothing for Transition
14 }
```

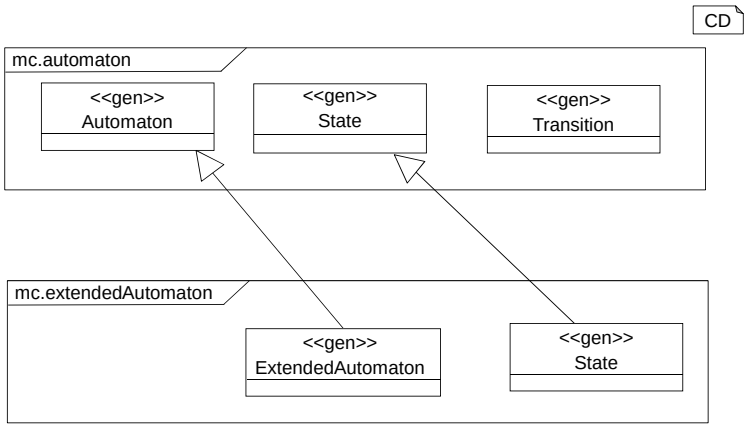


Abbildung 6.2.: Auswirkung von Grammatikvererbung, Regelvererbung und Überschreiben von Regeln

und den AST-Klassen der Supergrammatik. Daher können die Visitoren für Supergrammatiken unverändert bleiben bzw. gegebenenfalls mit leichten Änderungen für die Subklassen erweitert werden. Diese Erweiterungsmöglichkeiten werden im nächsten Abschnitt vorgestellt.

6.2. Visitoren

MontiCore stellt einen auf Kompositionalität spezialisierten Visitor-Mechanismus zur Verfügung. Dieser Mechanismus ist in der Lage, Visitoren bei Sprachkombination durch Mehrfachvererbung oder Einbettung zur Konfigurationszeit und ohne Neukompilierung zu kombinieren. Hierfür wird einerseits Reflection, andererseits auch Delegation verwendet. Die generelle Struktur für Visitoren wird in Abbildung 6.3 aufgezeigt.

Die vom Framework vorgegebene Klasse `Visitor` verfügt hierbei über mehrere Instanzen

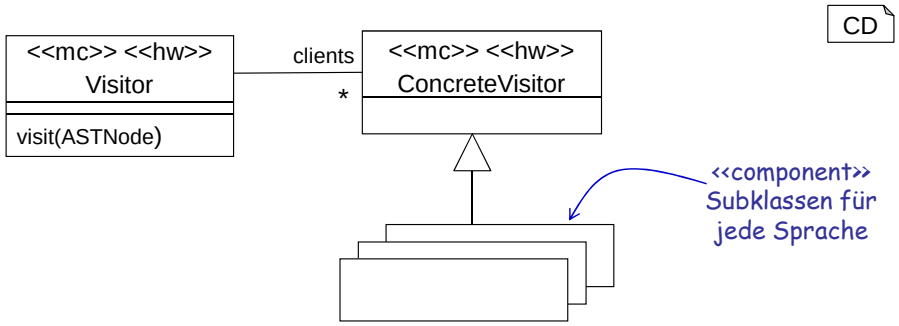


Abbildung 6.3.: Genereller Aufbau von Visitoren

von Subklassen des `ConcreteVisitor`. Diese Instanzen sind dabei sprachspezifisch, d.h. sie enthalten `visit`-Methoden, deren Parameter mit AST-Klassen der entsprechenden Grammatik getypt sind. Die Verwendung des Ansatzes ergibt sich dabei wie folgt: zunächst wird eine Instanz des `Visitors` erstellt und je nach beteiligten Sprachen Instanzen der entsprechenden Subklassen von `ConcreteVisitor` registriert. Anschließend wird die `visit`-Methode des `Visitors` mit dem Wurzelknoten des ASTs aufgerufen. Dieser ruft je nach Typ des Knotens die korrekte spezifische Methode eines `ConcreteVisitors` auf. Beim Besuch der untergeordneten Knoten wird entsprechend verfahren.

Die Technik zum Aufruf der korrekten Methoden ähnelt dabei der im Runabout [Gro03] bzw. Walkabout [PJ98] vorgeschlagenen Vorgehensweise. Hierbei werden die `visit`-Methoden nach dem Typ ihres Parameters in einer Lookup-Table aufgeschlüsselt, zur Laufzeit wird anschließend für jeden Knotenbesuch der Typ des Knotens analysiert und die entsprechende Methode herausgesucht. Existiert eine entsprechende Methode, wird diese anschließend aufgerufen. Hierbei ergibt sich ein gewisser Grad an Variabilität bezüglich der Auswahl der korrekten Methode und der Anzahl aufgerufener Methoden pro Knoten, die in unterschiedlichen Subklassen des `Visitors` realisiert sind:

1. Es wird die Methode aufgerufen, deren Parametertyp genau dem Typ des AST-Knotens entspricht. Existieren mehrere solcher Methoden in verschiedenen Instanzen von `ConcreteVisitor`, wird die Methode der zuletzt hinzugefügten Instanz aufgerufen. Dieses Verhalten ist in der Klasse `Visitor` implementiert.

2. Es wird die Methode aufgerufen, deren Parametertyp eine Superklasse des Typs des AST-Knotens ist. Existieren mehrere passende Methoden mit unterschiedlichem Parametertyp, so wird die Methode gewählt, deren Parametertyp in der Vererbungshierarchie am nächsten zum Knotentyp liegt. Sind hiervon mehrere Methoden vorhanden, wird wie in Punkt 1 verfahren. Diese Vorgehensweise ist im `InheritanceVisitor` implementiert.

3. Es werden die Methoden aufgerufen, deren Parametertypen genau dem Typ des AST-Knotens entsprechen. Diese Vorgehensweise entspricht dem ersten Fall, es werden jedoch alle passenden Methoden aufgerufen. Diese Vorgehensweise ist im `MultiVisitor` implementiert.

4. Es wird sowohl die Vererbungshierarchie beachtet, als auch mehrere Methodenaufrufe ermöglicht. Diese Strategie kombiniert den zweiten und dritten Anwendungsfall und wird im `MultiInheritanceVisitor` bereitgestellt.

6.2.1. Kombination von Visitoren

Einbettung

Die Kombination von Visitoren bei Einbettung erfolgt über die Registrierung von Instanzen der jeweiligen Subklassen von `ConcreteVisitor`. Da aus den beteiligten Grammatiken verschiedenartige AST-Klassen generiert wurden, können sich die Visitoren nicht gegenseitig beeinflussen. Insbesondere können keine `visit`-Methoden mit gleichem Parametertyp in verschiedenen Visitoren existieren.

Vererbung

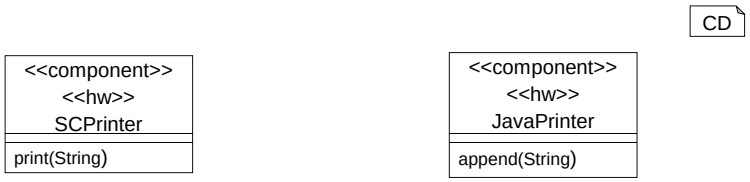
Auch bei Mehrfachvererbung können die Visitoren wie bei der Einbettung kombiniert werden. Da auch hier aus den beteiligten Grammatiken unabhängige AST-Klassen generiert wurden, kommt es auch bei der Mehrfachvererbung von Grammatiken zu keinen Überschneidungen. Zusätzlich können weitere Visitoren erstellt werden, welche die eventuell vorhandenen zusätzlichen AST-Klassen der Subgrammatik behandeln.

6.2.2. Informationsaustausch bei Einbettung

Visitoren werden meist eingesetzt, um Berechnungen auf Basis der Struktur des ASTs auszuführen. Diese Berechnungen hängen dabei oftmals nicht nur von einem einzelnen Knoten ab, sondern von einer Vielzahl von Informationen innerhalb des gesamten ASTs. Daher werden häufig während der Traversierung zunächst nötige Informationen gesammelt und aufbereitet, die eigentlichen Berechnungen finden parallel oder im Anschluss statt.

Diese Vorgehensweise wirkt sich unmittelbar auf die Spracheinbettung aus: die beteiligten Visitoren sind unabhängig voneinander entwickelt worden und haben keinerlei Kenntnis über Informationen, die von den Berechnungen der jeweils anderen Sprache benötigt werden. Hier kann es zu Situationen kommen, in denen ein Visitor einer einbettenden Sprache Informationen in einem bestimmten Format von einem Visitor der eingebetteten Sprache anfordert. Hier muss also ein Informationsaustausch zwischen zwei unabhängig voneinander entwickelten Komponenten - den Visitoren - erfolgen. Als exemplarisches Beispiel wird hierfür die Einbettung von Java in Statecharts verwendet, wobei jeweils ein Visitor als Prettyprinter vorhanden ist. Abbildung 6.4 illustriert das Beispiel.

In diesem Beispiel sind in der oberen Hälfte die Klassen dargestellt, in denen der jeweilige gedruckte Code hinterlegt wird. Beide Klassen sind unabhängig voneinander entwickelt worden und verfügen über unterschiedliche Methoden. Im unteren Teil der Abbildung sind die jeweiligen Visitoren skizziert. Diese instanziieren in Zeile 4 jeweils den entsprechenden Printer und fügen in den `visit`-Methoden den jeweiligen zu druckenden Text hinzu.



Java-Quellcode «hw»

```
1 class SCVisitor
2   extends ConcreteVisitor{
3
4  SCPrinter printer=
5     new SCPrinter();
6
7  public void visit(State s){
8    printer.print("...");
9  }
10
11 }
```

Java-Quellcode «hw»

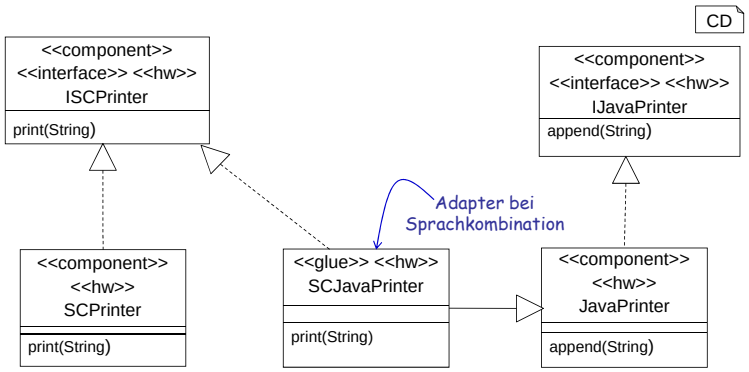
```
1 class JavaVisitor
2   extends ConcreteVisitor{
3
4  JavaPrinter printer=
5     new JavaPrinter();
6
7  public void visit(Statement s){
8    printer.append("...");
9  }
10
11 }
```

Abbildung 6.4.: Beispiel für Prettyprinter in Form von Visitoren

Die Problematik bei der Kombination der Visitoren besteht in diesem Beispiel darin, dass diese in verschiedene Zielobjekte verschiedener Klassen drucken. Zwar garantiert der MontiCore-Mechanismus die Richtigkeit der Aufrufreihenfolge der `visit`-Methoden, die gemeinsame Nutzung von Objekten in verschiedenen Visitoren kann jedoch nicht vom Framework vorgegeben werden. Der Grund hierfür besteht darin, dass nicht generell vorgegeben werden kann, welche Art von Informationen in welchem Format ausgetauscht werden müssen. Diese Eigenschaften hängen stark von den beteiligten Sprachen und dem Zweck der Visitoren ab.

Um dennoch einen Informationsaustausch zwischen unabhängig voneinander entwickelten Visitoren zu ermöglichen, muss bereits bei der Entwicklung der Visitoren ein auf Kompositionalität ausgelegtes Design verwendet werden. Insbesondere sollte es ein solcher Designvorschlag ermöglichen, die vorhandenen Visitoren ohne Änderung und unter eventueller Erstellung von Glue Code zu kombinieren. Hierzu bedient sich der Designvorschlag zweier bereits in Abschnitt 4.7 vorgeschlagener Mechanismen: Fabriken und Adaptoren. Fabriken werden verwendet, um die Objekte der Datenhaltungsklassen zu instanziieren, Adaptoren werden eingesetzt, um verschiedene Schnittstellen der Datenhaltungsklassen ineinander zu übersetzten. Abbildung 6.5 zeigt das Beispiel von Statecharts und Java unter Verwendung des vorgeschlagenen Designs.

Die Abbildung unterscheidet sich in einigen wesentlichen Punkten von der vorangegangenen Version. Zum Ersten werden innerhalb der Visitoren Fabriken benutzt, um die entsprechenden Printer zu instanziieren. Durch einfache Anpassungen der Fabriken ist es dann möglich, Instanzen von Subklassen (z.B. die Adaptoren) der eigentlichen Printer zu produzieren. Hierbei ist zusätzlich darauf zu achten, dass beide Fabriken dasselbe Objekt zurückliefern. Zum Zweiten



Java-Quellcode «hw»

```
1  class SCVisitor
2    extends ConcreteVisitor{
3
4   ISCPrinter printer=
5     SCFactory.createPrinter();
6
7   public void visit(State s){
8     printer.print("...");
9   }
10
11 }
```

Java-Quellcode «hw»

```
1  class JavaVisitor
2    extends ConcreteVisitor{
3
4   IJavaPrinter printer=
5     JavaFactory.createPrinter();
6
7   public void visit(Statement s){
8     printer.append("...");
9   }
10
11 }
```

Abbildung 6.5.: Beispiel für kompositionales Design von Visitoren

wurden explizite Interfaces verwendet, die einen gewissen Grad an Abstraktion vom verwendeten Printer erlauben. Die Visitoren haben dabei nur Kenntnis von den Interfaces, nicht jedoch von den implementierenden Klassen. Durch die Verwendung der Interfaces ist es möglich, eine Adaptor-Klasse zu erstellen, die den Anforderungen beider Visitoren genügt.

Das vorgestellte Design bietet mehrere Vorteile: Erstens können die Visitoren ohne Änderungen kombiniert werden. Zweitens ermöglicht das Design die gemeinsame Verwendung von Datenhaltungsklassen ohne dass diese im Voraus festgelegt werden müssen. Und drittens ermöglicht die Verwendung von Fabriken und Adaptoren eine einfache Übersetzung der Schnittstellen der Visitoren. Auf diese Art und Weise sind die erstellten Visitoren von Anfang an auf Kompositionalität ausgelegt.

6.2.3. Erweiterung bei Vererbung

Wie bereits in Abschnitt 6.1 beschrieben wurde, kann es zu verschiedenen Konstellationen zwischen den abstrakten Syntaxen von Grammatiken in einer Vererbungsbeziehung kommen. Dabei entsteht durch (1) Regelvererbung oder durch (2) das Überschreiben von Regeln eine objektorientierte Vererbung der AST-Klassen, (3) AST-Klassen unveränderter Regeln werden nicht neu generiert und (4) für neu angelegte Regeln werden neue Klassen erstellt. Für diese Konstella-

tionen stehen unterschiedliche Äquivalente zur Behandlung einer erweiterten bzw. geänderten abstrakten Syntax innerhalb der von MontiCore zur Verfügung gestellten Visitoren bereit. Im Folgenden werden diese Mechanismen für jeden der angesprochenen Fälle vorgestellt.

Erweiterung bei Regelvererbung

Bei der Regelvererbung zwischen einer Regel `SubFoo` und einer Regel `Foo` kann der resultierende Parser sowohl Instanzen der AST-Klasse der Basisregel als auch Instanzen der AST-Klasse der vererbenden Regel produzieren. Deshalb ist es in den meisten Fällen erwünscht, einen Visitor für die Obergrammatik um zusätzliche Funktionalität für die erbende Regel zu erweitern. Die Funktionalität für die Instanzen der Basisregel soll dabei erhalten bleiben.

Hierfür können zwei wesentliche Ansätze identifiziert werden. Erstens kann wie in Abbildung 6.6 aufgezeigt, eine Subklasse des Visitors für die Basisgrammatik gebildet werden, welche zusätzlich eine `visit`-Methode für die erbende Regel beinhaltet. Zweitens ist es möglich, einen unabhängigen Visitor zu entwickeln, welcher die neue Funktionalität beinhaltet. Hier erfolgt keine Vererbung zum ursprünglichen Visitor, vielmehr werden beide im Falle der Einbettung unabhängig registriert. Abbildung zeigt 6.7 diesen Ansatz.

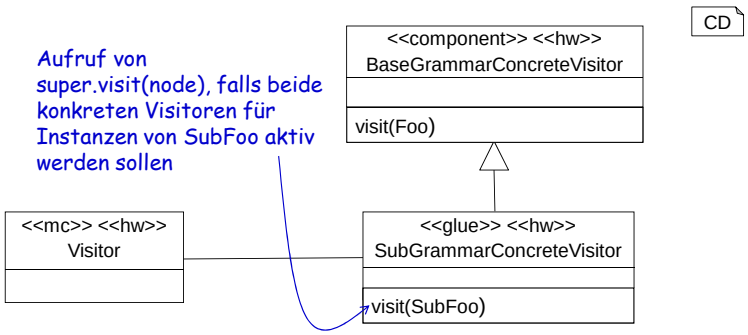


Abbildung 6.6.: Erweiterung der Visitorfunktionalität durch Vererbung

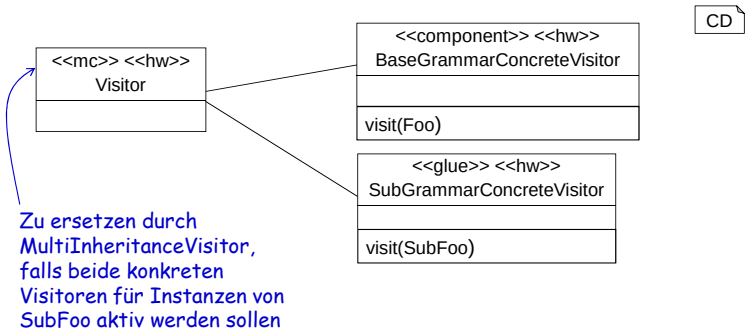


Abbildung 6.7.: Erweiterung der Visitorfunktionalität mittels unabhängiger Visitoren

Aus technischer Sicht sind beide Vorgehensweisen gleichwertig. Durch den Einsatz der Vererbung kann im Visitor der Subgrammatik zwar leichter auf die Daten des Visitors für die Ober-

grammatik zugegriffen werden, durch den Einsatz des im vorigen Abschnitts beschriebenen Designs ist dies jedoch auch im Falle der unabhängigen Visitoren möglich. Wenn zusätzlich die Funktionalität des Visitors für die Supergrammatik für die AST-Klasse verwendet werden soll, ist im Falle der Visitor-Vererbung das `super`-Konstrukt von Java zu verwenden, bei unabhängigen Visitoren ist ein `MultiInheritanceVisitor` einzusetzen. In beiden Fällen werden die `visit`-Methoden beider Visitoren für Instanzen der AST-Klasse der erbenden Regel aufgerufen.

Zusätzlich treten Fälle auf, in denen die vom Visitor der Obergrammatik bereitgestellte Funktionalität auch für die Instanzen der Subklasse aus der Subgrammatik ausreicht und somit kein neuer Visitor entwickelt werden muss. Hierbei ist darauf zu achten, dass ein `InheritanceVisitor` verwendet wird, da die `visit`-Methode im anderen Fall nicht für Instanzen der Subklasse aufgerufen wird. Abbildung 6.8 zeigt die Umsetzung.

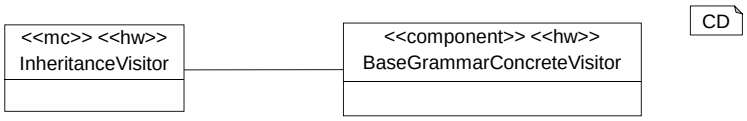


Abbildung 6.8.: Umsetzung ohne Hinzufügen von Funktionalitäten

Erweiterung beim Überschreiben von Regeln

Im Gegensatz zur Regelvererbung produziert der Parser beim Überschreiben von Regeln lediglich Instanzen der AST-Klasse der Subgrammatik. Falls für diese AST-Klasse zusätzliche Funktionalität erforderlich ist, muss auch hier entweder ein vorhandener Visitor durch Vererbung erweitert werden oder aber ein eigenständiger Visitor entwickelt und registriert werden. Wie bereits bei der Regelvererbung diskutiert, unterscheiden sich beide Möglichkeiten aus technischer Sicht nicht.

Falls auch beim Überschreiben die Funktionalität des Visitors der Obergrammatik ausreichend ist, muss auch hier ein `InheritanceVisitor` verwendet werden. Es existieren in jedem AST zwar grundsätzlich nur Instanzen der Subklasse, der korrekte Aufruf der entsprechenden `visit`-Methode wird jedoch nur von dieser Visitor-Version unterstützt, da die `visit`-Methode des Visitors der Supergrammatik mit der AST-Klasse der Regel der Supergrammatik parametrisiert ist.

Unveränderte und neu angelegte Regeln

Die letzten beiden Fälle der möglichen Beziehungen zwischen den AST-Klassen zweier in Vererbungsbeziehung stehenden Grammatiken sind die unveränderte Übernahme bestehende Regeln und das Hinzufügen neuer Regeln. Bei der unveränderten Übernahme der Regeln kann auch auf Seiten des Visitors eine unveränderte Übernahme erfolgen, d.h. der bestehende Visitor wird entweder neben eventuellen zusätzlichen Visitoren wiederverwendet oder zusätzliche Visitoren erben von der bestehenden Version und Überschreiben die entsprechenden `visit`-Methoden nicht.

Auch bei der Einführung zusätzlicher Regeln kann entweder ein neuer Visitor neben dem Visitor der Supergrammatik eingefügt werden oder aber eine Vererbungsbeziehung zu bestehenden Versionen eingeführt werden. Beide Vorgehensweisen unterscheiden sich aus technischer Sicht nicht.

Veränderung einzelner Berechnungen

Neben dem Hinzufügen oder Beibehalten existierender AST-Klassen kommt es auch zu Situationen, in denen die von einem Visitor zur Verfügung gestellte Funktionalität verändert werden muss. Hierbei wird also nicht auf neue Knotentypen reagiert, vielmehr wird eine bestehende Berechnung verändert. Grund hierfür kann etwa eine Änderung der Anforderungen an die Berechnung für eine bestimmte AST-Klasse sein, beispielsweise durch Änderung der abstrakten Syntax an einer anderen Stelle.

Auch hierfür lässt sich sowohl die Vererbung als auch die zusätzliche Entwicklung eines eigenständigen Visitors verwenden. Wenn die gegebene Funktionalität lediglich erweitert und nicht gänzlich überschrieben werden soll, muss wie bereits bei der Regelvererbung das `super`-Konstrukt von Java verwendet werden, bei unabhängigen Visitoren ist jedoch ein `MultiVisitor` ausreichend. Soll die Funktionalität komplett ersetzt werden, wird bei der ersten Version auf den `super`-Aufruf verzichtet, bei der zweiten Version wird ein normaler `Visitor` eingesetzt. Hierbei ist zusätzlich zu beachten, das zwingend der Visitor mit der neuen Funktionalität als letztes registriert wird.

Wiederverwendung der Funktionalität von Obergrammatiken

In den vorangegangenen Teilabschnitten wurden zwei unterschiedliche Strategien erläutert, um die Funktionalität des Visitors der Obergrammatik im Falle von Sprachvererbung wiederzuverwenden. Einerseits konnte eine Vererbungsbeziehung zwischen den Visitoren in Verbindung mit dem `super`-Aufruf verwendet werden, andererseits ist der Einsatz zweier Visitoren unter Verwendung eines `MultiVisitors` möglich. Wie bereits erläutert wurde, unterscheiden sich beide Lösungen aus technischer Sicht nicht: in beiden Fällen werden die `visit`-Methoden beider Visitoren aufgerufen, auch der Informationsaustausch zwischen beiden ist möglich.

Der einzige Unterschied zwischen beiden Versionen besteht in der Granularität: während bei der Visitorvererbung bei jedem Nichtterminal separat entschieden werden kann und muss, ob die Funktionalität des Visitors der Obergrammatik ausgeführt werden soll, ist bei der Verwendung unabhängiger Visitoren von vornherein festgelegt, dass prinzipiell alle gültigen `visit`-Methoden aufgerufen werden. Für beide Lösungen existieren bei unterschiedlichen Problemen Vor- und Nachteile, ersterer Ansatz bietet typischerweise mehr Flexibilität, der Aufruf der Funktionalität des übergeordneten Visitors muss jedoch an jeder Stelle explizit erfolgen. Der explizite Aufruf ist bei der zweiten Variante nicht nötig, das Maß an Flexibilität ist jedoch verringert.

6.3. Sprachvereinigung

Bei der Sprachvereinigung durch die Sprachfamilienbildung entstehen keine Auswirkungen bezüglich der abstrakten Syntax. Der wesentliche Grund hierfür ist, dass MontiCore jede Einga-

bedatei einzeln verarbeitet und im Falle mehrerer Eingaben absichtlich kein integriertes Modell bildet. Daher werden die Eingaben und insbesondere die ASTs separat gehalten und behandelt.

Zusätzlich sind in den beteiligten Sprachbausteinen einer Sprachfamilie gemäß Abbildung 4.8 separate Workflows registriert, die abhängig vom Typ der Root ausgeführt werden. Da jede Root einen Verweis auf den sprachspezifischen AST besitzt, werden die Workflows auch bei der Integration mehrerer Sprachen in einem Werkzeug nur für die angedachte Sprache ausgeführt.

Innerhalb der Workflows sind die definierten Berechnungen oftmals durch die Verwendung der Visitoren realisiert. Auch hierbei kommt es zu keinen Überschneidungen mit anderen Sprachen bezüglich der abstrakten Syntax. Auch bei nicht-Visitor-basierten Berechnungen auf der abstrakten Syntax entstehen keine Überschneidungen.

6.4. Nichtterminal-Erweiterung

Bei der Nichtterminal-Erweiterung wird ein Nichtterminal einer existierenden Grammatik durch ein Nichtterminal einer anderen Grammatik ergänzt. Die möglichen ASTs der entstehenden Grammatik können somit Instanzen der Nichtterminalklassen der entsprechenden Regeln beider Grammatiken enthalten. Zur Umsetzung bezüglich der konkreten Syntax für das Java-SQL-Beispiel wurden in Kapitel 5.4 verschiedene Vorgehensweisen vorgeschlagen, die auf der Ebene der abstrakten Syntax Unterschiede aufweisen. Die einzelnen Auswirkungen werden dabei im Folgenden vorgestellt.

6.4.1. Umsetzung durch Mehrfachvererbung

Bei der Umsetzung durch Mehrfachvererbung besteht die Definition der komponierten Sprache lediglich aus einer Vererbungsbeziehung zu beiden involvierten Grammatiken und aus der Aussage, dass ein Select-Statement auch als Expression verwendet werden kann. Die abstrakte Syntax der Grammatik aus Abbildung 5.10 ergibt sich daher wie in Abbildung 6.9 dargestellt.

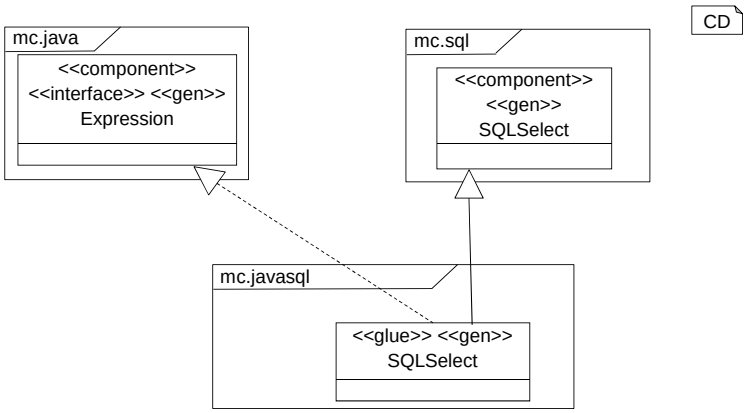


Abbildung 6.9.: Resultierende abstrakte Syntax

Hierbei entstehen die zu Beginn des Kapitels beschriebenen Auswirkungen: in der Subgrammatik werden nur die Anteile neu generiert, die Änderungen unterliegen. Im Beispiel betrifft dies

nur die Klasse `SQLSelect`, da für diese Klasse eine neue `implements`-Beziehung definiert wurde. Zusätzlich entsteht eine Vererbungsbeziehung zur gleichnamigen Nichtterminalklasse der ursprünglichen Grammatik, da die Regel erweitert wurde. Der Parser selbst produziert nur Instanzen der Klasse der Subgrammatik, durch die Vererbungsbeziehung können die Algorithmen, insbesondere die Visitoren der Supergrammatik wiederverwendet werden.

Auffallend an diesem Beispiel ist, dass das Nichtterminal `Expression` der Java-Grammatik als Interface definiert und demnach als möglicher Erweiterungspunkt der Sprache vorausgesehen wurde. Nur dadurch ist es möglich, die beschriebene Vorgehensweise zu verwenden. Falls `Expression` als Klassenproduktion definiert worden wäre, wäre statt der `implements`-Beziehung in der Grammatik die `extends`-Beziehung zu verwenden. Von der Seite der konkreten Syntax würden keine Unterschiede entstehen, von Seiten der abstrakten Syntax wäre jedoch eine Vererbungsbeziehung zur `Expression` nötig. Aufgrund der fehlenden Mehrfachvererbung in Java kann dieses Vorgehen jedoch nicht umgesetzt werden. Als Lösungsstrategie wurde eine Änderung der Subgrammatik gemäß Abbildung 5.11 vorgeschlagen. Abbildung 6.10 zeigt die entstehende abstrakte Syntax.

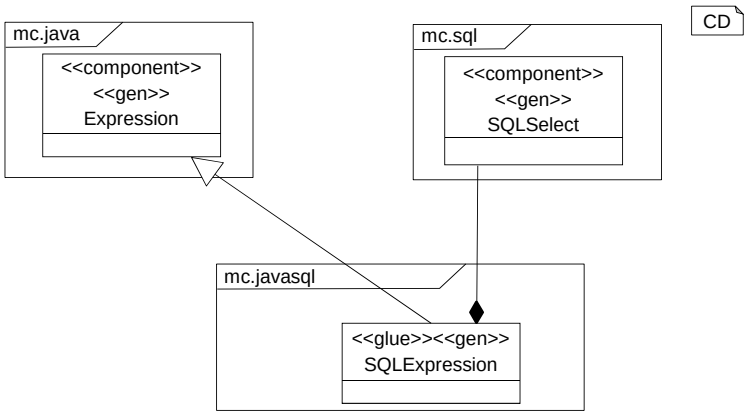


Abbildung 6.10.: Verwendung von Regel- und Grammatikvererbung für Nichtterminal-Erweiterung

Durch die Verwendung der `extends`-Beziehung zur `Expression` der Java-Grammatik entsteht auch in der abstrakten Syntax eine Vererbungsbeziehung zwischen den beteiligten Klassen. Eine Vererbungsbeziehung zum `SQLSelect` ist nicht mehr möglich, vielmehr entsteht durch die Referenz auf der rechten Regelseite eine Kompositionsbeziehung.

Bei der Benutzung der Visitoren entstehen unterschiedliche Effekte: Visitoren für die Sprache Java besuchen auch den neu eingeführten Knotentypen `SQLExpression`, da eine Vererbungsbeziehung zur Basisklasse der Java-Grammatik besteht - es muss lediglich ein `Inheritance-Visitor` verwendet werden. Visitoren der Sprache SQL besuchen auch die Knoten der Select-Anweisung, da diese unverändert übernommen und nicht neu generiert werden. Der einzige Unterschied besteht im Zugriff auf die Eigenschaften des Select-Statements beim Besuch der neu eingeführten Knotentypen: in der ersten Version können die Eigenschaften direkt im Knoten (`mc.javasql.SQLSelect`) abgefragt werden, in der zweiten Version muss eine Abfrage über die Nutzung der Kompositionsbeziehung von `mc.javasql.SQLExpression` nach

`mc.sql.SQLSelect` erfolgen.

6.4.2. Umsetzung durch Einbettung

Externe Nichtterminale als zusätzliche Alternative

Bei der Verwendung der externen Nichtterminale als zusätzliche Alternative entstehen ähnliche Strukturen wie bei der Verwendung der Regelvererbung im vorigen Abschnitt. Unterschiedlich hierbei ist jedoch, dass die Kompositionsbeziehung nicht mit `SQLSelect` getypt ist, sondern mit der Basisklasse `ASTNode`, da bei der Verwendung der Einbettung beliebige Sprachen und ASTs angedacht sind. Bei der Verwendung der Visitoren ist diese Eigenschaft jedoch weniger relevant, da diese auf den konkreten Typ des Knotens zur Laufzeit reagieren, lediglich die Navigation von einer `Expression` zum `SQLSelect` ist nicht getypt möglich.

Im Gegensatz zur reinen Einbettung ist das Ziel der Nichtterminal-Erweiterung die feste Verbindung zweier Sprachen. Daher kann die Grammatik so verändert werden, dass die Typisierung explizit vom Sprachentwickler vorgegeben wird. Abbildung 6.11 zeigt die Vorgehensweise.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javasql;
2
3 grammar JavaSQL extends mc.Java{
4
5   external Extension /mc.sql.SQLSelect;
6
7   Expression = ...  //Wiederholung alte Definition
8                | Extension;
```

Abbildung 6.11.: Beeinflussung der abstrakten Syntax als Erweiterung von Abbildung 5.15

Diese Grammatik entspricht der im Abschnitt der konkreten Syntax vorgestellten Grammatik aus Abbildung 5.15. Zusätzlich wurde nur die Typisierung des externen Nichtterminals in Zeile 5 vorgenommen und auf den Typ `SQLSelect` gesetzt. Daher ergibt sich die abstrakte Syntax gemäß Abbildung 6.12

Einführung zusätzlicher Regelvererbung

Im Gegensatz zur vorangegangenen Technik wird in dieser Version eine Alternative durch die Verwendung einer expliziten Regelvererbung gemäß Abbildung 5.18 eingeführt. Zusätzlich kann auch hier der Typ des externen Nichtterminals durch explizite Angabe des erwarteten Typs wie im vorigen Beispiel weiter eingeschränkt werden. Insgesamt ergibt sich hierdurch die gleiche abstrakte Syntax wie in Abbildung 6.10.

6.4.3. Verwendung der Visitoren

Die vorgestellten Lösungsstrategien unterscheiden sich grundlegend in der resultierenden abstrakten Syntax. In allen Fällen entstanden Subklassen des zu erweiternden Nichtterminals `Ex-`

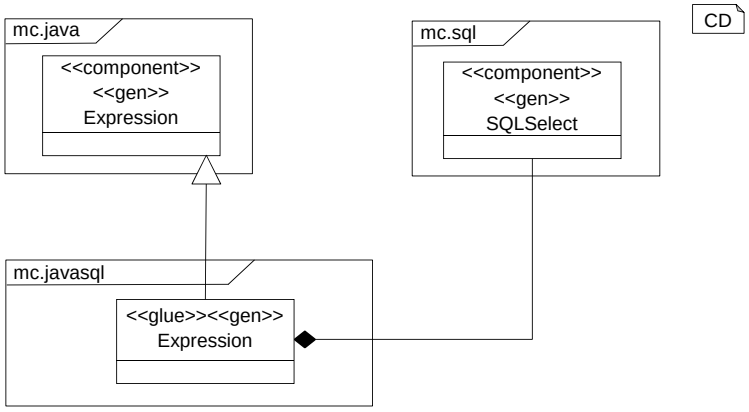


Abbildung 6.12.: Resultierende abstrakte Syntax

pression, bzw. Klassen, die das Interface `Expression` implementieren. Vom erweiternden Nichtterminal `SQLSelect` entstanden teilweise Subklassen, teilweise wurden jedoch auch Kompositionsbeziehungen zur Originalklasse erstellt und damit der AST um weitere Objekte ergänzt.

Die Auswirkungen der veränderten abstrakten Syntax auf die Visitoren sind hierbei je nach verwendeter Strategie unterschiedlich: Bei Subklassenbildung muss ein `InheritanceVisitor` verwendet werden, damit weiterhin die korrekten Methoden aufgerufen werden. Die Erstellung der Kompositionsbeziehung hat keine Auswirkungen auf die Verwendung der Visitoren, da keine neuen AST-Klassen erstellt werden und somit weiterhin nur Instanzen der Originalklasse existieren. Zusätzlich ist die Möglichkeit zur Veränderung der Berechnungen im Visitor für existierende Klassen gemäß Abschnitt 6.2.3 zu beachten: einerseits können Subklassen der existierenden Visitoren mit neuer Funktionalität und der eventuellen Verwendung des `super`-Konstrukts erstellt und verwendet werden, andererseits kann die Erstellung und Einbindung neuer Visitoren für die neu angelegten Klassen erfolgen. Hierbei kann zusätzlich ein `MultiVisitor` zur Einbindung der Originalfunktionalität verwendet werden.

6.5. Nichtterminal-Ersetzung

Die Nichtterminal-Ersetzung kann prinzipiell durch zwei grundlegende Ansätze umgesetzt werden: Einerseits kann Mehrfachvererbung und andererseits Einbettung verwendet werden. Wie im Folgenden dargelegt wird, ergeben sich bei beiden Strategien bezüglich der abstrakten Syntax keine signifikanten Unterschiede.

Umsetzung durch Mehrfachvererbung

Bei der Umsetzung durch Mehrfachvererbung wird zunächst das zu ersetzende Nichtterminal überschrieben und anschließend auf das ersetzende Nichtterminal abgeleitet. Hierdurch ergibt sich für das Beispiel der Automaten mit einer neuen Aktionssprache die in Abbildung 6.14 dargestellte abstrakte Syntax.

Ziel	Lösung
Komplette Wiederverwendung alter Funktionalität	• Einsatz eines `InheritanceVisitors`. • Registrierung der unveränderten konkreten Visitoren.
Änderung der Funktionalität für alle Instanzen des erweiterten Nichtterminals	• Einsatz eines `InheritanceVisitors`. • Registrierung des unveränderten konkreten Visitors der Grammatik des erweiternden Nichtterminals. • Einsatz des Musters aus Abbildung 6.6 oder aus Abbildung 6.7 für den konkreten Visitor der Grammatik des erweiterten Nichtterminals.

Tabelle 6.13.: Verwendung der Visitoren bei Nichtterminal-Erweiterung

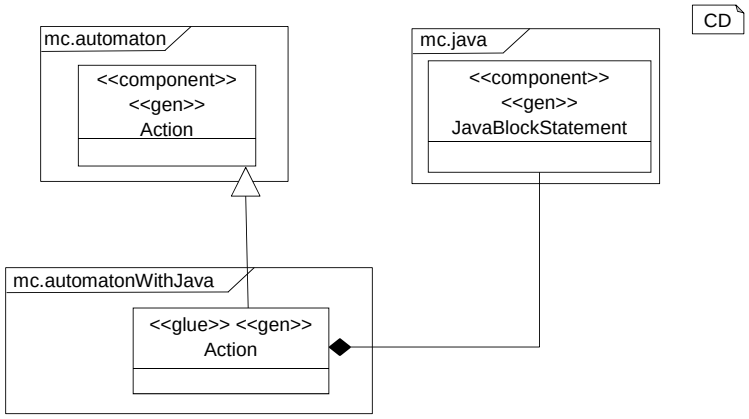


Abbildung 6.14.: Resultierende abstrakte Syntax bei Mehrfachvererbung der Grammatik

Wie bereits bei der Nichtterminal-Erweiterung durch Verwendung von Regel- und Grammatikvererbung entsteht auch hierbei eine neue Nichtterminal-Klasse (`mc.automatonWithJava.Action`), welche von der entsprechenden Nichtterminal-Klasse der Supergrammatik der Automaten erbt. Der Grund hierfür besteht jedoch nicht in der Verwendung der Regelvererbung, sondern im Überschreiben der Originalregel. Durch die Ableitung zu `JavaBlockStatement` entsteht zusätzlich eine Kompositionsbeziehung, neue Klassen werden hierdurch nicht generiert.

Umsetzung durch Einbettung

Auch bei der Umsetzung durch Einbettung wird zunächst das zu ersetzende Nichtterminal überschrieben, zusätzlich wird jedoch ein externes Nichtterminal eingeführt, auf welches das überschriebene Nichtterminal reduziert wird. Wie bereits bei der Nichtterminal-Erweiterung kann

der Typ des externen Nichtterminals durch explizite Angabe des erwarteten Typs eingeschränkt werden. Wird dabei der Typ des externen Nichtterminals genau auf den Typ des ersetzenden Knotens festgelegt, unterscheidet sich die resultierende abstrakte Syntax nicht von der aus der vorangegangenen Umsetzungsstrategie. Die Strategien unterscheiden sich demnach lediglich durch die Entkopplung der Lexer und Parser, also bezüglich der konkreten Syntax.

Verwendung der Visitoren

Im Gegensatz zur Nichtterminal-Erweiterung werden bei der Nichtterminal-Ersetzung prinzipiell keine Instanzen der Nichtterminal-Klasse des ersetzten Nichtterminals produziert. Vielmehr entsteht in beiden Umsetzungsstrategien eine Subklasse mit einer Kompositionsbeziehung zum ersetzenden Nichtterminal. Die Behandlung der abstrakten Syntax durch Visitoren kann dabei wie folgt umgesetzt werden: die Visitoren beider Grammatiken können zunächst unverändert wiederverwendet werden. Ist eine Wiederverwendung der Funktionalität für das zu ersetzende Nichtterminal erwünscht, ist ein `InheritanceVisitor` zu verwenden, soll die Funktionalität neu erstellt werden, kann entweder ein neuer Visitor für die neu entstandene Subklasse erstellt werden oder es wird vom vorhandenen Visitor der Grammatik geerbt und eine neue `visit`-Methode für die neue Knotenklasse hinzugefügt. Insgesamt ergeben sich die Verwendungsmöglichkeiten gemäß Tabelle 6.15.

Ziel	Lösung
Komplette Wiederverwendung alter Funktionalität	• Einsatz eines `InheritanceVisitors`. • Registrierung der unveränderten konkreten Visitoren.
Änderung der Funktionalität	• Einsatz eines `Visitors`. • Registrierung des unveränderten konkreten Visitors der Grammatik des ersetzenden Nichtterminals. • Einsatz des Musters aus Abbildung 6.6 oder aus Abbildung 6.7 für den konkreten Visitor der Grammatik des ersetzten Nichtterminals.

Tabelle 6.15.: Verwendung der Visitoren bei Nichtterminal-Ersetzung

6.6. Einbettung einer einzelnen Sprache

Bei der Einbettung einer einzelnen Sprache entstehen bezüglich der abstrakten Syntax die gleichen Strukturen wie bei der Umsetzung der Nichtterminal-Erweiterung und der Nichtterminal-Ersetzung durch Einbettung. Wesentlicher Unterschied ist jedoch, dass bei der Einbettung einer einzelnen Sprache die beteiligten Grammatiken nicht verändert werden, sondern lediglich ein übergeordneter Parser entwickelt bzw. durch die Angabe einer Sprachdatei generiert wird.

Da die beteiligten Grammatiken separat entwickelt wurden, kann es hierdurch zu Problematiken bezüglich der Typisierung an den Übergangsstellen der Sprachen kommen. Es ist möglich, dass die einbettende Grammatik einen Typ für das externe Nichtterminal festlegt, der nicht dem Typ des Wurzelknotens der eingebetteten Grammatik entspricht. Somit kann ein AST der eingebetteten Sprache nicht in die entsprechende Stelle eines AST der einbettenden Sprache eingesetzt werden. Als Lösung muss eine Anpassung der Typisierung erfolgen.

Die Anpassung des Wurzelknotens der eingebetteten Sprache auf einen vorgegebenen Typ kann dabei in einer Subgrammatik unter Zuhilfenahme des `astimplements`-Konstrukts erfolgen. Unter der Annahme, dass die Automatengrammatik für das externe Nichtterminal `Action` den Typ `mc.automaton.IBasicAction` vorgibt, muss die Java-Grammatik wie in Abbildung 6.16 erweitert werden.

MontiCore-Grammatik «hw»

Glue

```
1 package mc.javaForEmbedding;
2
3 grammar JavaForEmbedding extends mc.Java{
4
5   BlockStatement astimplements /mc.automaton.IBasicAction;
6
7 }
```

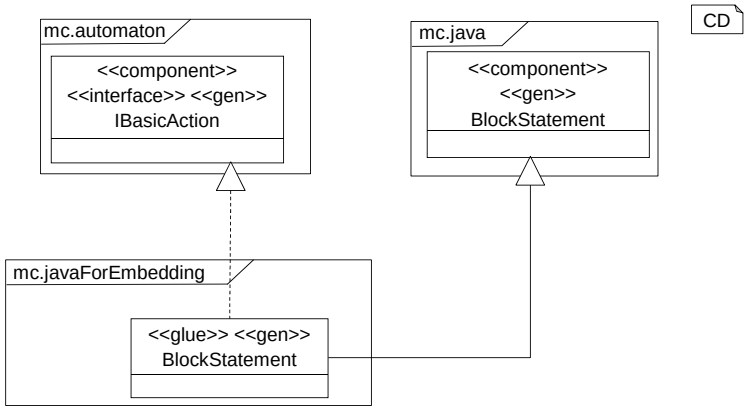


Abbildung 6.16.: Erweiterung der Java-Grammatik zur Einbettung

Die Wiederholung des `BlockStatements` unter Auslassung der rechten Regelseite bewirkt dabei keine Änderung bezüglich der konkreten Syntax. Auf der Seite der abstrakten Syntax wird jedoch eine neue Subklasse generiert, die zusätzlich in einer `implements`-Beziehung zum vorgegebenen Interface der Automaten-Grammatik steht. Eventuell vom Interface vorgegebene Methoden können durch die direkte Angabe der Methode innerhalb der Grammatik [GKR+06] eingebunden werden.

Durch die vorgeschlagene Vorgehensweise kann das `BlockStatement` der Subgrammatik in die Automaten-Grammatik eingebettet werden. In der resultierenden abstrakten Syntax entsteht hierdurch eine Kompositionsbeziehung, welche zwar mit `IBasicAction` getypt ist,

die Knoten in konkreten ASTs sind jedoch immer vom Typ `BlockStatement` des Paketes `mc.javaForEmbedding`. Zusätzlich werden hierdurch die von der Automatengrammatik in Form des Interfaces vorgegebenen Anforderungen an eingebettete Sprachen erfüllt.

Verwendung der Visitoren

Im Szenario der Einbettung einer einzelnen Sprache sind die Visitoren für die beteiligten Grammatiken voneinander unabhängig und können unverändert wiederverwendet werden. Hierbei spielt es eine zentrale Rolle, dass im Normalfall keine Redefinition von Produktionen erfolgt und daher keine Subklassen existierender Nichtterminal-Klassen entstehen. Lediglich bei der beschriebenen Problematik der vorgegebenen Typisierung externer Nichtterminale müssen Produktionen insoweit redefiniert werden, dass sie ein gegebenes Interface implementieren. In diesem Fall ist ein `InheritanceVisitor` zu verwenden. Sollen Funktionalitäten der bestehenden Visitoren verändert werden, muss gemäß Abschnitt 6.2.3 verfahren werden.

6.7. Einbettung von Sprachalternativen

Im Gegensatz zum vorangegangenen Szenario werden bei der Einbettung von Sprachalternativen zur Konfigurationszeit alle möglichen verwendbaren Sprachen festgelegt, die Auswahl der verwendeten Sprache erfolgt jedoch dynamisch während der Erstellung des Modells. Nichtsdestotrotz greifen hierbei die gleichen Mechanismen bezüglich der abstrakten Syntax wie bei der Einbettung einer einzelnen Sprache. Insbesondere kann auch hier die Problematik der festgelegten Schnittstelle des externen Nichtterminals auftreten, die Lösungsstrategie ergibt sich dabei ebenfalls durch Grammatikvererbung und Verwendung des `astimplements`-Konstrukts. Diese Strategie muss jedoch für alle möglichen eingebetteten Sprachen angewandt werden. Zusätzlich ist durch die Verwendung mehrerer eingebetteter Sprachen der Typ des Knotens an der Stelle der Einbettung je nach aktuell verwendeter Sprache unterschiedlich.

Verwendung der Visitoren

Die Dynamik der eingebetteten Sprache hat keinen Einfluss auf die Strukturen der Visitoren, da der Aufruf der korrekten `visit`-Methode im selben Maße dynamisch erfolgt. Daher ergeben sich bezüglich der Verwendung der Visitoren keine signifikanten Unterschiede zur Einbettung einer einzelnen Sprache.

6.8. Entwicklung einer erweiterbaren Sprache

Bei der Entwicklung einer erweiterbaren Sprache lassen sich verschiedene Richtlinien zur Unterstützung der Kompositionalität identifizieren. Diese Richtlinien stellen dabei ein Fortsetzung der im entsprechenden Abschnitt 5.8 der konkreten Syntax aufgestellten Empfehlungen dar.

Richtlinie 7: Verwendung von Schnittstellenproduktionen als Erweiterungspunkte

Wie im Abschnitt 6.4 dargestellt, bieten Schnittstellenproduktionen eine bessere Möglichkeit zur Erweiterung als Klassenproduktionen. Im Szenario der Nichtterminal-Erweiterung können diese Schnittstellenproduktionen durch Verwendung der `implements`-Anweisung einfach um ein weiteres Nichtterminal erweitert werden, bei Klassenproduktionen ist die Einführung eines neuen Nichtterminals notwendig. Daher bieten sich Schnittstellenproduktionen an allen Stellen an, bei denen eine Erweiterung des Nichtterminals denkbar ist.

Richtlinie 8: Vermeidung von Enumerationsproduktionen

Das MontiCore-Grammatikformat ermöglicht die Definition von Enumerationsproduktionen wie in Abbildung 6.17 dargestellt.

MontiCore-Grammatik «hw»

```
1 grammar Enums{
2
3   enum Day = "Monday" | "Tuesday" | "Wednesday" ...
4
5 }
```

Abbildung 6.17.: Beispiel einer Enumerationsproduktion

Wie im Beispiel aufgezeigt, sind Enumerationsproduktionen Disjunktionen von konstanten Werten, auf Ebene der abstrakten Syntax werden sie auf Enumerationen abgeleitet. Da Enumerationen in Java nicht erweiterbar sind, sollten sie innerhalb von MontiCore auch nur im Falle einer definitiv unveränderlichen Auswahl von Alternativen eingesetzt werden. Im Zweifelsfall ist auf Klassen- oder Schnittstellenproduktionen zurückzugreifen.

Richtlinie 9: Verwendung der vorgeschlagenen Visitor-Architektur

Insbesondere bei der Verwendung von Einbettung ist ein Informationsaustausch zwischen Visitoren der beteiligten Sprache nötig. Da die Visitoren jedoch unabhängig voneinander entwickelbar sein sollen, müssen zum Zweck dieses Informationsaustausches explizite Schnittstellen und Adaption eingesetzt werden. Hierfür empfiehlt sich die in Abbildung 6.5 vorgeschlagene und insbesondere auf Kompositionalität ausgelegte Architektur der Visitoren. Dieses Design sollte in jedem Fall angewandt werden, auch wenn eine Kombination mit einer anderen Sprache zur Entwicklungszeit zunächst nicht vorhergesehen wird.

6.9. Verwandte Arbeiten

Zur Definition der abstrakten Syntax einer Sprache eignen sich insbesondere Metamodellierungswerkzeuge. Bekannte Frameworks sind unter anderem die Generic Metamodeling En-

vironment GME [LMB+01], das Eclipse Modeling Framework EMF [BSM+03] oder MetaEdit+ [KT08]. Diese Ansätze erlauben jedoch keine kompositionale Entwicklung der abstrakten Syntax.

Da die meisten Ansätze für die Definition abstrakter Syntax auf zu Klassendiagrammen ähnlichen Notationen beruhen, kann der UML package merge bzw. package import [OMG10c] als theoretische Fundierung für eine kompositionale Entwicklung angesehen werden. Obwohl diese Mechanismen Teil der UML Spezifikation sind, zeigen diverse wissenschaftliche Veröffentlichungen Defizite in Bezug auf Inkonsistenzen, fehlende Teile der Spezifikation oder Ambiguitäten auf (z.B. [DDZ08]). Weiterhin ist noch nicht untersucht worden, ob der package merge Mechanismus unverändert auf die Komposition abstrakter Syntaxen von Sprachen angewandt werden kann oder ob Anpassungen nötig sind.

Ein auf der MOF basierender Ansatz für Metamodellkomposition wird in [WS08] diskutiert. Im Gegensatz zum obigen UML package merge/package import wird dabei speziell auf eine abstrakte Syntax abgezielt und explizit Schnittstellen zwischen Sprachen vorgestellt. Obwohl der vorgestellte Ansatz vielversprechend ist, werden hier nur die Kernkonzepte der MOF verwendet. Eine Erweiterung auf die vollständige MOF bzw. auf alle Konzepte, die für die Definition abstrakter Syntax nötig sind, ist bisher nicht erfolgt. Auch existiert bisher keine Implementierung des Ansatzes.

In [Win00] werden grafische Modellierungsspachen anhand der ihnen zu Grunde liegenden Beschreibungsparadigmen klassifiziert. Die Paradigmen werden in sogenannten Referenz-Metaschemata formalisiert. Die Referenz-Metaschemata können einerseits zu konkreten Sprachen spezialisiert werden. Andererseits können mehrere Metaschemata über gemeinsame Konzepte miteinander integriert werden, um so die Basis für eine problemübergreifende Modellierungssprache zu bilden. Der Fokus dieser Arbeit liegt auf der Formalisierung der Referenz-Metaschemata zur Syntaxbeschreibung sowie ihrer Spezialisierung und Komposition. Eine werkzeugtechnische Umsetzung zur kompositionalen Sprachentwicklung existiert nicht.

Eine weitere Vorgehensweise zur Komposition der abstrakten Syntax besteht in der Template Instanziierung [ES06]. Hierbei werden oft auftretende Strukturen in Metamodellen (z.B. Hierarchie) als Templates vorgegeben. Elemente der durch den Nutzer erstellten Metamodelle für eigene Sprachen werden anschließend durch Rollenbeziehungen an das Template zugewiesen. Hierdurch können Teile des Templates durch verschiedene Sprachen realisiert werden.

Ähnlich zu den in dieser Arbeit verwendetem Konzept der Visitoren sind Runabouts [Gro03] und Walkabouts [PJ98]. Beide Versionen sind zwar nicht auf Kompositionalität ausgelegt, da sie aber auf Java-Reflection basieren, können sie als Grundlage für relativ frei kombinierbare Visitoren genutzt werden. Der Zusammenhang zwischen der Komposition der zu besuchenden Elemente und der verwendeten Visitor-Strategie ist aufgrund des unterschiedlichen Fokus der Arbeiten nicht untersucht worden.

6.10. Zusammenfassung

In diesem Kapitel wurden die Kompositionalitätsmechanismen auf Ebene der abstrakten Syntax vorgestellt. Hierzu erfolgte zunächst die Erläuterung, wie die Mechanismen Spracheinbettung, Sprachaggregation und Vererbung bei der Umsetzung berücksichtigt wurden. Da die abstrakte

Syntax Grundlage für die Weiterverarbeitung von Sprachinstanzen ist, wurde zusätzlich diskutiert, wie die Algorithmik in Form von Visitoren an die Kompositionalität angepasst werden kann.

Der von MontiCore zur Verfügung gestellte Visitor-Ansatz unterstützt dabei die Kompositionalität durch verschiedenen Mechanismen: einerseits können konkrete Visitoren für Sprachbestandteile separat entwickelt werden, andererseits ist eine flexible Kombination je nach aktueller Anforderungslage möglich. Zusätzlich wurde eine Architektur vorgeschlagen, die den Informationsaustausch zwischen unabhängig voneinander entwickelten Visitoren ermöglicht.

Aufbauend auf diesen grundlegenden Mechanismen wurden die Szenarien zur kompositionalen Sprachentwicklung untersucht. Hierbei zeigt sich, dass alle Szenarien unterstützt werden, wobei die unterschiedlichen Lösungsstrategien je Szenario teilweise verschieden bezüglich der abstrakten Syntax umgesetzt werden.

Sowohl bezüglich der abstrakten Syntax als auch auf Ebene der Visitoren zeigen sich die verschiedenen Eigenschaften der kompositionalen Entwicklung. Artefakte werden separat voneinander entwickelt bzw. generiert und erst bei der konkreten Kombination unter Einbeziehung von Glue Code kombiniert. Besonders beim Informationsaustausch zwischen den Visitoren wurden Schnittstellen und Adaption als Mittel zur Umsetzung der Kompositionalität verwendet. Weiterhin vereinfacht das standardisierte Design algorithmischer Anwendungen auf Basis der Visitoren eine Kombination unabhängig entwickelter Sprachen.

Insgesamt zeigt sich, dass die bezüglich der konkreten Syntax entwickelten Lösungen in der abstrakten Syntax und der darauf aufbauenden Algorithmik reflektiert werden. Hierdurch können bereits grundlegende Anwendungen auf Basis kompositional entwickelter Sprachen realisiert werden. Um eine umfassende Unterstützung der Kompositionalität bereitzustellen, sind jedoch die in den weiteren Kapiteln behandelten Artefakte einzubeziehen.

Kapitel 7.

Kompositionale Entwicklung von Symboltabellen

In den vorangegangenen Kapiteln wurden Sprachen bezüglich ihrer Eigenschaften auf Ebene der konkreten und abstrakten Syntax behandelt. Zusätzlich hierzu besitzen Sprachen jedoch auch Typsysteme und darauf aufbauende Kontextbedingungen. Dabei bilden Symboltabellen die Grundlage zur Überprüfung der Kontextbedingungen der Sprache [ASU86]. Die Strukturen von Symboltabellen und die Vorgehensweisen zum Aufbau einer konkreten Symboltabelle für ein konkretes Modell stellen jedoch eine recht komplexe Problematik dar, welche zusätzlich durch die Anforderung der Kompositionalität erschwert wird.

Daher erfolgt in diesem Kapitel zunächst eine grundlegende Einführung in die Thematik der Symboltabellen. Im Anschluss wird eine Formalisierung der beteiligten Artefakte sowie der Kompositionalitätsaspekte vorgestellt. Darauf aufbauend wird der gewählte Ansatz aus technischer Sicht beschrieben, anschließend erfolgt wie bereits in den vorangegangenen Kapiteln eine Untersuchung der konkreten Szenarien der kompositionalen Sprachentwicklung. Abschließend wird eine Übersicht zu verwandten Arbeiten gegeben.

7.1. Einführung

Symboltabellen und deren Inhalte sind von wesentlicher Bedeutung für das Verständnis der weiteren Teilkapitel. In der Literatur ist diese Terminologie jedoch oft nicht eindeutig, in manchen Fällen sogar widersprüchlich. Deshalb werden an dieser Stelle zunächst einige Grundbegriffe eingeführt und näher erläutert.

Definition 7.1.1 (Symboltabelle (informell))**.** *Eine Symboltabelle ist eine Datenstruktur zum Speichern und Auflösen von Bezeichnern in einer Sprache. Kernaufgabe besteht im Auflösen von Namen mit dem Ziel, weitere Informationen wie Typ oder Signatur zu diesem Namen zu erhalten.*

Beispiel 7.1.1. *Die Informationen, die ein Symboltabelleneintrag für eine Methode in der Sprache Java zur Verfügung stellt, lässt sich sehr gut anhand der Klasse* `Method` *des Paketes* `java.lang.reflect` *ablesen. Hier können zum Beispiel die Modifikatoren, der Rückgabetyp, die Parametertypen und alle deklarierten Exceptions der Methode abgefragt werden.*

7.1.1. Entries

Ein grundlegendes Konzept von Symboltabellen sind sogenannte *Entries*. Diese stellen die eigentlichen Einträge in der Symboltabelle dar und enthalten spezifische Informationen für das repräsentierte Element des Modells.

Die Informationen, die für ein Modellelement in die Symboltabelle aufgenommen werden, unterscheiden sich je nach *Sorte* des Elementes. Dabei existieren in vielen Sprachen unterschiedliche Sorten: Klassendiagramme weisen als Sorten Klassen, Methoden und Variablen auf, in der Sprache Statecharts werden typischerweise sowohl Zustände als auch die Statecharts an sich als Sorte aufgefasst, da beide in der Symboltabelle aufgenommen und abgefragt werden können.

Definition 7.1.2 (Entry und Sorte (informell))**.** *Ein Entry ist ein Symboltabelleneintrag für ein konkretes Element des Modells. Entries haben dabei eine für die Sorte des Elementes (z.B. Klasse oder Methode in Klassendiagrammen oder Zustand in Statecharts) spezialisierte Form durch die sie in der Lage sind, spezifische Informationen aufzunehmen und wiederzugeben.*

Beispiel 7.1.2. *Ein Entry für eine Variable in Klassendiagrammen enthält den Namen des Typs der Variable und die Modifikatoren der Variable.*

7.1.2. Scopes und Namespaces

Die meisten Sprachen erlauben es, gleiche Namen an verschiedenen Stellen zu deklarieren. Hierbei ist es oft sogar erlaubt, dass ein bereits belegter Name eine neue Bedeutung bekommt, man spricht hier von der *Namensverdeckung*. So ist es in Java erlaubt, einer lokalen Variable in einer Methode den gleichen Namen wie einer Instanzvariablen zu geben. Diese Variable ist dann innerhalb der Methode sichtbar, nach außen hin ist sie jedoch gekapselt und somit nicht sichtbar.

Um diese Namensverdeckung umzusetzen, werden die Konzepte *Scope* und *Namespace* verwendet. Scopes sind dabei die Bereiche, in denen Elemente durch ihren Namen angesprochen werden können [Lou02]. Visuell lassen sie sich wie in Abbildung 7.1 darstellen.

Definition 7.1.3 (Scope)**.** *Der Scope eines Entries ist der Bereich im Modell, in dem das Element des Entries durch die Verwendung seines Namens referenziert werden kann, also sichtbar und nicht verdeckt ist.*

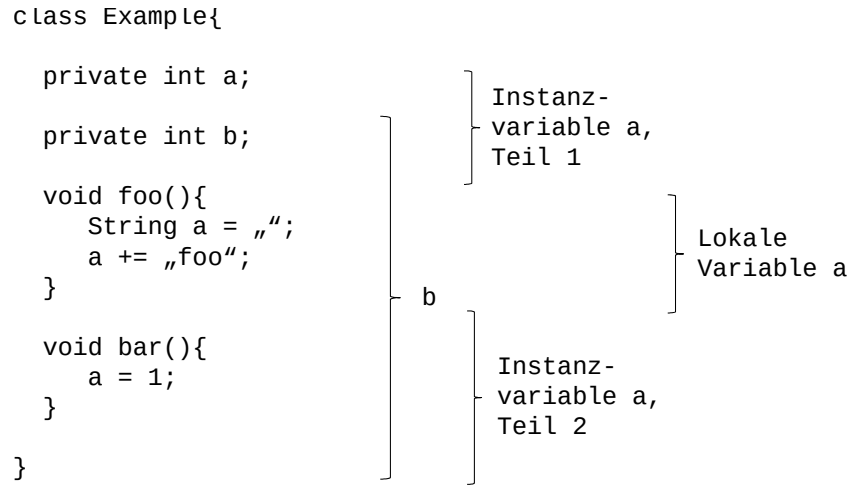


Abbildung 7.1.: Eine einfache Java-Klasse mit Scopes.

Um die Verdeckung von Namen umzusetzen und Entries effizient zu verwalten, werden Modelle in verschiedene Einheiten eingeteilt - die Namespaces [CT04] .

Definition 7.1.4 (Namespace (informell))**.** *Ein Namespace ist ein Abschnitt eines Modells, in dem Namen und zugehörige Entries gemeinsam verwaltet werden. Namespaces sind dabei typischerweise hierarchisch aufgebaut. Zusätzlich kann ein Namespace Namen und Entries von anderen Namespaces importieren (z.B. vom übergeordneten Namespace). Das Erstellen eines Entries für einen Namen im Namespace kann ein anderes Entry mit demselben Namen eines anderen Namespaces verdecken.*

Abbildung 7.2 zeigt eine visuelle Darstellung von Namespaces. Hierbei wird auch die enge Kopplung zwischen der Struktur des Modells und der Namespaces ersichtlich. In der Tat sind die Namespaces Parallelstrukturen zum AST, lediglich die Namespaces auf den obersten Ebenen haben typischerweise keine Entsprechung im AST, sondern bilden Namensräume wie Pakete ab oder definieren einen globalen Namespace.

Beispiel 7.1.3. *Gegeben sei die Java-Klasse aus Abbildung 7.3. Der Java-Parser des MontiCore-Frameworks erstellt daraus den abstrakten Syntaxbaum aus Abbildung 7.4, links*[1]*. Die rechte Seite der Abbildung zeigt die Struktur der entsprechenden Namespaces.*

Die in Definition 7.1.1 erwähnte hierarchische Struktur der Symboltabellen ergibt sich also aus der hierarchischen Struktur der Namespaces. Ein Namespace besitzt demnach eine Symboltabelle, welche die Zuordnung von Namen zu Entries beinhaltet. Wie im Folgenden ausgeführt

[1] Der AST ist hierbei skizzenhaft dargestellt. Zum einen existieren weitere Objekte, zum anderen sind die vorhandenen Objekte teilweise nicht direkt verlinkt, sondern über mehrere Zwischenobjekte. Aus Übersichtsgründen wurde auf die detaillierte Darstellung verzichtet.

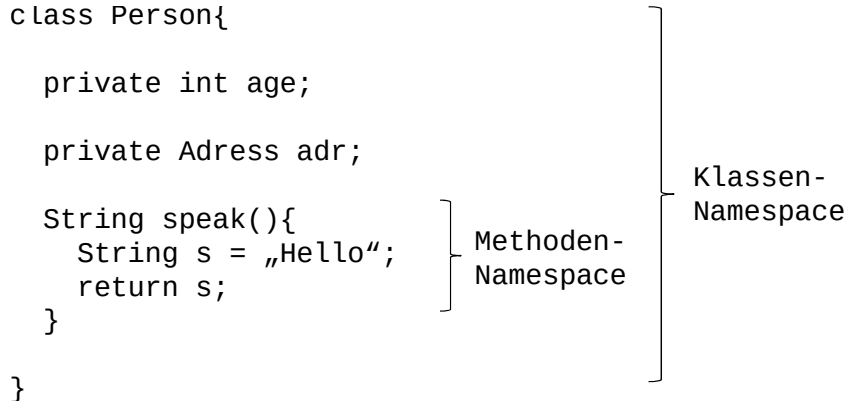


Abbildung 7.2.: Eine einfache Java-Klasse mit Namespaces.

Java-Quellcode

```
1 import java.util.*;
2 import java.io.*;
3
4 class Person{
5
6    void speak(){
7       int x=0;
8       while(x<5){
9          x++;
10         System.out.println("Hello");
11      }
12      while(x>0){
13         x--;
14         System.out.println("World");
15      }
16   }
17
18   void beQuiet(){
19   }
20 }
```

Abbildung 7.3.: Einfache Java-Klasse.

wird, existiert jedoch nicht genau eine Symboltabelle pro Namespace, sondern mehrere, die jeweils unterschiedliche Funktionen einnehmen.

7.1.3. Arten von Symboltabellen

Eine wesentliche Rolle spielen bei der Unterscheidung der Funktionen von Symboltabellen die Verbindungen der Namespaces untereinander. In den meisten Fällen spiegelt sich die Sichtbarkeit in der hierarchischen Struktur wider: Elemente übergeordneter Namespaces sind in unter-

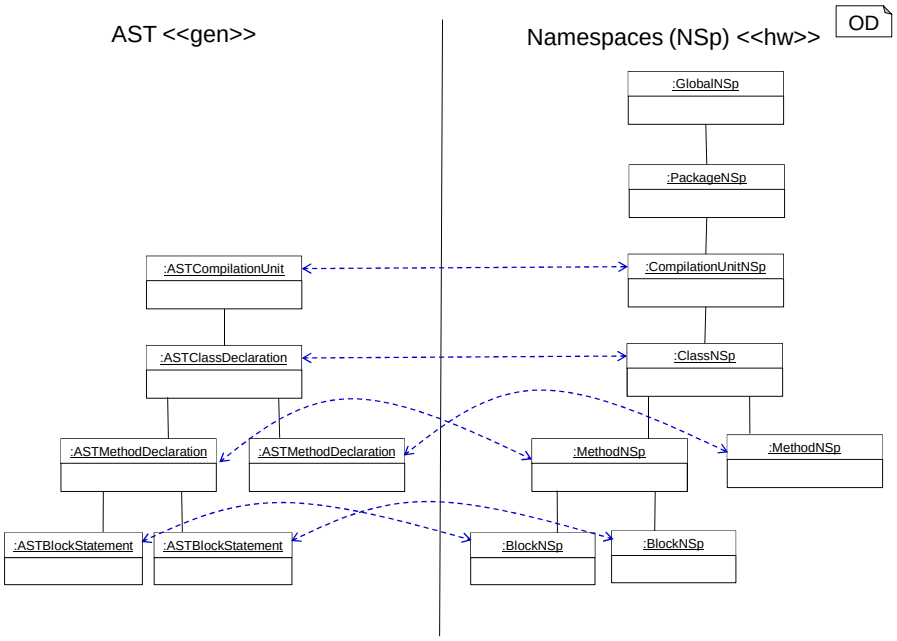


Abbildung 7.4.: Simplifizierte Variante des AST und entsprechende Namespace-Struktur.

geordneten Namespaces sichtbar, sofern diese nicht durch andere Elemente verdeckt werden. So sind Methoden in Java im Klassen-Namespace registriert, Anweisungen im untergeordneten Methoden-Namespace. Über die hierarchische Struktur können die Anweisungen im Methoden-Namespace dann Methoden im Klassen-Namespace referenzieren. Die Wirkung dieser Hierarchie kann daher als *Importieren* von Symboltabellen gesehen werden. Namen werden also immer zuerst in der Symboltabelle des aktuellen Namespace gesucht, werden sie dort nicht gefunden, wird in den *importierten Symboltabellen* weitergesucht.

Neben den importierten Symboltabelle existiert in jedem Namespace eine weitere Form: die *exportierten Symboltabellen*. Diese besagen, welche Einträge nach außen sichtbar sind. Die exportierten Symboltabellen des Namespaces einer Klasse im Klassendiagramm beinhaltet so alle Attribute und Methoden, die von außen sichtbar sind, d.h. von anderen Klassen verwendet werden können. Anzumerken ist hierbei, dass in manchen Sprachen durchaus verschiedene exportierte Symboltabellen existieren können. Im Beispiel der Klassen in Klassendiagrammen existiert beispielsweise eine *protected*-Version, die von Subklassen und Klassen im gleichen Paket importiert wird und und eine *public*-Version, die von allen Klassen importiert wird.

Diese Überlegungen führen unmittelbar zur dritten Art der Symboltabellen: der *gekapselten (encapsulated) Symboltabelle*. Diese Symboltabelle beinhaltet genau die Elemente, die nicht importiert und nicht nach außen weitergegeben werden, sondern nur für die interne Benutzung gedacht sind. Ein Beispiel hierfür sind lokale Variablen in Java-Methoden.

Als letzte Art der Symboltabellen existieren die *weitergeleiteten (forwarded) Symboltabellen*. Diese sind von den importierten Symboltabellen abhängig und beschreiben, welche Entries weiter exportiert werden. Als Beispiel hierfür dient wiederum Java: Eine Klasse importiert alle

nicht-privaten Methoden der Superklasse und exportiert diese an die Außenwelt bzw. an Subklassen. Insgesamt existieren also folgende Arten (engl. kinds) von Symboltabellen für einen Namespace:

- Die gekapselte Symboltabelle beinhaltet Elemente, die nur innerhalb des Namespaces referenziert werden können. Es existiert maximal eine gekapselte Symboltabelle je Namespace.
- Die importierten Symboltabellen beinhalten Elemente, die in anderen Namespaces definiert wurden. Es können mehrere Varianten existieren.
- Die exportierten Symboltabellen beinhalten Elemente, die an die Außenwelt propagiert werden. Es können mehrere Varianten existieren.
- Die weitergeleiteten Symboltabellen beinhalten Elemente, die in anderen Namespaces definiert wurden und an die Außenwelt propagiert werden. Es können mehrere Varianten existieren.

Insgesamt ergibt sich daher die in Abbildung 7.5 zusammengefasste Beziehung zwischen Namespaces, Symboltabellen und Entries.

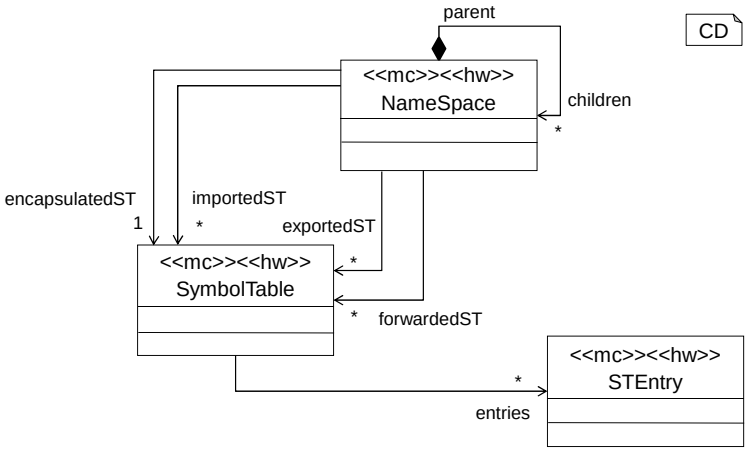


Abbildung 7.5.: Zusammenhänge zwischen Namespaces, Symboltabellen und Entries

7.2. Theoretische Grundlagen

Die im vorigen Teilkapitel erfolgten Erläuterungen sind informeller Natur und können aus diesem Grunde nicht oder nur eingeschränkt als Grundlage weiterer Diskussionen dienen. Daher wird im Folgenden eine formalere Grundlage vorgestellt. Diese formale Grundlage stellt die Infrastrukturen von Namespaces, Symboltabellen und deren Interaktionen sowohl anhand bekannter mathematischer Theorien als auch auf Grundlage von Konzepten des Compilerbaus dar.

Entries

Wie bereits im vorigen Abschnitt beschrieben, bilden Entries das zentrale Element der Symboltabellen und haben eine je nach Sorte des beschriebenen Elementes spezialisierte Form. Sie beinhalten alle für das bezeichnete Element benötigten Informationen.

Definition 7.2.1 (Entry)**.** *Ein Entry ist ein Tupel einer für die Sorte des bezeichneten Elementes spezialisierten Relation. Die beteiligten Mengen der Relation können unter anderem Namen oder andere meist sprachspezifische Mengen zur Darstellung von Eigenschaften des Elementes sein. Die Menge aller Entries wird im Folgenden mit* $\mathbb{E}\mathrm{ntry}$ *bezeichnet.*

Die Namen sind hierbei oftmals als Referenzen auf andere Elemente der Symboltabelle zu sehen. Im Folgenden wird die Menge der Namen mit $\mathbb{N}\mathrm{ame}$ bezeichnet.

Beispiel 7.2.1. *Die Relation für eine Variable in Klassendiagrammen ist*

$$V \subseteq \mathbb{T}\mathrm{ype} \times \wp(\mathbb{M}\mathrm{od})$$

mit

- $\mathbb{T}\mathrm{ype} \subseteq \mathbb{N}\mathrm{ame}$ *die Menge der Typen (als Referenz für den Typ der Variable)*
- $\mathbb{M}\mathrm{od} \subseteq \mathbb{N}\mathrm{ame}$ *die Menge der Modifikatoren (für die Modifikatoren der Variablen)*

Anzumerken ist, dass der Name der Variable nicht Teil des Entries ist. Dieser wird wie später beschrieben, in den Symboltabellen gespeichert. In diesen findet dann eine Zuordnung des Namens zum Entry und somit zu den zusätzlichen Eigenschaften statt.

Um die Entries verschiedener Sorten unterscheiden zu können, wird die Funktion $sort$ verwendet.

Definition 7.2.2 (Sorten und Funktion $sort$)**.** $\mathbb{S}\mathrm{ort}$ *ist eine endliche Menge von Sorten von Entries. Aufbauend hierauf ordnet die Funktion*

$$sort : \mathbb{E}\mathrm{ntry} \to \mathbb{S}\mathrm{ort}$$

jedem Entry die korrekte Sorte zu.

Symboltabelle

Die wesentliche Eigenschaft von Symboltabellen besteht darin, dass sie zu einem gegebenen Namen weitergehende Informationen zur Verfügung stellt. Daher beinhalten Symboltabellen eine einfache Zuordnung von Namen zu den dazugehörigen Entries. Insgesamt ergibt sich daher die folgende Definition:

Definition 7.2.3 (Symboltabelle)**.** *Eine Symboltabelle st ist*

$$st \subseteq \mathbb{N}\text{ame} \times \mathbb{E}\text{ntry}$$

Im Folgenden wird die Menge aller Symboltabellen mit $\mathbb{S}\text{ymtab}$ *notiert. Zusätzlich werden die Symboltabellen in vier disjunkte Teilmengen unterteilt:*

- $\mathbb{S}\text{ymtab}_{exp}$ *die Menge der exportierten Symboltabellen.*
- $\mathbb{S}\text{ymtab}_{enc}$ *die Menge der gekapselten (encapsulated) Symboltabellen.*
- $\mathbb{S}\text{ymtab}_{imp}$ *die Menge der importierten Symboltabellen.*
- $\mathbb{S}\text{ymtab}_{for}$ *die Menge der weitergeleiteten (forwarded) Symboltabellen.*

In den meisten Sprachen ist es möglich, dass zu einem Namen mehrere Symboltabelleneinträge existieren. Diese können einerseits derselben Sorte angehören, ein Beispiel hierfür sind überladene Methoden in Klassendiagrammen. Andererseits können die Einträge auch verschiedenen Sorten angehören, so können in Klassendiagrammen Methoden und Attribute den gleichen Namen haben. Beide Eigenschaften sind durch die obige Definition abgedeckt.

Andererseits ist es auch manchmal wünschenswert, dass unter einem Namen maximal ein Eintrag für eine bestimmte Sorte zulässig ist. So müssen Attribute einer Klasse in Java verschiedene Namen haben. Dies lässt sich durch zusätzliche Beschränkungen ausdrücken, für das gegebene Beispiel verschiedennamiger Attribute etwa durch:

$$\forall (n_1, e_1), (n_2, e_2) \in st : n_1 = n_2 \wedge sort(e_1) = sort(e_2) = \text{„}Attribut\text{“} \Rightarrow e_1 = e_2$$

Grammatiken mit Typisierungsstrukturen

Um Typisierungsstrukturen in Sprachen darzustellen, muss die klassische Beschreibung kontextfreier Grammatiken um zusätzliche Elemente erweitert werden. Als Ausgangsbasis dient hierfür die Definition 7.2.4.

Definition 7.2.4 (Kontextfreie Grammatik)**.** *Eine kontextfreie Grammatik G ist ein Tupel $G = (N, T, P)$ mit*

- N *eine endliche Menge von Nichtterminalsymbolen,*
- T *eine endliche Menge von Terminalsymbolen*
- $P \subseteq N \times (N \cup T)^*$ *Produktionen der Grammatik*

Diese Definition entspricht dabei den in der Standardliteratur existierenden Versionen für kontextfreie Grammatiken. Einziger Unterschied besteht im Fehlen eines expliziten Startsymbols,

da MontiCore-Grammatiken hierüber nicht verfügen. Das Fehlen eines expliziten Startsymbols ist jedoch für die folgenden Betrachtungen nicht relevant.

Aufbauend auf dieser grundlegenden Definition können Grammatiken um Typisierungsstrukturen erweitert werden. Hierbei werden sowohl die genutzten Sorten als auch alle möglichen Symboltabelleneinträge eingeführt. Aufgrund der Betrachtungen des letzten Teilkapitels, insbesondere der Betrachtungen zur Parallelität von AST und Namespace-Strukturen werden weiterhin diejenigen Nichtterminale ausgezeichnet, die einen neuen Namespace aufspannen.

Definition 7.2.5 (Grammatik mit Typisierungsstrukturen)**.** *Eine Grammatik mit Typisierungsstrukturen ist ein Tupel* $G_T = (G, \mathbb{S}\mathrm{ort}_G, \mathbb{E}\mathrm{ntry}_G, NT_s)$ *mit*

- G *eine Grammatik mit den Nichtterminalsymbolen* N
- $\mathbb{S}\mathrm{ort}_G \subseteq \mathbb{S}\mathrm{ort}$ *eine Menge von Sorten*
- $\mathbb{E}\mathrm{ntry}_E \subseteq \mathbb{E}\mathrm{ntry}$ *die Menge aller möglichen Symboltabelleneinträge (Entries)*
- $NT_s \subseteq N$ *eine ausgezeichnete Menge von Nichtterminalsymbolen, die einen neuen Namespace aufspannen.*

Beispiel 7.2.2. *Klassendiagramme enthalten die Sorten „Typ“, „Methode“ und „Attribut“. Die Menge der Entries besteht aus allen möglichen Symboltabelleneinträgen für Typen, Methoden und Attributen. Die Nichtterminalsymbole, welche einen neuen Namespace aufspannen, sind die Nichtterminalsymbole für eine Klassen-, Interface- und Enumerationsdeklarationen.*

Namespaces

Aufbauend auf den Symboltabellen können nun Namespaces definiert werden. Diese beinhalten die im vorigen Teilkapitel besprochenen gekapselten, importierten, exportierten, und weitergeleiteten Symboltabellen.

Definition 7.2.6 (Namespace)**.** *Ein Namespace* ns *ist ein Tupel* $ns = (st_{enc}, st_{imp}, st_{exp}, st_{for})$ *mit*

- $st_{enc} \in \mathbb{S}\mathrm{ymtab}_{enc}$ *die gekapselte Symboltabelle*
- $st_{imp} \in \wp(\mathbb{S}\mathrm{ymtab}_{imp})$ *die importierten Symboltabellen*
- $st_{exp} \in \wp(\mathbb{S}\mathrm{ymtab}_{exp})$ *die exportierten Symboltabellen*
- $st_{for} \in \wp(\mathbb{S}\mathrm{ymtab}_{for})$ *die weitergeleiteten Symboltabellen*

Die Menge der Namespaces wird im Folgenden mit $\mathbb{N}\mathrm{sp}$ *bezeichnet.*

An dieser Stelle sind alle nötigen Strukturen der Symboltabellen definiert worden. Darauf aufbauend wird nun untersucht, wie die Namespace-Struktur für einen konkreten AST aufgebaut wird. Hierzu wird die Technik der attributierten Grammatiken verwendet.

7.2.1. Attributierte Grammatiken

Attributgrammatiken [Knu68] bilden ein Mittel, um Attributflüsse innerhalb von abstrakten Syntaxbäumen zu beschreiben. Hierbei werden Grammatiksymbole (Terminale und Nichtterminale) um sogenannte Attribute ergänzt und entsprechende Berechnungsvorschriften an den Grammatikregeln angegeben. Dabei kann zwischen zwei Arten von Attributen unterschieden werden: inherite und synthetisierte Attribute. Im ersten Fall befindet sich die Berechnungsvorschrift an Regeln, in denen das Symbol auf der rechten Seite erscheint, im zweiten Fall an Regeln, in denen das Symbol auf der linken Seite ist. Intuitiv hängen also synthetisierte Attribute von Kindknoten ab, inherite hingegen von Vater- und/oder Geschwisterknoten.

Insgesamt ergibt sich folgende Definition für Attributgrammatiken [GE99]:

Definition 7.2.7 (Attributgrammatik)**.** *Sei* $\mathbb{A}\mathrm{tt}$ *die Menge der Attribute. Eine Attributgrammatik ist ein Tupel* (G_T, A_G, R_G) *mit*

- $G_T = (G, \mathbb{S}\mathrm{ort}_G, \mathbb{E}\mathrm{ntry}_G, NT_s)$ *eine Grammatik mit Typisierungsstrukturen mit* $G = (N, T, P)$ *eine Grammatik,*
- $A_G = ((N \cup T) \times \mathbb{A}\mathrm{tt})$ *eine Zuordnung von Grammatiksymbolen zu deren Attributen und*
- R_G *die Menge der Berechnungsvorschriften der Form*

$$X_i.a_i = f(X_j.a_j, ..., X_k.a_k)$$

für eine Regel der Grammatik

$$X_0 \to X_1 X_2 ... X_n$$

wobei $\forall x \in \{X_i, X_j, ..., X_k\} : x \in \{X_0, X_1, ..., X_n\}$ *und* $(X_i, a_i) \in A_G \wedge \forall r \in \{j..k\} : (X_r, a_r) \in A_G$.

Für jedes Attribut kann die Menge aller Regeln in drei Teilmengen unterschieden werden: die Menge der nutzenden Regeln verwendet das Attribut auf der rechten Seite, die Menge der zuweisenden Regeln benutzt es auf der linken Seite[2] und schließlich die Menge der Fremdregeln, die das Attribut nicht zuweisen oder nutzen.

[2] Es sei angemerkt, dass keine Regel sowohl zuweisend als auch nutzend sein kann, da sonst ein Zyklus bei der Attributberechnung entsteht. Auf eine formale Definition dieser Eigenschaft soll an dieser Stelle jedoch verzichtet werden.

Definition 7.2.8 (Nutzende, zuweisende und Fremdregeln)**.** *Sei (G_T, A_G, R_G) eine Attributgrammatik. Für ein Attribut a des Symbols $s \in (N \cup T)$ ist*

- $\mathbb{Z}_{s,a} = \{r \in R_G | r = (s.a = f(x_1.a_1, ..., x_n.a_n))\}$ *die Menge der zuweisenden Regeln,*
- $\mathbb{N}_{s,a} = \{r \in R_G | r = (x_0.a_0 = f(x_1.a_1, ..., s.a, ..., x_n.a_n))\}$ *die Menge der nutzenden Regeln und*
- $\mathbb{F}_{s,a} = R_G \backslash (\mathbb{Z}_{s,a} \cup \mathbb{N}_{s,a})$ *die Menge der Fremdregeln.*

7.2.2. Assoziierte Namespaces

Um in späteren Schritten Symboltabellen-Einträge in den Namespaces bzw. deren Symboltabellen zu erstellen, muss jeder Knoten in einem AST den jeweils umgebenden Namespace - auch als *assoziierter* Namespace bezeichnet - kennen. Die Information über den assoziierten Namespace ist dabei insbesondere bei Knoten, die keinen neuen Namespace aufbauen, vom Kontext - d.h. von übergeordneten Knoten und damit der Hierarchie des ASTs - abhängig und kann daher als inherites Attribut dargestellt werden. Insgesamt ergibt sich hierbei folgende allgemeine Berechnungsvorschrift:

Definition 7.2.9 (Assoziierte Namespaces)**.** *Sei $G_T = (G, \mathbb{S}\mathrm{ort}_G, \mathbb{E}\mathrm{ntry}_G, NT_s)$ eine Grammatik mit Typisierungsstrukturen, wobei $G = (N, T, P)$ ist. Jedem Grammatiksymbol (Terminal oder Nichtterminal) wird anschließend der assoziierte Namespace im Attribut $assNsp$ zugeordnet. Die Berechnungsvorschrift für den Attributwert $s.assNsp$ für ein Grammatiksymbol s ist dabei wie folgt:*

$$s.assNsp = \begin{cases} newNSP & \forall (s, r_1...r_n) \in P \text{ falls } s \in NT_s \\ l.assNSP & \forall (l, r_1...s...r_n) \in P \text{ sonst} \end{cases}$$

Hierbei bezeichnet $newNSP$ die Erstellung eines neuen Namespaces, $r_1, ..., r_n, l$ bezeichnen Elemente aus $N \cup T$.

Die Auswirkungen dieser Attributberechnungen sind unmittelbar ersichtlich: falls ein Symbol einen neuen Namespace aufbaut, wird dem Attribut $assNsp$ ein neuer Namespace zugeordnet (Zeile 1). Falls das Symbol keinen neuen Namespace aufbaut, wird der Namespace des Vaterknotens übernommen (Zeile 2). Nach dieser Berechnung ist damit jedem Symbol der assoziierte Namespace bekannt.

Durch diese Definition wird die bereits in Abbildung 7.4 dargstellte Parallelität von AST und Namespaces unterstrichen. Da sich die Namespace-Struktur aufgrund der ausgezeichneten Menge von Namespace-aufspannenden Nichtterminalen ergibt, orientiert sich die Struktur des Namespace-Baumes genau an der Struktur des gegebenen ASTs.

7.2.3. Eintragungen in Symboltabellen

Im nächsten Schritt des Symboltabellenaufbaus werden die eigentlichen Symboltabelleneinträge in den entsprechenden Symboltabellen registriert. Hierzu wird zunächst der assoziierte Namespace anhand des oben beschriebenen Attributs $assNsp$ ermittelt, anschließend wird das Symbol in eine oder mehrere Symboltabellen eingetragen. In welchen Symboltabellen ein Entry registriert wird, hängt dabei stark von der verwendeten Sprache, von den Eigenschaften des Elementes und von den Beziehungen der Symboltabellen untereinander ab.

Beispiel 7.2.3. *Betrachtet wird Java und der Eintrag eines Symbols für eine Instanzvariable*

```
protected int age;
```

Hierbei lassen sich folgende Betrachtungen machen:

1. *Der Symboltabelleneintrag für die Variable muss im Klassen-Namespace erfolgen. Dies ist durch den assoziierten Namespace gesichert.*
2. *Das Symbol muss aufgrund der Eigenschaften der Variable in die protected-Symboltabelle eingetragen werden.*
3. *Aufgrund der Sichtbarkeitsregeln von Java muss das Symbol zusätzlich in die private und die default-Symboltabelle eingetragen werden*[3].

7.2.4. Namespace-Grammatiken

In den bisherigen Ausführungen wurden immer die kompletten Grammatiken mit allen Regeln betrachtet. Um den Informationsfluss zwischen den Namespaces zu beschreiben, reicht jedoch eine reduzierte Form der Ursprungsgrammatik aus. Diese abstrahiert von Regeln und Regelanteilen, die nicht zur Bestimmung des gültigen Namespace-Baumes beitragen. In dieser simplifizierten Version der Grammatik können anschließend die Beziehungen der Symboltabellen und die Weitergabe von Symboltabelleneinträgen kompakter und verständlicher dargestellt werden.

Beispiel 7.2.4. *Gegeben sei die Grammatik für hierarchische Automaten aus Abbildung 7.6, links. Unter der Annahme, dass nur der Automat selbst und alle Zustände neue Namespaces aufspannen, besteht die Namespace-Grammatik nur aus den beiden Regeln aus Abbildung 7.6, rechts.*

Wie in diesem Beispiel deutlich wird, dienen die Namespace-Grammatiken nicht dem Parsen bzw. dem Erkennen von Worten, sondern beschreiben lediglich den möglichen Aufbau von Namespace-Bäumen. Aus diesen Gründen existieren weder Terminale, noch gelten andere aus dem Compilerbau bekannte Einschränkungen wie Eindeutigkeit oder LL(k)-Eigenschaft.

Aufbauend auf einer solchen Namespace-Grammatik lässt sich das Verhalten der verschiedenen Symboltabellenarten (gekapselt, importiert, exportiert, weitergeleitet) wie bereits bei den

[3] In einer Implementierung kann dies effizient durch sich enthaltende Symboltabellen gelöst werden. Hierdurch werden Duplikate vermieden.

```
EBNF
1  Automaton ::= Name State* Transition*
2
3  State ::= Name EntryAction?
4            State* ExitAction?
5
6  Transition ::= Name "->" Name
7
8  EntryAction ::= ...
9
10 ExitAction ::= ...
```

```
EBNF
1  Automaton ::= State*
2
3  State ::= State*
4
5  //andere Regeln haben
6  //keinen Einfluss auf
7  //die Namespace-Struktur
8
9
10
```

Abbildung 7.6.: Ursprungsgrammatik und zugehörige Namespace-Grammatik

assoziierten Namespaces als Attributgrammatik darstellen. Die Attribute sind dabei die Symboltabellen, die in der Namespace-Hierarchie weitergegeben werden. Die Regeln zur Weitergabe werden vom Sprachentwickler aufgrund sprachspezifischer Eigenschaften festgelegt, dennoch können einige grundlegende Eigenschaften der Regeln festgestellt werden.

Die Grundidee ist dabei die Verwendung der zuweisenden und nutzenden Regeln: importierte Symboltabellen für den assoziierten Namespace eines Nichtterminalsymboles werden diesem per Attributierungsregel zugewiesen, es existiert demnach eine zuweisende Regel. Eine nutzende Regel existiert jedoch nicht, da es sich in diesem Falle um eine weitergeleitete Symboltabelle handeln würde. Insgesamt ergeben sich folgende Eigenschaften für den assoziierten Namespace $assNsp$ eines Namespace-aufspannenden Nichtterminals nt[4]:

1. Die gekapselte Symboltabelle des Namespaces wird in keiner Regel verwendet:

$$\mathbb{Z}_{nt,\pi_1(assNsp)} = \emptyset \wedge \mathbb{N}_{nt,\pi_1(assNsp)} = \emptyset$$

2. Eine importierte Symboltabelle wird dem Namespace zugewiesen, darf jedoch nicht von anderen Regeln genutzt werden:

$$\forall st \in \pi_2(assNsp) : \mathbb{Z}_{nt,st} \neq \emptyset \wedge \mathbb{N}_{nt,st} = \emptyset$$

3. Eine exportierte Symboltabelle wird nie zugewiesen, jedoch von anderen Regeln genutzt:

$$\forall st \in \pi_3(assNsp) : \mathbb{Z}_{nt,st} = \emptyset \wedge \mathbb{N}_{nt,st} \neq \emptyset$$

4. Weitergeleitete Symboltabellen werden sowohl importiert als auch exportiert:

$$\forall st \in \pi_4(assNsp) : \mathbb{Z}_{nt,st} \neq \emptyset \wedge \mathbb{N}_{nt,st} \neq \emptyset$$

[4] π_i bezeichnet hierbei die i-te Projektion auf das Namespace-Tupel.

7.2.5. Auflösung und Hiding

Nachdem die Namespaces aufgebaut und die Symboltabellen mit Einträgen gefüllt sind, kann in weiteren Schritten (wie zum Beispiel bei der Überprüfung von Kontextbedingungen) nach Symbolen gesucht werden. Dieser Vorgang wird als *Auflösung* bezeichnet. Eine Anfrage an einen Namespace enthält dabei mindestens zwei Informationen: Erstens, den Namen und zweitens die Sorte des gesuchten Elementes. Die Sorte wird dabei benötigt, da es in den meisten Sprachen möglich ist, verschiedene Elemente unterschiedlicher Sorten zu definieren. So ist es in Java beispielsweise möglich, sowohl ein Attribut `foo` als auch eine gleichnamige Methode zu definieren.

Neben dem Namen und der Sorte des gesuchten Symbols können jedoch auch weitere Eigenschaften das Ergebnis einer Auflösung beeinflussen. Bekanntes Beispiel ist das Überladen von Methoden, hier spielen die Typen der Parameter eine wesentliche Rolle. Daneben können auch spezielle Konstrukte von Sprachen die Auflösung beeinflussen. So behandelt die Methodenauflösung in Java auch generische Methoden und Varargs [GJS05].

Insgesamt lässt sich die Auflösung demnach wie folgt darstellen: zunächst existiert ein sprachübergreifender bzw. generischer Anteil, der anhand des Namens und der gesuchten Sorte entsprechende Einträge findet. Anschließend existiert für jede Sprache eine spezifische Funktion, die in der Lage ist, Sprachspezifika mit einzubeziehen und damit die Ergebnisse des generischen Anteils zu filtern. Zusammenfassend ergeben sich folgende Definitionen:

Definition 7.2.10 (Auflösungsfunktionen resolve und resolveSpecific)**.** *Die sprachübergreifende Auflösungsfunktion*

$$resolve : \mathbb{N}\mathrm{sp} \times \mathbb{N}\mathrm{ame} \times \mathbb{S}\mathrm{ort} \to \wp(\mathbb{E}\mathrm{ntry})$$

liefert ausgehend von einem Namespace die Menge der Symboltabelleneinträge mit gegebenem Namen und gegebener Sorte.
Zusätzlich existieren für jede Sprache spezifische Funktionen $resolveSpecific$, die in der Lage sind, das Ergebnis von $resolve$ gemäß sprachspezifischer Eigenschaften weiter zu filtern. Für jede Sorte existiert hierbei je eine Funktion, die sortenspezifische Funktionsargumente besitzt.

Neben dem Überladen von Elementen ist das *Überlagern* ein in vielen Sprachen angewandtes Konzept. Hierbei werden in unterschiedlichen Namespaces Elemente gleichen Namens und gleicher Sorte definiert, wobei diese sich dann gegenseitig verschatten. Dies geschieht meist entlang des Namespace-Baumes: in untergeordneten Namespaces werden dabei Elemente eines (transitiv) übergeordneten Namespaces verdeckt. Die verdeckten Elemente sind in diesen Fällen in den importierten Symboltabellen registriert, die überdeckenden Elemente hingegen in der gekapselten bzw. in den exportierten Symboltabellen. Es existieren jedoch auch Fälle, in denen sowohl die verdeckten als auch die überdeckenden Einträge in importierten Symboltabellen stehen und entsprechende Vorrangregeln die Sichtbarkeiten definieren.

Die Überlagerungsregeln basieren zunächst wie der Auflösungsmechanismus auf Name und

Sorte des Elementes, d.h. nur gleichnamige Elemente gleicher Sorte können einander überdecken. Zusätzlich können jedoch auch hier sprachspezifische Eigenschaften eine Rolle spielen, Methoden in Java überlagern sich nur, falls die Anzahl und die Typen der Parameter gleich sind. Aus diesen Gründen wird an dieser Stelle nur die Existenz einer Funktion $hides$ verlangt, die entscheidet, ob ein Symboltabelleneintrag einen anderen überdeckt.

Definition 7.2.11 (Funktion hides)**.** *Die Funktion* $hides : \mathbb{E}\text{ntry} \times \mathbb{E}\text{ntry} \rightarrow BOOL$ *besagt, ob das erste Argument vom zweiten Argument überdeckt wird und liefert in diesem Falle* `true`*, im anderen Falle* `false`*.*

Aus der Definition wird ersichtlich, dass die Funktion nicht die Namen und die Sorten der Elemente vergleicht, da diese nicht Teil der Entries, sondern Teil der Symboltabelle bzw. der Grammatik mit Typisierungsstrukturen sind. Die Anwendung der Funktion ergibt jedoch nur für namens- und sortengleiche Elemente Sinn und wird daher als Teil von $resolve$ verwendet. Hier kann daraus Nutzen gezogen werden, dass $resolve$ eine Menge namens- und sortengleicher Symboltabelleneinträge liefert und auf ein erneutes Vergleichen verzichtet werden kann. Demnach liefert die $resolve$-Funktion nur die Menge der Einträge zurück, die von keinem anderen Eintrag überdeckt sind. Demnach gilt für das Ergebnis r eines $resolve$-Aufrufs:

$$\forall r_1, r_2 \in r : hides(r_1, r_2) = true \Rightarrow r_1 = r_2$$

7.2.6. Sprachkombinationen

Die bisher erfolgten Betrachtungen behandeln die Symboltabellenkonzepte unabhängig vom übergeordneten Ziel der Kompositionalität. Zum Erreichen einer sprachübergreifenden Infrastruktur sind daher zusätzliche Komponenten nötig. Diese Komponenten verbinden Sprachen an zwei wesentlichen Stellen: zum einen werden Entries unterschiedlicher Sprachen aufeinander übertragen, zum anderen interagieren die Auflösungsmechanismen miteinander. Verdeutlicht werden können diese Mechanismen anhand des Beispiels der Kombination von Java und Statecharts aus Teilkapitel 4.1.3.

Beispiel 7.2.5. *Die im Teilkapitel 4.1.3 vorgestellte Verbindung zwischen Statecharts und Java verdeutlicht zwei Effekte: Erstens werden Symboltabelleneinträge der Zustände in Statecharts als Symboltabelleneinträge für Variablen in Java interpretiert. Zweitens existiert ein Auflösungsmechanismus, der aus Sicht von Java Variablen auflöst, intern jedoch nach verschiedennamigen Zuständen im Statechart sucht.*

Der Mechanismus zur Übersetzung der Symboltabelleneinträge kann aus theoretischer Sicht wie in Definition 7.2.12 formuliert werden.

Definition 7.2.12 (Funktion translateEntry)**.** *Die Funktion* $translateEntry : \mathbb{E}\text{ntry} \rightarrow \mathbb{E}\text{ntry}$ *liefert ausgehend von einem gegebenen Symboltabelleneintrag einen neuen Entry. Die konkrete Übersetzung ist dabei abhängig von der gegebenen Sprachkombination.*

Bei der Kombination von Sprachen müssen demnach im Falle von zu übersetzenden Symboltabelleneinträgen ein oder mehrere Funktionen definiert werden, die eine intendierte Übersetzung realisieren. Da jedoch auch der Auflösungsmechanismus angepasst werden muss, werden zusätzliche $resolve$-Funktionen benötigt. Diese Funktionen sind dabei nicht von der aus Definition 7.2.10 bekannten Struktur verschieden, lediglich die interne Arbeitsweise unterscheidet sich: während die sprachspezifischen Mechanismen typischerweise die gegebenen Namespaces und Symboltabellen durchsuchen, verwenden sprachübergreifenden Auflösungsmechanismen die sprachspezifischen Funktionen wieder und übersetzen deren Ergebnisse in Symboltabelleneinträge der Zielsprache.

Beispiel 7.2.6. *Im Beispiel der Kombination von Statecharts und Java existiert nach der Sprachkombination eine zusätzliche Auflösungsfunktion für Variablen in Java. Diese verwendet die Auflösungsfunktion der Statecharts für Zustände und übersetzt deren Ergebnisse in Symboltabelleneinträge für Java-Variablen.*

Anhand der beschriebenen Funktionen können anschließend sprachübergreifende Auflösungen und Übersetzungen der Symboltabelleneinträge erfolgen. Dabei werden die ursprünglichen Funktionen wiederverwendet und somit der Aufwand bei Sprachkombination minimiert.

7.3. Struktur und Arbeitsweise

Die generelle Struktur und Arbeitsweise der Symboltabellen richtet sich stark nach den Erkenntnissen der vorangegangenen theoretischen Abhandlungen. Aufbauend darauf sind jedoch Konzepte und Methodiken enthalten, die zusätzliche Aufgaben übernehmen. Im Folgenden werden die einzelnen Teilaspekte eingeführt und deren Funktionsweisen sowie das Zusammenspiel der Komponenten erläutert.

7.3.1. Zeit- und Speichermanagement

Ein wichtiger Aspekt bei der Erstellung eines Ansatzes für Symboltabellen ist das Zeit- und Speichermanagement. Hintergrund hierfür ist, dass Systeme typischerweise aus einer großen Zahl komplexer Modelle erstellt werden. Hierdurch entstehen verschiedene Implikationen für die Infrastruktur des Sprachwerkzeugs:

1. Aufgrund des begrenzten Speichers eines Rechners ist es nicht sinnvoll, alle Modelle zur gleichen Zeit im Speicher zu halten. Daher müssen die Modelle modular verarbeitet werden.

2. Für die Korrektheitsprüfung eines Modells müssen typischerweise Informationen anderer Modelle benutzt werden, insbesondere sind dies alle referenzierten Modelle. Hierbei muss jedoch nicht das komplette referenzierte Modell betrachtet werden, sondern nur ein Teil davon: die nach außen sichtbare Schnittstelle bzw. das *Interface des Modells*.

3. Im Sinne der Agilität sollten prinzipiell nur Modelle verarbeitet werden, die sich seit der letzten Verarbeitung verändert haben. Zwar hat diese Eigenschaft schon beim Einlesen der Modelle Auswirkungen - d.h. nur diese Modelle werden geparst und ein AST aufgebaut - besonders in Verbindung mit dem vorangegangen Punkt ergeben sich jedoch zusätzliche Implikationen: selbst wenn ein Modell unverändert bleibt, muss die Schnittstelle möglicherweise erneut eingelesen werden. Dies ist genau dann der Fall, wenn dieses Modell von einem geänderten Modell referenziert wird.

Die generelle Idee zur Behandlung dieser Aspekte besteht in der expliziten Erstellung und Speicherung von Modellinterfaces. Die Idee ähnelt dabei der Vorgehensweise von Java-class Dateien [GJS05]: diese beinhalten im Header die nach außen sichtbare Information der entsprechenden Klassen, d.h. deren Signatur, Methoden und Variablen unter Beachtung der Modifikatoren. Ähnlich hierzu werden im gewählten Ansatz die Symboltabellen in separaten Dateien abgespeichert, wobei verschiedene Symboltabellen je Modell erstellt werden können. Dies wird insbesondere beim Vorhandensein unterschiedlicher Sichtbarkeiten in der Sprache genutzt: je Sichtbarkeit wird genau eine Symboltabellendatei erstellt und bei Bedarf geladen.

Durch die explizite Erstellung der Modellinterfaces wird bereits ein wesentlicher Aspekt des Zeit- und Speichermanagements behandelt: für referenzierte Modelle müssen jeweils nur die Schnittstellendateien geladen werden, auf das Einlesen des kompletten Modells wird verzichtet. Um jedoch die Korrektheit der Schnittstellen sicherzustellen, ist eine geeignete Ablaufsteuerung zu wählen: für alle geänderten Modelle müssen zunächst die Schnittstellen neu erstellt werden, bevor mit der eigentlichen Behandlung der Modelle begonnen wird. Diese Vorgehensweise kann sehr leicht mit geeigneten Werkzeugen wie *make* oder *ant* realisiert werden. Zusätzlich wird hierdurch sichergestellt, dass lediglich geänderte Modelle eingelesen werden.

Insgesamt bietet die Vorgehensweise der expliziten Modellschnittstellen Unterstützung für die aufgeführten Aspekte des Zeit- und Speichermanagements. Zur Erläuterung der detaillierten technischen Umsetzung müssen jedoch erst einige grundlegende Aspekte der Symboltabellen-Infrastruktur sowie der Arbeitsweise erläutert werden.

7.3.2. Strukturen

Entries

Zentrales Element der Symboltabellen-Struktur sind die Einträge bzw. Entries. Diese haben prinzipiell eine sprachabhängige Struktur, einige Eigenschaften sind jedoch von genereller Natur: so haben Entries prinzipiell Namen, über die die Einträge referenziert werden können. Neben dem Namen haben Einträge zusätzlich einen Zustand, dessen Funktion anhand Abbildung 7.7 erläutert werden kann.

Die abgebildete Java-Klasse für Personen hat dabei zwei Attribute: einen Namen und ein Partner. Bei der Erstellung von Symboltabelleneinträgen für diese Attribute kommt es hierbei zu

Java-Quellcode

```
1 class Person{
2
3    String name;
4    Person partner;
5
6 }
```

Abbildung 7.7.: Auszug aus einer Java-Klasse.

verschiedenen Effekten:

- Die Symboltabelleneinträge für die Attribute referenzieren Symboltabelleneinträge für den jeweiligen Typ (`String` bzw. `Person`) der Attribute.
- Beim Auffinden der Attribute ist für die Typen zunächst nur der unqualifizierte Name bekannt. Zusätzlich kann dieser Name noch nicht qualifiziert werden: für den Typ `String` könnte etwa in der gleichen Klasse an späterer Stelle eine innere Klasse `String` definiert worden sein oder es wird auf die Standardklasse `java.lang.String` verwiesen. Daher kann erst am Ende der Klasse entschieden werden, wie der vollqualifizierte Typname lautet.
- Ein vollqualifizierter Typname ist in jedem Fall eindeutig. Für den Symboltabelleneintrag des Namens einer Person ist es deswegen ausreichend, den vollqualifizierten Typnamen (`java.lang.String`) zu kennen bzw. eine Referenz auf diesen Typnamen zu besitzen. Die Eigenschaften des Typs sind jedoch zunächst irrelevant, insbesondere braucht die Schnittstelle von `String` nicht betrachtet bzw. geladen zu werden.
- Für den Symboltabelleneintrag des Partners einer Person ist der vollqualifizierte Typname auch ausreichend. Da jedoch dieser Typ vollständig bekannt ist (da im gleichen Modell definiert), kann gleich der eigentliche Symboltabelleneintrag für den Typ referenziert werden. Auf eine Referenzierung auf den Typnamen kann verzichtet werden.

Insgesamt ergeben sich durch diese Betrachtungen drei verschiedene *Zustände* eines Entries:

1. *Unqualifiziert:* Dies ist der initiale Zustand eines referenzierten Eintrages, wobei nur der unqualifizierte Name des Elementes bekannt ist.
2. *Qualifiziert:* Hierbei ist der vollqualifizierte Name des referenzierten Elementes bekannt, nicht jedoch seine Eigenschaften. Dieser Zustand tritt auf, falls das referenzierte Element nicht im aktuellen Modell definiert ist.
3. *Vollständig:* Neben dem vollqualifizierten Namen sind hier alle Eigenschaften des referenzierten Elementes bekannt. Entweder ist dieses Element im gleichen Modell definiert oder aber die Schnittstelle, in der das Element beschrieben ist, wurde geladen.

Zur Umsetzung der Zustände von Symboltabelleneinträgen wurde wie folgt vorgegangen: jeder Eintrag verfügt zunächst über Referenzen zur qualifizierten bzw. vollständigen Version gemäß Abbildung 7.8.

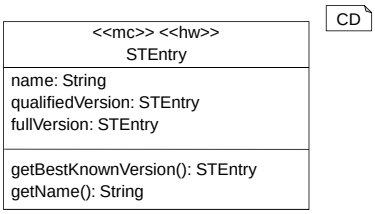


Abbildung 7.8.: Attribute zur Umsetzung der Zustände von Symboltabelleneinträgen.

Zusätzlich zu den Referenzen existiert eine Methode, welche die jeweils bestbekannte Version des Symboltabelleneintrags zurückliefert. Diese Methode wird von allen anderen Methoden wie in Abbildung 7.9 genutzt.

Java-Quellcode «hw»

```
1 public String getName() {
2   if (getBestKnownVersion() != this) {
3     return getBestKnownVersion().getName();
4   }
5   return name;
6 }
```

Abbildung 7.9.: Implementierung der Methode `getName()`.

Wie in diesem Codefragment dargestellt ist, liefert die Methode prinzipiell den Namen zurück, der in der bestbekannten Version abgelegt ist. Dadurch müssen Referenzen zu Symboltabelleneinträgen nicht verändert werden: existiert zunächst eine Referenz zu einem unqualifizierten Eintrag, liefert die Methode `getName()` den unqualifizierten Namen. Wird der Eintrag später qualifiziert, so wird die neue Version lediglich als qualifizierte Version bei der unqualifizierten Variante registriert. Referenzen auf die unqualifizierte Version können erhalten bleiben, da die Methode immer den Namen der bestbekannten Version zurückliefert. Ähnlich wird bei allen anderen Methoden eines Symboltabelleneintrages vorgegangen.

Ausgehend von der Basisversion der Entries können vom Benutzer Subklassen für jede Sprache entwickelt werden. Diese beinhalten sprach- und entitätenspezifische Attribute und nutzen dabei die von der Basisklasse zur Verfügung gestellte Funktionalität. Insbesondere verwenden sie die beschriebenen Methoden zur Umsetzung der Zustände der Symboltabelleneinträge. Zusätzlich müssen die Subklassen die abstrakte Methode `getKind()` von `STEntry` implementieren und hierin die Sorte des Symboltabelleneintrags (z.B. „Klasse“, „Methode“ oder „Zustand“) zurückliefern. Aus Effizienzgründen ist dabei die Sorte als Singleton-Konstante unter Verwendung der `intern()`-Methode von `java.lang.String` zu speichern.

Symboltabellen und Namespaces

Symboltabellen bilden die Struktur, welche die eigentlichen Einträge verwaltet. Die Verwaltung der Symboltabellen erfolgt im zugeordneten Namespace. Diese sind hierarchisch angeordnet und verfügen über eine gekapselte Symboltabelle und Mengen von importierten, exportierten und weitergeleiteten Symboltabellen. Insgesamt ergibt sich somit in der Implementierung die bereits in Abbildung 7.5 aus Abschnitt 7.1.3 eingeführte Beziehung zwischen Namespaces, Symboltabellen und Entries.

7.3.3. Symboltabellenaufbau

Prinzipiell gliedert sich der Symboltabellenaufbau in verschiedene Arbeitsschritte mit jeweils festgelegter Aufgabenstellung. Der Ablauf des Aufbaus ist dabei vom Framework festgelegt: es existieren verschiedene Workflows, die unterschiedlich parametrisiert sind. Zur konkreten Konfiguration der Workflows sind vom Sprachentwickler verschiedene Komponenten zu entwickeln und in den Sprachbestandteilen aus Abbildung 4.8 zu registrieren. Im Folgenden werden die in Abbildung 7.10 aufgelisteten Arbeitsschritte, die beteiligten Workflows sowie Parameter der Sprachbestandteile vorgestellt.

1. Aufbau der Namespace-Struktur
2. Erstellung und Eintragung von Entries
3. Verbinden der Namespaces des Modells (optional)
4. Qualifizierung referenzierter Entries
5. Speichern der Modellschnittstelle
6. Importieren von Symboltabellen
7. Verbinden der Namespaces des Modells (optional)
8. Setzen der Auflösungskomponenten
9. Nachladen qualifizierter Entries

Abbildung 7.10.: Schritte des Symboltabellenaufbaus

1. Aufbau der Namespace-Struktur

Der erste Arbeitsschritt bei der Erstellung der Symboltabellen ist der Aufbau der Namespace-Struktur. Wie bereits in Abbildung 7.4 verdeutlicht wurde, bilden die Namespaces eine Parallelstruktur zum AST. Daher ist es ausreichend, wenn der Sprachentwickler definiert, wel-

che AST-Knoten einen neuen Namespace aufspannen. Hierzu verfügt jede Sprachkomponente über ein mengenwertiges Attribut, in der die entsprechenden Knotenklassen registriert werden können. Diese Knotenklassen entsprechen dabei der ausgezeichnete Menge von Namespace-aufspannenden Nichtterminalsymbolen einer Grammatik mit Typisierungsstrukturen gemäß Definition 7.2.5.

Zum Aufbau der Namespace-Strukturen verfügt das Framework über einen Workflow, der im ersten Schritt die Knotenklassen aller registrierten Sprachkomponenten ausliest. Im zweiten Schritt traversiert ein Visitor über den AST und baut den Namespace-Baum entsprechend der Konfiguration auf. Anschließend wird sowohl der Namespace-Baum als auch die Zuordnung von Knoten zum assoziierten Namespace gespeichert. Aus theoretischer Sicht entspricht diese Zuordnung den assoziierten Namespaces gemäß Definition 7.2.9.

2. Erstellung und Eintragung von Entries

Im zweiten Schritt des Symboltabellenaufbaus werden zunächst die Entries abhängig vom Inhalt des ASTs instanziiert. Hierfür stellt der Sprachentwickler eine Subklasse von `Concrete-ASTAndNameSpaceVisitor` zur Verfügung. Dieser ist ein spezieller Visitor, der neben der Traversierungsfunktionalität Informationen über die im ersten Schritt aufgebaute Namespace-Struktur enthält, insbesondere kann hierin zu jedem Knoten der assoziierte Namespace abgerufen werden.

Innerhalb des Visitors erfolgt die Instanziierung der Entries. Anschließend werden diese in die korrekten Symboltabellen des aktuellen Namespaces hinzugefügt. Abbildung 7.11 zeigt ein Beispiel für das Hinzufügen eines Entries für Klassen in einer Symboltabelle für Java.

In dieser Abbildung wird dabei in den ersten 11 Zeilen zunächst der Namespace und die entsprechende Symboltabelle herausgesucht, anschließend wird der Entry instanziiert und in Zeile 14 der Symboltabelle hinzugefügt. Zusätzlich kommt es zu den bereits vorgestellten Effekten der Zustände von Einträgen: Einträge des aktuellen Modells sind stets vollständig, daher wird in Zeile 11 der Zustand entsprechend gesetzt. Referenzen zu anderen Einträgen sind zunächst unqualifiziert - im Beispiel anhand des Supertyps ab Zeile 16 dargestellt. Hierfür wird ebenso ein Symboltabelleneintrag erstellt und der Zustand entsprechend gesetzt. Dieser Eintrag wird jedoch nicht der Symboltabelle hinzugefügt, da der Typ nicht an dieser Stelle definiert ist. Vielmehr wird der Eintrag als Referenz zum Supertyp des aktuellen Typs (Zeile 20) und zusätzlich in einem Speicher hinterlegt (Zeile 23). Dieser Speicher beinhaltet dabei alle unqualifizierten Entries und wird in Schritt 4 - der Qualifizierung - wiederverwendet.

Insgesamt unterliegen diesem Arbeitsschritt verschiedene theoretische Aspekte. Zum einen werden auch hier die assoziierten Namespaces gemäß Definition 7.2.9 verwendet. Zum anderen spielen sowohl die Entries aus Definition 7.2.1 als auch verschiedenen Symboltabellenarten aus Definition 7.2.3 und deren Anordnung im Namespace gemäß Definition 7.2.6 eine tragende Rolle.

3. Verbinden der Namespaces des Modells (optional)

Prinzipiell existieren zwei verschiedene Strategien zur Organisation der Entries in Namespaces: Erstens ist es möglich, jeden Entry nur in dem Namespace einzutragen, in dem er definiert wur-

Java-Quellcode «hw»

```
1 public void visit(ASTClassDeclaration d) {
2   //aktuellen Namespace heraussuchen
3   NameSpace nsp=getNameSpace(d);
4   //korrekte Symboltabelle heraussuchen
5   SymbolTable target=getExportedSymbolTable(nsp, d.getModifier());
6
7   //Entry instanziieren
8   TypeEntry te = JavaEntryFactory.createTypeEntry();
9   String typeName = d.getName();
10  te.setName(typeName);
11  te.setEntryState(STEntryState.FULL);
12
13  //Entry der Symboltabelle hinzufügen
14  target.addEntry(te);
15
16  if (d.getSuperType()!=null){
17    String superTypeName = d.getSuperType().getName();
18    TypeEntry superType = JavaEntryFactory.createTypeEntry();
19    superType.setName(superTypeName);
20    te.setSuperType(superType);
21    te.setEntryState(STEntryState.UNQUALIFIED);
22
23    storage.addUnqualifiedEntry(superType);
24  }
25 }
```

Abbildung 7.11.: Beispiel für die Instanziierung eines Symboltabelleneintrages

de. Beim Suchen ausgehend von einem untergeordneten Namespace muss anschließend eventuell in der Namespace-Hierarchie nach oben weiter durchsucht werden, um einen Eintrag zu finden. Zweitens besteht die Möglichkeit, in einer zusätzlichen Phase sichtbare Entries übergeordneter Namespaces in die importierten Symboltabellen untergeordneter Namespaces explizit einzutragen[5]. In dieser Variante sind in jedem Namespace alle sichtbaren Entries verfügbar, auf eine Durchsuchung anhand der Namespace-Hierarchie kann verzichtet werden.

Der wesentliche Unterschied zwischen beiden Versionen besteht in der Laufzeit: während die erste Variante kein explizites Eintragen von importierten Einträgen benötigt, kann der Suchprozess aufgrund der hierarchischen Suche jedoch aufwendig sein. Im Gegensatz dazu muss bei der zweiten Variante ein explizites Eintragen importierter Entries erfolgen, die Suche gestaltet sich dafür weniger aufwendig, da nur der aktuelle Namespace einbezogen werden muss.

Innerhalb des Frameworks können beide Varianten verwendet werden. Da bei der ersten Variante beim Aufbau der Symboltabellen keine speziellen Aktionen auszuführen sind, muss der Sprachentwickler hierfür keine Komponenten zur Verfügung stellen. Soll die zweite Variante angewandt werden, ist eine Klasse zu entwickeln, die das Interface `InheritedEntries-`

[5] Hierbei sollten keine Entries kopiert werden, sondern nur Referenzen auf bestehende Entries in die Symboltabelle eingetragen werden

`CalculatorClient` implementiert. Die entsprechende Instanz wird während des Symboltabellenaufbaus aufgefordert, für jeden Namespace die vom übergeordneten Namespace importierten Entries zu berechnen und anschließend in die entsprechende Symboltabelle einzutragen.

Aus theoretischer Sicht werden in diesem Arbeitsschritt vor allem die verschiedenen Symboltabellenarten gemäß Definition 7.2.3 und deren Anordnung im Namespace aus Definition 7.2.6 verwendet. Hierbei werden die Inhalte der exportierten, importierten und weitergeleiteten Symboltabellen verschiedener Namespaces anhand der sprachspezifischen Regeln synchronisiert. Weiterhin wird die Funktion $hides$ aus Definition 7.2.11 verwendet, da nur Einträge weitergegeben werden, die nicht lokal verdeckt sind.

4. Qualifizierung referenzierter Entries

Im nächsten Schritt werden die referenzierten Entries qualifiziert. Hierzu stellt ein Sprachentwickler eine Klasse innerhalb seiner Sprachkomponente zur Verfügung, die das Interface `IQualifierClient` implementiert. Dieses Interface hat im Wesentlichen zwei Methoden:

- `qualify(Entry, NameSpace)`: Hiermit wird die Komponente zunächst aufgefordert, anhand des aktuellen Namespaces den gegebenen Entry zu qualifizieren. Hierbei wird nur das aktuelle Modell betrachtet.

- `qualify(Entry, ModelNameQualifier)`: Ist der Versuch anhand des aktuellen Modells nicht erfolgreich, können anhand des `ModelNameQualifiers` andere Modelle mit in die Suche einbezogen werden. Hierfür werden `ModelNameQualifier` zur Verfügung gestellt, die unter anderem das Nachladen von Modellinterfaces erlauben. Zusätzlich existieren Methoden zur Prüfung der bloßen Existenz von Modellen. Dies wird unter anderem in Java verwendet, da es hier zum Qualifizieren eines Typs ausreicht, nach einer Klasse in einem bestimmten Paket zu suchen. Zur Qualifizierung muss die Klasse dabei nicht geladen werden, die Information über die Existenz ist ausreichend.

5. Speichern der Modellschnittstelle

Nachdem alle unqualifizierten Entries qualifiziert worden sind, kann die Schnittstelle des Modells gespeichert werden. Der Ansatz zur Speicherung der Entries ist dabei grammatikbasiert: Sprachentwickler schreiben eine einfache Beschreibungssprache für die Symboltabelleneinträge, woraus ein Parser/Lexer zum späteren Einlesen generiert wird. Zusätzlich verfügen Entries über eine Methode zum Serialisieren, worin der Entry gemäß der Grammatik in eine textuelle Form gebracht wird.

Die Auswahl, welche verschiedenen Modellschnittstellen serialisiert werden, erfolgt über die Symboltabellen: diese verfügen über eine Methode `isInterface()`, anhand derer abgefragt werden kann, ob der Inhalt der Symboltabelle eine Schnittstelle des Modells ist. Die Art der Symboltabelle (z.B. protected, public) wird zusätzlich verwendet, um mehrere Schnittstellen für ein Modell zu erstellen. Das Framework übernimmt dabei die Speicherung einer Modellschnittstelle an einem vom Modellnamen und der Symboltabellenart abhängigen Ort, sodass die Schnittstelle in weiteren Schritten leicht gefunden und geladen werden kann.

Als Beispiel einer Modellschnittstelle für eine einfache Java-Klasse dient Abbildung 7.12. Diese Schnittstelle beschreibt dabei eine Klasse Person mit einer öffentlichen Methode `getAge`. Hierfür werden die verschiedenen Entries jeweils in einem `kind`-Block unter Angabe der Sorte des Entries (Typ oder Methode) abgelegt. Innerhalb des `description`-Blocks erfolgt die Beschreibung der Eigenschaften des Entries, zusätzlich können wie bei der Methode gezeigt, Entries hierarchisch aufgebaut werden.

Modellschnittstelle «gen»

```
1 kind Type{
2   description{
3     name: example.Person;
4   }
5   kind Method{
6     description{
7       name: getAge;
8       modifiers: public;
9       returntype: java.lang.Integer;
10    }
11  }
12 }
```

Abbildung 7.12.: Beispiel für eine Modellschnittstelle einer Java-Klasse

Anzumerken ist, dass die vom Sprachentwickler zu erstellende Grammatik lediglich den Teil innerhalb des `description`-Blocks behandeln muss. Aus dieser einfachen Grammatik wird durch MontiCore ein Parser generiert, der im Framework zum Einlesen von Modellschnittstellen bzw. Teilen davon verwendet werden kann. Da jede Sprache und jeder Entry eine unterschiedliche Grammatik besitzt, muss zusätzlich definiert werden, für welche Sorte von Entry die jeweilige Grammatik genutzt werden soll. Hierfür ist in jeder Sprachkomponente ein `InterfaceParserExtender` anzugeben, der über eine wie in Abbildung 7.13 dargestellte Methode verfügt.

Java-Quellcode «hw»

Glue

```
1 public void extend(MonticoreParser p) {
2  p.addMCConcreteParser(
3    new JavaInterfaceTypeDescriptionMCConcreteParser("typeParser"));
4
5  p.addExtension(INTERFACE_PARSER_ID, "typeParser", "Description",
6    TypeEntry.KIND );
7 }
```

Abbildung 7.13.: Erweiterung des Schnittstellenparsers

In dieser Abbildung wird dem vom Framework vorgegebenen Parser für Modellschnittstellen der generierte Parser für die Beschreibung von Typen hinzugefügt (Zeile 2 und 3), gleichzeitig erhält dieser Parser dabei einen Namen („typeParser“). In Zeile 5 wird zusätzlich definiert, dass

vom Parser für Modellschnittstellen (identifiziert durch die Konstante `INTERFACE_PARSER_ID`) auf den Typparser (`typeParser`) übergegangen wird, falls der Parameter `Description` den Wert der Konstante `TypeEntry.KIND` hat. Der Wert dieses Parameters wird automatisch vom Framework auf die Sorte des aktuellen Entries gesetzt, hierfür dient der auf `kind` folgende Identifikator aus Abbildung 7.12. Daher muss im gegebenen Beispiel `TypeEntry.KIND` den Wert „Type“ haben, damit der Schnittstellenparser korrekt auf den Typparser wechselt. Durch diese Vorgehensweise können verschiedene Parser für die Entries von möglicherweise mehreren Sprachen registriert werden.

6. Importieren von Symboltabellen

Bereits nach Schritt 4 - der Qualifizierung referenzierter Entries - steht die Schnittstelle des Modells fest. Die gesamten Eigenschaften eines Modells hängen jedoch nicht nur von dem im Modell definierten Entries ab, vielmehr ist es möglich, zusätzliche Elemente anderer Modelle einzubinden. Das bekannteste Beispiel hierfür ist die objektorientierte Vererbung: hier sind die Methoden und Variablen der Superklasse in der Subklasse sichtbar, daher beinhaltet die importierte Symboltabelle der Subklasse die exportierte Symboltabelle der Superklasse. Von großer Wichtigkeit ist dabei der Unterschied zum Importieren von Namen (z.B. per `import`-Statement in Java): Beim Namensimport ist es lediglich möglich, die Namen der importierten Typen an späterer Stelle auch unqualifiziert zu verwenden, der Namensimport wird beim Qualifizieren referenzierter Entries behandelt. Beim Symboltabellen-Import wird zusätzlich die gesamte Symboltabelle inklusive aller dort definierten Einträge importiert, hierfür ist ein zusätzlicher Schritt beim Symboltabellenaufbau nötig.

Zur Umsetzung wird bereits in Schritt 2, d.h. beim Traversieren des AST, in einem `Storage` vorgemerkt, welche Art von Symboltabelle welches Modells importiert werden soll. Hierzu dient die Klasse `ImportedSTData`, welche in der Lage ist, die für das Nachladen benötigten Informationen zu speichern. Diese Informationen beinhalten unter anderem die zu ladende Symboltabelle und einen Symboltabelleneintrag für das zu ladende Element. Der Grund, einen Symboltabelleneintrag anstatt des Namens für das zu ladende Modell zu verwenden, liegt in der Qualifizierung: im Beispiel von Java kann die Superklasse unqualifiziert angegeben werden, durch die Verwendung eines Entries wird dieser automatisch im Schritt 4 - der Qualifizierung der Entries - durch einen qualifizierten Entry ersetzt. Hierdurch entsteht eine einheitliche Vorgehensweise und Sprachentwickler müssen die importierten Symboltabellen nicht in einem zusätzlichen Schritt behandeln. Als Endergebnis verfügt ein `ImportedSTData` über einen qualifizierten Entry, der den vollqualifizierten Namen beinhaltet.

Da das `Storage` über alle nötigen Informationen verfügt, die korrekte Symboltabelle nachzuladen, wird vom Framework automatisch ein Workflow eingebunden, der die Interfaces der Modelle einliest und der Ziel-Symboltabelle hinzufügt. Hierbei treten die bereits genannten Vorteile expliziter Schnittstellen in Erscheinung: es werden nicht die eigentlichen Modelle eingelesen, sondern nur die Interfaces. Hierdurch wird sowohl die benötigte Zeit, als auch der benötigte Speicherbedarf reduziert.

Diesem Arbeitsschritt unterliegen aus theoretischer Sicht wiederum die verschiedenen Symboltabellenarten aus Definition 7.2.3 und deren Anordnung im Namespace gemäß Definition 7.2.6. Sprachspezifische Regeln bestimmen dabei das Verhalten der Weitergabe von Entries

in exportierten, importierten und weitergeleiteten Symboltabellen verschiedener Namespaces. Auch hier wird die Funktion $hides$ aus Definition 7.2.11 verwendet, da nur Einträge weitergegeben werden, nicht lokal verdeckt sind.

7. Verbinden der Namespaces des Modells (optional)

Wie bereits in Schritt 3 müssen die durch das Importieren von Symboltabellen eingefügten Entries in untergeordneten Namespaces verbreitet werden. Auch hierbei ist dieser Schritt optional: falls die spätere Suche durch Traversierung der Namespaces erfolgen soll, kann dieser Schritt entfallen. Soll die Suche nur im jeweils aktuellen Namespace durchgeführt werden, muss eine explizite Verbreitung der Entries durch die gleichen Mechanismen wie in Schritt 3 erfolgen.

Da dieser Arbeitsschritt den gleichen Aspekten wie Schritt 3 unterliegt, entsprechen sich auch die theoretischen Grundlagen beider Arbeitsschritte.

8. Setzen der Auflösungskomponenten

Am Ende des vorangegangenen Schrittes ist der Symboltabellenaufbau abgeschlossen: alle Eigenschaften des Modells sind in der Symboltabellen-Struktur eingetragen und können entsprechend abgefragt werden. Zur Auflösung von Elementen ist jedoch eine weitere Komponente nötig: der `Resolver`. Zur Erläuterung der Funktionsweise muss zunächst die Struktur einer Anfrage an eine Symboltabelle analysiert werden. Eine solche Anfrage besteht grundsätzlich aus folgenden Informationen:

- Sorte: Die erste Information besteht in der Sorte des gesuchten Entries. Diese muss dem Rückgabewert der `getKind()`-Methode der gesuchten Entry-Sorte entsprechen. Die Sorte muss angegeben werden, da es in den meisten Sprachen möglich ist, verschiedenen Elemente unter gleichem Namen zu verwenden: in Java können beispielsweise Methoden und Variablen den gleichen Namen besitzen.
- Name: Der Name des gesuchten Elementes.
- Namespace: Der Namespace, in dem der Name verwendet wurde. Dieser wird als Ausgangsort für die Suche genutzt.
- Zusatzinformationen: In einigen Sprachen sind Zusatzinformationen bei der Suche nötig, die Suche nach Methoden in Java bezieht beispielsweise die Typen der Parameter mit ein.

Zur Umsetzung von Anfragen arbeitet der Auflösungsmechanismus ähnlich wie andere Sprachkomponenten mit Delegation. Hierfür verfügt ein `Resolver` über eine Menge von `IResolverClients`, die die Auflösungslogik implementieren und konkrete Anfragen bearbeiten können. Jeder `IResolverClient` ist dabei für genau eine Sorte von Symboltabelleneinträgen verantwortlich, eine Anfrage an den `Resolver` wird je nach Sorte weiterdelegiert und die Anfrage aufgelöst.

Die theoretische Grundlage dieses Arbeitsschritts bilden vor allem die Funktionen $resolve$ und $resolveSpecific$ aus Definition 7.2.10, wobei die `IResolverClients` gleichzeitig beide Funktionen implementieren.

9. Nachladen qualifizierter Entries

Wie bereits in den vorangegangenen Schritten diskutiert wurde, sind Entries anderer Modelle nach dem Symboltabellenaufbau im qualifizierten Zustand. Hierbei ist lediglich der vollqualifizierte Name des entsprechenden Elementes bekannt, nicht jedoch weitere Eigenschaften. Um diese Eigenschaften abfragen zu können, muss der Entry in den vollständigen Zustand versetzt werden, wobei intern das Nachladen der Schnittstelle des entsprechenden Modells nötig ist. Aus Effizienzgründen erfolgt dieses Nachladen nur bei Bedarf, typischerweise entsteht dieser bei der Überprüfung von Kontextbedingungen.

Zur Umsetzung verfügt jeder Entry über eine Methode `toFullVersion()`, welche anhand verschiedener Parameter in der Lage ist, Modellschnittstellen zu laden. Eine geladene Modellschnittstelle wird dann nach dem entsprechenden Entry durchsucht und anschließend die korrekte vollständige Version des Entries gesetzt. Von außen ist demnach nur die Methode `toFullVersion()` aufzurufen, die komplette Logik ist innerhalb des Entries gekapselt.

7.4. Sprachkombinationen

Die im vorigen Teilkapitel erläuterte Funktionsweise des Symboltabellenaufbaus zeigte die prinzipiellen Arbeitsschritte ohne Bezug auf Kompositionalität auf. Um sprachübergreifende Symboltabellen zu erstellen, sind einige zusätzliche Komponenten an verschiedenen Stellen einzubinden. Diese haben die Hauptaufgabe, Entries einer Sprache in Entries einer anderen Sprache zu übersetzen. Hierfür werden zwei wesentliche Mechanismen verwendet: das Adaptieren der Symboltabelleneinträge und das Einbringen zusätzlicher Auflösungskomponenten. Im Folgenden werden beide Mechanismen anhand des aus Abbildung 4.3 bekannten Beispiels der Verbindung von Java und Statecharts erläutert. Hierbei werden Zustände von Statecharts in boolesche Java-Variablen mit leicht verändertem Namen übersetzt.

7.4.1. Adaption der Symboltabelleneinträge

Der erste Schritt bei der Sprachkombination ist das Erstellen von Adaptoren für Symboltabelleneinträge. Abbildung 7.14 zeigt eine solche Adaption für das Statechart/Java-Beispiel.

In dieser Abbildung wurde ein Entry für einen Zustand in einen Entry für Variablen übersetzt, wobei ersteres als *Ursprungseintrag* und zweiteres als *Zieleintrag* bezeichnet wird. Hierbei wurde folgende allgemeine Vorgehensweise verwendet:

- Erstellung eines Adapters als Subklasse des Zieleintrags.
- Erstellung einer Assoziation zum Ursprungseintrag.
- Überschreiben/Implementierung der Methoden des Zieleintrags unter Verwendung des Ursprungseintrags bzw. unter Verwendung allgemein verfügbarer Komponenten (z.B. `Resolver`).

Die theoretische Grundlage der Entwicklung von Adaptoren bildet die in Definition 7.2.12 eingeführte Funktion $translateEntry$ zur Übersetzung von Entries.

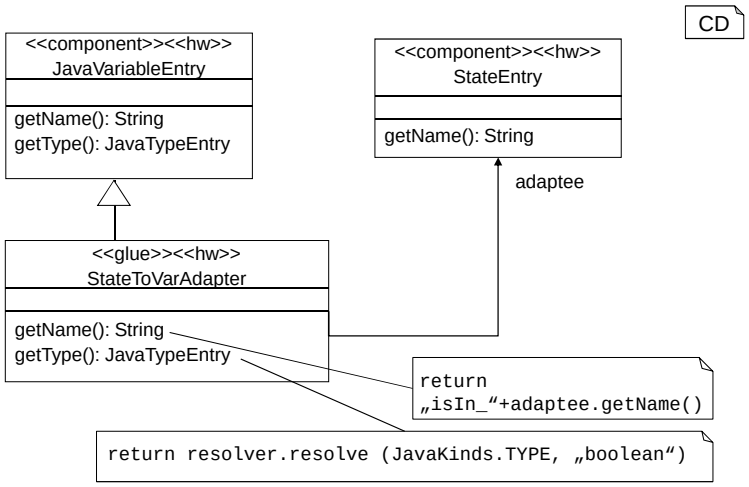


Abbildung 7.14.: Adaption von Symboltabelleneinträgen

7.4.2. Einbringen der Adaptoren

Entries einer Symboltabelle lassen sich in zwei verschiedene Kategorien einteilen: definierende und referenzierende Einträge. Definierende Einträge werden immer in die Symboltabelle eingetragen, da die entsprechende Entität der Sprache an eben dieser Stelle definiert wird. Referenzierende Einträge treten hingegen auf, wenn eine Entität der Sprache lediglich verwendet und somit nicht an dieser Stelle definiert wird. So entsteht beim Java-Ausdruck

```
String x;
```

ein definierender Entry für die Variable `x` mit einer Assoziation zu einem referenzierenden Entry für den Typ `String`.

Unmittelbar ersichtlich ist, dass zu adaptierende Symboltabelleneinträge prinzipiell in Form referenzierender Entries auftreten, da nur diese nicht an der aktuellen Stelle sondern andernorts - und möglicherweise in einer anderen Sprache - definiert worden sind. Daher kommen nur die referenzierenden Entries für eine Adaption in Frage.

Wie bereits im vorangegangenen Teilkapitel erläutert wurde, haben Referenzen auf andernorts definierte Entitäten zunächst entweder einen unqualifizierten oder einen qualifizierten Zustand. Um diese durch Adaptoren zu ersetzen, werden sie zunächst im 2. Schritt des Symboltabellenaufbaus - der Erstellung und Eintragung von Entries - in einem Speicher vorgemerkt. Bei der Kombination von Sprachen sind anschließend zwei verschiedene Komponenten einzubringen:

1. `IQualifierClient`. Dieser hat die Aufgabe, einen unqualifizierten Entry zu qualifizieren. Zwar sind diese Komponenten bereits ohne Sprachkombination vorhanden, zusätzlich sind jedoch weitere `IQualifierClients` einzubringen, die erstens eine Qualifizierung über Sprachgrenzen hinweg realisieren und zweitens als qualifizierte Version einen Adapter setzen.

2. `IQualifiedEntryHandler`. Diese Komponenten haben die Aufgabe, jeden qualifizierten Entry zu betrachten und eventuell durch Adaptoren zu ersetzen.

Anzumerken ist, dass die aufgeführten Komponenten automatisch vom Framework an der korrekten Stelle aufgerufen werden, am Ende der Bearbeitungsphase sind für alle zu adaptierenden Entries vollqualifizierte Adaptoren eingesetzt. Dies hat den Vorteil, dass beim Übergang auf die vollständige Version des Entries die Funktionalität des Adapters verwendet wird. Hierdurch ist es beispielsweise für den Entwickler der Kontextbedingungsprüfung unerheblich, ob ein Eintrag adaptiert ist oder nicht.

Insgesamt können die aufgeführten Komponenten ähnlich wie andere Teilaspekte in jeder Stufe der Composite-Struktur von Sprachen eingebracht werden. Hierdurch ist es möglich, Sprachen feingranular zu kombinieren und existierende Sprachkombinationen mit einer weiteren Sprache zu komponieren.

7.4.3. Einbringen zusätzlicher Auflösungskomponenten

Zusätzlich zu den Adaptoren müssen in den meisten Fällen von Sprachkombinationen weitere Auflösungskomponenten eingebracht werden. Diese Komponenten sind dabei die gleichen wie in Schritt 8 des Symboltabellenaufbaus, die interne Arbeitsweise unterscheidet sich jedoch. Abbildung 7.15 zeigt das Vorgehen am Beispiel der Übersetzung von Zuständen in Java-Variablen.

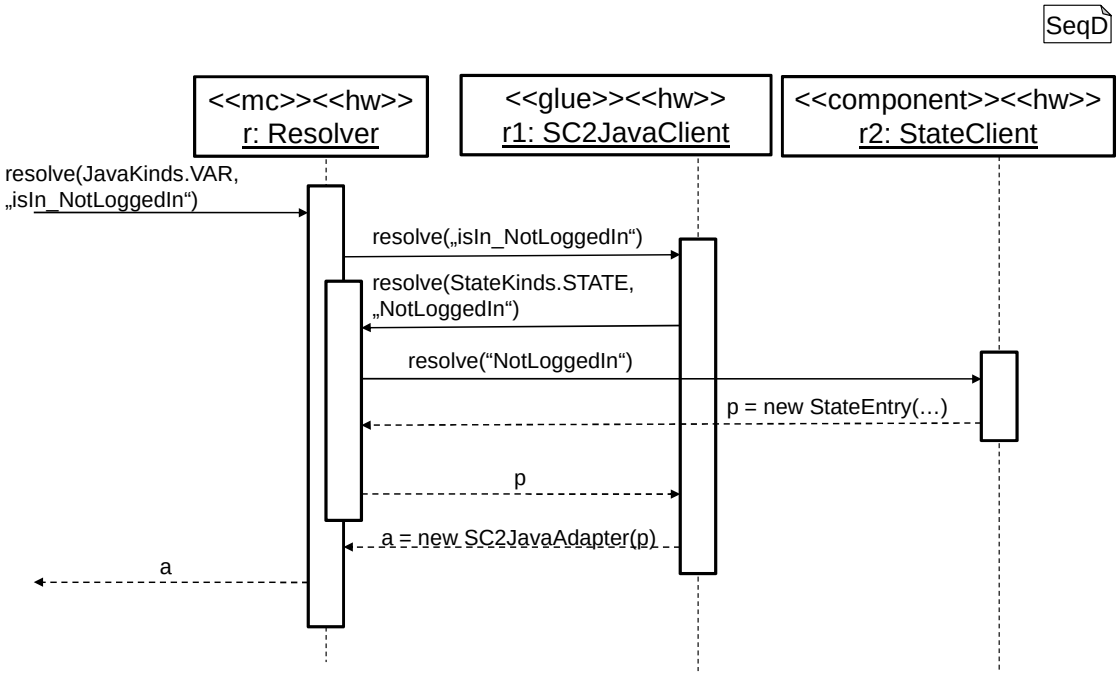


Abbildung 7.15.: Adaption von Symboltabelleneinträgen

Das Beispiel zeigt dabei die Auflösung einer booleschen Variable `isIn_NotLoggedIn` auf, wobei diese implizit durch den Zustand `NotLoggedIn` definiert wurde. Der Ablauf ergibt

sich dabei wie folgt:

- Eine Kontextbedingungsüberprüfung gibt dem Auflösungsmechanismus (d.h. dem `Resolver`) die Aufgabe, eine Java-Variable `isIn_NotLoggedIn` aufzulösen.
- Der bei Sprachkombination neu eingebrachte `SC2JavaClient` gibt an, Java-Variablen auflösen zu können. Die Ausführung besteht jedoch nicht in der eigentlichen Suche, sondern vielmehr in der Umsetzung des Namens durch Weglassen des Präfixes `isIn_` und dem Auftrag an den `Resolver`, einen Zustand aufzulösen.
- Der `Resolver` löst den Zustand mit Hilfe des `StateClients` auf. Dieser ist bereits durch den Entwickler der Sprache Statecharts gegeben. Das Ergebnis der Auflösung wird an den `SC2JavaClient` weitergegeben.
- Der `SC2JavaClient` adaptiert das Ergebnis mit Hilfe der neu entwickelten Adaptoren. Der adaptierte Eintrag wird über den `Resolver` an die Kontextbedingungsüberprüfung zurück gegeben.

Die theoretische Grundlage der Einbringung zusätzlicher Auflösungskomponenten bilden vor allem die Funktionen $resolve$ und $resolveSpecific$ aus Definition 7.2.10, auch hier implementieren die `IResolverClients` wie bei der nicht-sprachübergreifenden Version gleichzeitig beide Funktionen. Zusätzlich kommt aufgrund der Adaption der Entries die Funktion $translateEntry$ aus Definition 7.2.12 zum Tragen.

Insgesamt haben die zusätzlichen Auflösungskomponenten demnach typischerweise nur den Auftrag, Anfragen auf bestehende Komponenten umzuleiten und deren Ergebnis zu adaptieren. Daher ist der Aufwand bei der Entwicklung minimal.

Die bisher erfolgten Diskussionen erläutern die grundsätzliche Funktionsweise der Symboltabellen-Infrastruktur und sind Basis zur Umsetzung der konkreten Szenarien zur Sprachkomposition. Im Folgenden werden diese Szenarien wie bereits in den vorangegangenen Kapiteln einzeln besprochen und zusätzliche spezifische Anmerkungen gemacht.

7.5. Umsetzung der Szenarien

7.5.1. Sprachvereinigung

Bei der Sprachvereinigung werden die erforderlichen Komponenten auf der Ebene der Sprachfamilie registriert. Diese Komponenten umfassen dabei typischerweise `IQualifierClients` zur Qualifizierung von unqualifizierten Entries und `ResolverClients` zur sprachübergreifenden Auflösung. Zusätzlich sind Adaptoren für die Symboltabelleneinträge zu erstellen, welche in den beiden angesprochenen Komponenten verwendet werden. Tabelle 7.16 fasst die zu entwickelnden Komponenten zusammen.

Durch die Einbindung dieser Komponenten ist bereits eine sprachübergreifende Symboltabellen-Infrastruktur vorhanden. Dabei werden alle vorhandenen Anteile der Ursprungssprachen ohne Änderungen wiederverwendet, zusätzlich bauen die neuen Komponenten in ihrer Funktionsweise auf die vorhandenen Strukturen auf. Hierdurch beschränkt sich der Entwicklungsaufwand auf wenige Zeilen.

Komponente	Aufgabe	Registrierungsort
Adaptoren	Übersetzen der Entries verschiedener Sprachen gemäß Abschnitt 7.4.1	Nur als Rückgabetyp für Auflösungskomponenten
Auflösungskomponenten	Realisierung der sprachübergreifenden Auflösung gemäß Abschnitt 7.4.3	`LanguageFamily`
Qualifikatoren	Qualifizierung unqualifizierter Entries gemäß Schritt 4 des Symboltabellenaufbaus	`LanguageFamily`

Tabelle 7.16.: Zu entwickelnde Komponenten für die Sprachvereinigung

7.5.2. Nichtterminal-Erweiterung

Bei der Nichtterminal-Erweiterung werden zunächst die Sprachen und somit die registrierten Komponenten zum Symboltabellenaufbau durch die Bildung einer `ModelingLanguage` vereint. Die Funktionalität ergibt sich dabei automatisch aus der Gesamtfunktionalität der beteiligten Sprachen. Hauptgrund hierfür ist, dass das Symboltabellen-Framework intern kompositionale Visitoren verwendet. Hierdurch ist es möglich, die einzelnen Komponenten ohne Veränderungen oder Rekompilierung einzubinden.

Als neu zu entwickelnde Anteile kommen auch bei der Nichtterminal-Erweiterung Qualifikatoren, Auflösungskomponenten und Adaptoren für Symboltabelleneinträge gemäß Tabelle 7.16 in Frage. Die Qualifikatoren und die Auflösungskomponenten werden im Gegensatz zur Sprachvereinigung jedoch auf der Ebene der `ModelingLanguage` registriert. Die Funktionalitäten unterscheiden sich jedoch nicht von der bei der Sprachvereinigung diskutierten Version. Daher entsteht ebenfalls ein minimaler Entwicklungsaufwand.

7.5.3. Nichtterminal-Ersetzung

Im Gegensatz zur Nichtterminal-Erweiterung werden bei der Nichtterminal-Ersetzung die Funktionalitäten der Sprachkomponenten nicht vereint, sondern Teile der Funktionalitäten einer Sprache durch eine andere Sprache ersetzt. Hierbei kann es vorkommen, dass insbesondere Teilkomponenten für den Symboltabellenaufbau nicht für die neu entwickelte Sprache übernommen werden.

Unterstützt wird diese Eigenschaft der Nichtterminal-Ersetzung durch die Architektur der Symboltabellen-Infrastruktur. Alle benötigten Komponenten werden durch einfache Registrierung per `add`- oder `set`-Methode in der jeweiligen Ebene der Sprachhierarchie eingebunden. Die Erstellung einer Sprache, welche nur Teilfunktionalitäten enthält, kann dann durch Registrierung einer Teilmenge der originalen Funktionen erstellt werden. Alternativ kann zunächst die Originalsprache instanziiert und anschließend erneut die jeweiligen `set-` oder `remove`-Methoden aufgerufen und dadurch Teilfunktionalitäten entfernt oder ersetzt werden.

Zusätzlich zum Entfernen und Ersetzen existierender Funktionalitäten entstehen bei der Nichtterminal-Ersetzung ähnliche Anforderungen wie bei der Nichtterminal-Erweiterung: auch hier

können zusätzliche Komponenten wie Auflösungsmechanismen, Qualifikatoren oder Adaptoren auf der Ebene der `ModelingLanguage` registriert werden. Dementsprechend gilt auch für die Nichtterminal-Erweiterung die Übersicht aus Tabelle 7.16.

7.5.4. Einbettung einer einzelnen Sprache und Einbettung von Sprachalternativen

Sowohl bei der der Einbettung einer einzelnen Sprache als auch bei der Einbettung von Sprachalternativen kommen die Vorteile der Verwendung kompositionaler Visitoren innerhalb des Symboltabellen-Frameworks zum Tragen: die beteiligten Sprachen werden durch Bildung einer `ModelingLanguage` vereint, das Framework fragt automatisch die benötigten Teilkomponenten ab und fügt sie zu den entsprechenden Visitoren hinzu. Hierdurch entfällt eine Neuentwicklung oder Rekompilierung der Teilaspekte, der Visitor-Mechanismus sorgt automatisch für den Aufruf der entsprechenden Komponenten an den jeweils richtigen Stellen. Dabei können sich die Teilaspekte nicht gegenseitig beeinflussen.

Zusätzliche Komponenten wie Auflösungsmechanismen, Qualifikatoren oder Adaptoren werden auch hier an der neu erstellten `ModelingLanguage` gemäß Tabelle 7.16 registriert.

7.5.5. Entwicklung einer erweiterbaren Sprache

Bei der Entwicklung einer erweiterbaren Sprache sind zunächst verschiedene Komponenten zum Symboltabellenaufbau zu entwickeln. Diese wurden innerhalb dieses Kapitels genauer vorgestellt, Tabelle 7.17 fasst die Komponenten nochmals übersichtsartig zusammen.

Aufbauend auf diesen Komponenten unterstützt das vorgegebene Framework zur Erstellung von Symboltabellen die Kompositionalität von Sprachen. Daher bieten Symboltabellen, die mit diesem Ansatz entwickelt wurden, bereits ein Grundmaß an Kombinierbarkeit. Nichtsdestotrotz lassen sich zusätzliche Richtlinien identifizieren, die die Kompositionalität weiter unterstützen. Diese Richtlinien stellen dabei eine Erweiterung der Richtlinien der vorangegangenen Kapitel dar.

Richtlinie 10: Verzicht auf AST-Rückgriffe

Die Informationen der Symboltabelleneinträge sind oftmals bereits in den AST-Klassen der entsprechenden Sprach-Entität vorhanden. Daher besitzen viele Frameworks den Ansatz, nach der Auflösung von Namen den AST-Knoten direkt oder indirekt über eine Referenz von einem Symboltabelleneintrag aus zu erreichen. Informationen werden dann vom AST-Knoten abgefragt.

Im Sinne der Kompositionalität ist ein solcher Ansatz jedoch nicht empfehlenswert. Der Hauptgrund hierfür besteht in der Typisierung. Ein Zugriff auf den AST-Knoten verlangt statische Kenntnis des Typs des Knotens, um dessen Methoden und Attribute zugreifbar zu machen. Da jedoch eine referenzierte Entität einer Sprache auch in einer anderen - möglicherweise noch unbekannten - Sprache definiert worden sein kann, ist diese Typisierung nicht möglich. Im Beispiel der Interpretation von Zuständen aus Statecharts als Java-Variable ist der definierende AST-Knoten keine Variablen- sondern eine Zustandsdeklaration. Der Entwickler der Java-

Komponente	Aufgabe	Registrierungsort
Menge Namespace-aufspannender Knotentypen	Werden verwendet, um den Namespace-Baum gemäß Schritt 1 des Symboltabellenaufbaus zu erstellen.	`LanguageComponent`
Entries	Symboltabelleneinträge	Keine explizite Registrierung
`Concrete-ASTAnd-Namespace-Visitor`	Traversierung des AST, Instanziierung und Eintragung der Entries	`LanguageComponent`
`Inherited-Entries-Calculator-Client` (optional)	Weitergabe der sichtbaren Entries im Namespace-Baum gemäß Schritt 3 und 7 des Symboltabellenaufbaus	`LanguageComponent`
`IQualifier-Client`	Qualifizierung unqualifizierter Entries gemäß Schritt 4 des Symboltabellenaufbaus	`LanguageComponent`
Grammatik zur Einlesen der Entries	Implizite Verwendung in der Serialisierung der Entries gemäß Schritt 5 des Symboltabellenaufbaus, zusätzlich Registrierung des generierten Parsers zum Einlesen von Modellschnittstellen gemäß Schritt 6 und 9	`LanguageComponent`
`IResolver-Client`	Auflösen von Namen gemäß Schritt 8 des Symboltabellenaufbaus	`LanguageComponent`

Tabelle 7.17.: Zu entwickelnde Komponenten für eine erweiterbare Sprache

Sprache kann dies nicht antizipieren und somit den definierenden Knoten nicht oder nur falsch typisieren.

Neben der Typisierung spielt zusätzlich die Geschwindigkeit eine Rolle: wenn ein Zugriff auf den AST-Knoten eines anderen Modells erfolgen soll, muss dringend das entsprechende Modell eingelesen werden. Der in dieser Arbeit verwendete Ansatz der Modellinterfaces zur Steigerung der Effizienz ist nicht verwendbar, da diese Interfaces eine vom eigentlichen Modell verschiedene Sprache verwenden.

Insgesamt zeigt sich daher, dass Zugriffe auf den AST in Symboltabelleneinträgen kontraproduktiv im Sinne der Kompositionalität und der Effizienz sind. Daher sollten solche Zugriffe vermieden werden.

Richtlinie 11: Vervollständigen der Entries nur bei Bedarf

Nach dem Symboltabellenaufbau sind referenzierte Entries nur dann im vollständigen Zustand, wenn diese im gleichen Modell definiert worden sind, andernfalls befinden sie sich im vollqualifizierten Zustand. Der Übergang in die vollständige Version erfolgt wie beschrieben durch den Aufruf der `toFullVersion()`-Methode. Ein Aufruf dieser Methode sollte nur bei Bedarf erfolgen, d.h. wenn weitergehende Eigenschaften der Entität erfragt werden müssen. Grund hierfür ist, dass der Aufruf das Framework veranlasst, die entsprechende Modellschnittstelle zu laden. Dieser Vorgang ist prinzipiell mit Aufwand und Zeit verbunden, weshalb dieser nur bei Bedarf erfolgen sollte. Anzumerken ist zusätzlich, dass bei geeigneter Programmierung der Entries kein expliziter Aufruf von `toFullVersion()` von außen erfolgen muss, sondern diese intern durch andere Methoden aufgerufen wird. Somit ist der nötige Übergang vom vollqualifizierten auf den vollständigen Zustand gekapselt.

Richtlinie 12: Strukturierung des Komponentenaufbaus

Für den Aufbau der Symboltabellen benötigt das Framework unterschiedliche Komponenten mit spezifischen Aufgaben. Besonders bei komplexen Sprachen ist dabei eine Vielzahl von Komponenten innerhalb des Sprachbausteins zu registrieren. Diese Registrierung sollte dabei schrittweise und strukturiert innerhalb einer Fabrik erfolgen. Hierbei sollten die Komponenten nach Aufgabe (z.B. Qualifikatoren, Auflösungskomponenten etc.) angeordnet in jeweils einer spezifischen Methode angemeldet werden.

Diese Vorgehensweise erleichtert es dabei, die Sprache zu verändern. So können beispielsweise einzelne Methoden in einer Subklasse der Fabrik überschrieben und die angemeldeten Komponenten leicht erweitert oder eingeschränkt werden.

7.6. Verwandte Arbeiten

Die meisten in der Literatur vorhanden verwandten Arbeiten umfassen sowohl das Thema der Symboltabellen-Infrastrukturen als auch darauf aufbauende Kontextbedingungsprüfungen, da beide Themen eine starke Verzahnung aufzeigen. Die folgende Übersicht konzentriert sich jedoch auf Ansätze, die sich auf Symboltabellen und verwandte Strukturen konzentrieren. Weitergehende Ansätze werden im Kapitel der Kontextbedingungen vorgestellt.

Ein bekannter Ansatz zur Integration verschiedener Sprachen wird mit dem Common Language Runtime (CLR) [Gou01] als Teil der .NET Initiave verfolgt. Hier wird eine Zwischenrepräsentation definiert, auf welche verschiedene Sprachen wie C# oder VB.NET transformiert werden. Durch die gemeinsame Basis und das damit verbundene gemeinsame Typsystem ist es möglich, Verbindungen zwischen verschiedenen Eingangssprachen herzustellen und somit sprachübergreifende Symboltabellen zu erstellen. Die Domänen von DSLs sind jedoch oftmals stark verschieden und haben sehr unterschiedliche Konzepte und Entitäten (z.B. Zustände in Statecharts, Tabellen in SQL, Klassen in Klassendiagrammen oder Bausteine eines Systems in Matlab/Simulink), sodass ein integriertes Typsystem für DSLs nicht erfolgversprechend ist.

Einige Ansätze (z.B. [BHD+01, OAW]) zur Definition domänenspezifischer Sprachen bieten verschiedene Möglichkeiten zur Definition von Symboltabellen-Strukturen. Oftmals ist es

hier möglich, Namespaces zu definieren und die darin enthalten Entitäten durch die Angabe bestimmter Regeln zu identifizieren. Diese werden dann automatisch den Namespaces hinzugefügt. Zusätzlich erlauben vorgefertigte und zumeist einfache Auflösungsmechanismen die Suche nach Einträgen. Insgesamt ermöglichen diese Frameworks daher die Definition von Symboltabellen für einfach strukturierte Sprachen, komplexe Auflösungslogiken werden nicht unterstützt.

Ein bekannter Ansatz zur Realisierung von in GPLs eingebetteten DSLs besteht im forwarding [WMBK02], welches als Grundlage des Intentional Programming [SCC06] dient. Hierbei werden die eingebetteten Anteile in Konstrukte der Host-GPL übersetzt, um die dort vorhanden Symboltabellen-Infrastruktur zu nutzen. Insgesamt ist jedoch die Technik des Übersetzens einer eingebetteten DSL in die Host-GPL schwierig, da Fehlermeldungen auf die Ursprungs-DSL zurückgerechnet werden müssen. Des Weiteren werden DSLs oftmals ohne GPL eingesetzt, wodurch dieser Ansatz nicht in allen Szenarien der DSL-Nutzung angewandt werden kann.

7.7. Zusammenfassung

In diesem Kapitel wurde ein weiterer Aspekt der Sprachentwicklung vorgestellt: die Symboltabellen. Diese bilden einen integralen Bestandteil eines sprachspezifischen Werkzeuges und bilden die Grundlage für weitere Aspekte wie die Überprüfung von Kontextbedingungen oder die Codegenerierung.

Ausgangsbasis der Diskussionen bildeten dabei zunächst informelle Betrachtungen, die das Thema näher vorstellten. Darauf aufbauend erfolgte eine theoretische Untersuchung der Strukturen und Arbeitsweisen unter Beachtung des Kompositionalitätsaspektes. Anschließend wurde das entwickelte Framework vorgestellt, wobei auch hier eine Unterstützung der Kompositionalität wesentliches Kernziel bildete. Die Betrachtung der einzelnen Szenarien der kompositionalen Sprachentwicklung beleuchtete zusätzlich die Möglichkeiten zur Kombination von Sprachen bezüglich der Symboltabellen-Infrastruktur.

Insgesamt zeigt dieses Kapitel auf, wie Symboltabellen auf Basis einer einheitlichen und strukturierten Vorgehensweise kompositional gestaltet werden können. Hierbei wurden die benötigten Komponenten zunächst ohne Beachtung einer konkreten Sprachkombination entwickelt. Diese Komponenten konnten anschließend bei der Zusammenführung von Sprachen ohne Änderungen wiederverwendet werden, wobei zusätzliche Artefakte die Aspekte der Komposition einbringen. Hierdurch wird der Wiederverwendungsgrad erhöht und der Aufwand bei der Kombination stark reduziert.

Kapitel 8.

Kompositionale Entwicklung von Kontextbedingungen

Der letzte Schritt bei der Analyse der Eingabe besteht in der Überprüfung von Kontextbedingungen, welche die Korrektheit der Eingabe über die kontextfreie Syntax hinaus prüfen. Zur Umsetzung bedienen sich die Sprachentwickler dabei unterschiedlicher Mechanismen: zum einen werden die Symboltabellen-Infrastrukturen des letzten Kapitels verwendet, um insbesondere Eigenschaften referenzierter Elemente abzufragen. Zum anderen dienen speziell entwickelte Techniken dazu, komplexe Berechnungen in einem gegebenen AST effizient zu ermöglichen. Beide Mechanismen sind innerhalb des entwickelten Ansatzes von zentraler Bedeutung.

Sowohl die dabei unterliegende Architektur als auch die Vorgehensweise bei der Überprüfung von Kontextbedingungen orientieren sich stark am speziellen Fokus dieser Arbeit. Hierbei sind insbesondere Eigenschaften der DSLs und Randbedingungen aufgrund der Kompositionalität der Sprachen von Bedeutung. Diese beeinflussen dabei wesentlich die zur Verfügung gestellten Mechanismen, welche innerhalb dieses Kapitels näher erläutert werden. Hierzu ist das Kapitel wie folgt gegliedert:

- Zunächst erfolgt eine Einführung in die Prüfung von Kontextbedingungen.
- Im Anschluss werden die Anforderungen an die Strukturen und Arbeitsweisen aufgrund des speziellen Fokus dieser Arbeit näher vorgestellt.
- Darauf aufbauend wird die Architektur der Überprüfungen unter Beachtung der identifizierten Anforderungen erläutert.
- Der vierte Abschnitt stellt das Attributgrammatik-System von MontiCore vor und untersucht, wie eine Attributierung sprachübergreifend umgesetzt werden kann.
- In den nächsten Abschnitten erfolgt eine Diskussion der einzelnen Szenarien der kompositionalen Sprachentwicklung bezüglich der Überprüfungen von Kontextbedingungen.
- Abschließend werden verwandte Arbeiten vorgestellt und das Kapitel zusammengefasst.

8.1. Prüfung von Kontextbedingungen

Bei der Erstellung einer Sprache stellt sich zunächst die Frage, welche Fehler bereits durch die Grammatik bzw. durch den Parser abgefangen werden sollen und welche erst in der späteren

Phase der Kontextbedingungsprüfung gefunden werden. Diese Entscheidung ist dabei nicht immer eindeutig sondern abhängig von den Präferenzen des Sprachentwicklers. In vielen Fällen empfiehlt es sich, insbesondere häufig auftretende Fehler in Form expliziter Regeln zuzulassen und in einem späteren Schritt die Eingabe zu analysieren und zu prüfen, ob diese Regeln angewendet wurden [ASU86]. Diese Vorgehensweise erleichtert dem Parser dabei das Erkennen von im eigentlichen Sinne fehlerhafter Eingaben, da dieser die Instanz gemäß der vorliegenden Grammatik als korrekt einstuft. Nach der Überprüfung der Kontextbedingungen wird die Eingabe dennoch als fehlerhaft ausgewiesen. Insgesamt lassen sich durch diese Vorgehensweise jedoch nur Bedingungen behandeln, die anhand der kontextfreien Syntax der Grammatik formulierbar sind.

Über die kontextfreie Syntax hinaus wird bereits ein Teil der Prüfungen durch die Symboltabellen übernommen. Dies betrifft vor allem die existenziellen Bedingungen, welche des Vorhandensein eines referenzierten Elementes verlangen und bei der Qualifizierung unqualifizierter Entries geprüft werden. Daher ist im Schritt der Qualifizierung referenzierter Entries (siehe Abschnitt 7.3.3) im Falle eines Fehlers bereits eine entsprechende Meldung an den Benutzer zu geben.

Neben der Fehlerprüfung aufgrund der Grammatik und der Symboltabellen erfolgen zusätzliche Kontextbedingungsprüfungen in einer weiteren Bearbeitungsphase. Hierzu lassen sich Kontextbedingungen in zwei unterschiedliche Arten einteilen: zum einen existieren Bedingungen, die bereits ohne die Symboltabellen-Infrastruktur prüfbar sind. Hierzu zählen beispielsweise verbotene Modifikatorkombinationen für Methoden in Klassendiagrammen oder das Verbot eines `return`-Ausdrucks mit Wert in einer `void`-Methode in Java. Zum anderen existieren jedoch auch Kontextbedingungen, die eine Symboltabelleninfrastruktur zwingend benötigen. Diese treten meist an Stellen auf, in denen Eigenschaften referenzierter Entries abgefragt werden müssen. Demnach benötigen Kontextbedingungsprüfungen prinzipiell Zugriff sowohl zum AST als auch zur Symboltabelle.

8.2. Anforderungen an Kontextbedingungsprüfungen

Bei der Entwicklung von Kontextbedingungsprüfungen entstehen unterschiedliche Anforderungen sowohl an die Architektur als auch an die Verwendungsmöglichkeiten. Hierbei spielt der Charakter domänenspezifischer Sprachen eine tragende Rolle: im Gegensatz zu GPLs sind die Kontextbedingungen einer DSL typischerweise nicht feststehend, sondern variieren je nach Einsatzgebiet der Modelle oder nach der Art der Codegenerierung (z.B. unterschiedliche Zielsprachen). Aus diesem Grunde entstehen Anforderungen, die über die für GPLs benötigten Funktionalitäten hinausgehen.

Im Folgenden werden die existierenden Anforderungen näher vorgestellt. Anzumerken ist hierbei, dass nicht jede Sprache zwingend den vollen Umfang an Funktionalitäten erfordert. Vielmehr umfassen die aufgeführten Punkte alle möglichen Szenarien der Verwendung von DSLs und sind daher als Obermenge der Anforderungen zu sehen. Weiterhin dienen die Anforderungen zwei verschiedenen Zwecken: zum einen finden sie innerhalb des vorgegeben Frameworks Beachtung, zum anderen sollten sie auch bei der Implementierung konkreter Überprüfungen durch den Sprachentwickler beachtet werden.

Kompositionalität

Wie in allen Aspekten dieser Arbeit steht die Kompositionalität auch bei der Prüfung von Kontextbedingungen im Vordergrund. Dabei sollen die einzelnen Prüfmechanismen für Sprachen separat entwickelt und im Falle der Sprachkombination unverändert unter eventueller Einbindung von Glue Code wiederverwendet werden.

Selektierbarkeit

Für die Eigenschaft der Selektierbarkeit von Kontextbedingungen sind im Wesentlichen zwei Eigenschaften domänenspezifischer Sprachen verantwortlich: erstens werden DSLs - insbesondere bei der Modellierung von Softwaresystemen - in unterschiedlichen Design-Phasen eingesetzt. Hierbei werden in den ersten Phasen typischerweise weniger starke Kontextbedingungen gestellt, da hier Design-Entscheidungen noch nicht getroffen wurden. So kann es beispielsweise vorkommen, dass Klassen in Klassendiagrammen Attribute besitzen, die noch nicht typisiert sind oder deren Typen noch nicht existieren.

Zweitens werden für DSLs oftmals verschiedene Codegeneratoren entwickelt. Je nach Art der Codegenerierung sind einzelne Kontextbedingungen zu aktivieren bzw. zu deaktivieren. Als Beispiel können auch hier Klassendiagramme genannt werden: im Falle einer Java-Codegenerierung wird typischerweise die Mehrfachvererbung verboten, im Falle eine Codegenerierung nach C++ ist sie erlaubt.

Insgesamt stellt sich die Anforderung, einzelne Kontextbedingungen und deren Überprüfungen zu aktivieren und zu deaktivieren. Die Entscheidung, welche Bedingungen geprüft werden, kann dabei sowohl vom Sprachentwickler als auch vom Sprachnutzer erfolgen.

Konfigurierbarkeit

Eine zusätzliche Unterstützung für Sprachentwickler und -nutzer stellt die Konfigurierbarkeit von Kontextbedingungsprüfungen dar. Hierbei werden die Kontextbedingungen nicht nur aktiviert bzw. deaktiviert, sondern mit einer Fehlerstufe (z.B. „Info“, „Warnung“ oder „Fehler“) konfiguriert. Dies hat insbesondere auf die bereits genannten Design-Phasen in der Modellierung Auswirkungen: in frühen Phasen können manche Analysen den Nutzer über die Ergebnisse der Prüfungen informieren bzw. warnen, in späten Phasen wird die Fehlerstufe so angepasst, dass die Eingabe als falsch abgelehnt wird. Hierdurch wird der Sprachnutzer bereits früh über mögliche Fehler informiert, die Eingabe ist jedoch dennoch korrekt bezüglich der Kontextbedingungen.

Gruppierbarkeit

Kontextbedingungen können in vielen Sprachen in zusammenhängende hierarchische Einheiten gruppiert werden. Als Beispiel dienen hierfür die in Abbildung 8.1 dargestellten Kontextbedingungen zu Klassendiagrammen: auf den oberen Ebenen existieren hierbei abstrakte Kontextbedingungen zur logischen Strukturierung, welche sich jeweils aus den darunter liegenden Bedingungen zusammensetzen. Abschließend befinden sich in der untersten Ebene konkrete atomare Überprüfungen.

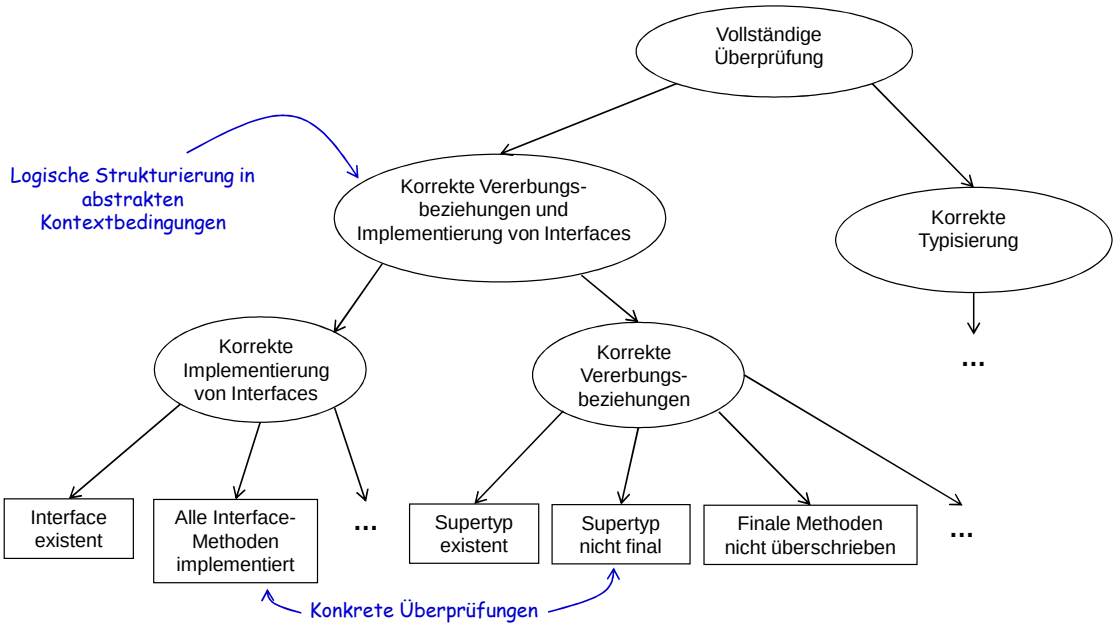


Abbildung 8.1.: Gruppierung von Kontextbedingungen in Klassendiagrammen

Die Gruppierungen anhand abstrakter Kontextbedingungen können zusätzlich in Verbindung mit der Selektier- und Konfigurierbarkeit arbeiten: werden Elemente in den höheren Ebenen aktiviert/deaktiviert oder wird ihnen eine Fehlerstufe zugewiesen, sollte sich diese Eigenschaft transitiv auf die untergeordneten Ebenen auswirken. Dies gilt insbesondere für die atomaren Prüfungen, da nur deren Fehlerstufe bzw. Selektion für die letztendliche Prüfung interessant ist. Insgesamt können hierdurch gezielt logische Einheiten von Kontextbedingungen beeinflusst werden, wobei die konkreten Einzelprüfungen nicht bekannt sein müssen.

Speicherbarkeit

Wie bereits in den vorangegangenen Punkten erläutert wurde, treten unterschiedliche Konfigurationen von Kontextbedingungen aufgrund der Design-Phase oder der spezifischen Codegenerierung auf. Daher ist es sinnvoll, existierende Konfigurationen zu speichern und je nach Bedarf die benötigte Konfiguration einzubinden.

Die Speicherung kann dabei auf zwei Arten erfolgen: einerseits ist es möglich, Konfigurationsdateien zu erstellen. Hier wird in einer spezifischen Sprache festgelegt, welche Kontextbedingungen in welcher Fehlerstufe aktiviert sind. Ein Framework liest anschließend diese Datei ein und konfiguriert die Kontextbedingungsprüfung entsprechend. Andererseits ist es auch möglich, die Konfiguration direkt zu programmieren und in speziellen Klassen (z.B. Fabriken) zu hinterlegen. Im Anschluss wird lediglich die korrekte Fabrik ausgewählt, welche dann die Konfiguration in Form einer geeigneten Objektstruktur liefert.

Erweiterbarkeit

Im Falle einer Sprachkombination ist es möglich, dass zusätzliche Kontextbedingungen auftreten: so ist Mehrfachvererbung in Klassendiagrammen prinzipiell erlaubt, in Verbindung mit Java ist es jedoch sinnvoll, diese zu verbieten. Da auch Sprachen in mehr als einer Ebene kombinierbar sind, ist das Hinzufügen der Kontextbedingungen ebenso hierarchisch zu gestalten.

Das Hinzufügen von Kontextbedingungen ist jedoch nicht nur auf die Situation der Sprachkombination beschränkt. Häufig existieren zusätzliche Bedingungen, die vom Sprachentwickler nicht vorhersehbar sind. Diese umfassen unter anderem anwenderspezifische bzw. spezifisch für eine Nutzergruppe aufgestellte Coding Guidelines.

Aus diesen Gründen muss ein Framework zur Definition von Kontextbedingungen zusätzlich über die Möglichkeit verfügen, sowohl bei der Sprachkombination als auch nach der Auslieferung der Sprache zusätzliche Kontextbedingungen zu definieren und einzubinden. Diese Kontextbedingungen sollten dabei über die gleichen Eigenschaften wie die standardisiert vorgegebenen Prüfungen (Selektierbarkeit, Konfigurierbarkeit etc.) verfügen.

Zentrale Datenaufbereitung

Die Prüfung von Kontextbedingungen ist häufig mit einer Vielzahl aufwendiger Berechnungen verbunden. Diese Berechnungen finden dabei vor allem in der Symboltabelle statt, wobei die Ergebnisse von unterschiedlichen Kontextbedingungen verwendet werden. So müssen beim Java-Ausdruck

```
this.name = new String();
```

zunächst die Variable `name` und deren Typ sowie der Typ `String` aufgelöst werden. Diese Informationen können dann in unterschiedlichen Kontextbedingungen verwendet werden. Beispiel hierfür ist die Prüfung, ob der Typ der Variable `name` zuweisungskompatibel zu `String` ist, ob die Variable `name` als final deklariert wurde oder ob eine parameterloser Konstruktor für `String` existiert.

Aus diesem Beispiel ist ersichtlich, dass es sinnvoll ist, benötigte Daten für Kontextbedingungen zentral zu berechnen und zu speichern. Konkrete Prüfungen können über diese Daten anschließend ohne wiederholte Berechnungen verfügen.

Verständlichkeit für Nutzer

Bei den vorangegangen Punkten zeigte sich, dass nicht nur die Sprachentwickler Kontakt mit den Kontextbedingungen und deren Überprüfungen haben. Vielmehr müssen auch Sprachnutzer in der Lage sein, Kontextbedingungen zu selektieren, zu konfigurieren und zu erweitern. Daher ist eine Implementierung so zu gestalten, dass sie auch von Sprachnutzern verstanden und verwendet werden kann, insbesondere sollten technische Details soweit wie möglich verborgen werden.

Robustheit

Kontextbedingungsprüfungen arbeiten prinzipiell auf Modellen, deren Korrektheit noch nicht sichergestellt ist. Daher ist bei der Erstellung der Überprüfungen darauf zu achten, dass diese robust gegenüber falschen Eingaben sind. So kann es bei einer Überprüfung durchaus vorkommen, dass referenzierte Elemente nicht existieren und somit auch keine tiefergehenden Informationen darüber abgefragt werden können bzw. die Symboltabelle auf eine Anfrage `null` zurückliefert. Hierdurch kann es leicht zu unerwünschtem Verhalten beispielsweise in Form von Exceptions kommen.

Die Robustheit ist bei den existenziellen Prüfungen innerhalb des Symboltabellenaufbaus bereits beachtet, für konkrete Implementierungen von Prüfungen durch den Sprachentwickler können jedoch von Seiten des Frameworks keine Sicherungsmaßnahmen erfolgen. Vielmehr muss der Entwickler in seiner Implementierung selbst auf mögliche Fehlerquellen achten.

8.3. Architektur

Die grundlegende Architektur der Kontextbedingungsüberprüfungen richtet sich nach den im vorigen Teilabschnitt aufgestellten Anforderungen. Abbildung 8.2 zeigt den wesentlichen Aufbau der Strukturen.

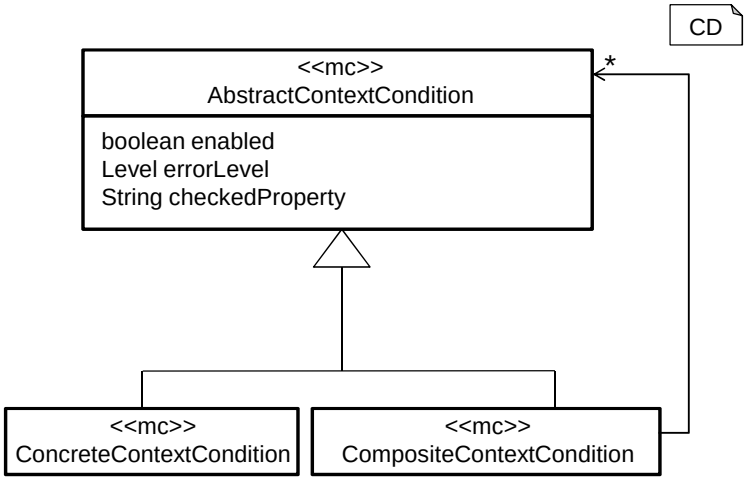


Abbildung 8.2.: Architektur der Kontextbedingungsüberprüfungen

Hierbei wird das Composite-Muster eingesetzt, um eine hierarchische Strukturierung der Kontextbedingungen umzusetzen. Konkrete Prüfungen sind dabei prinzipiell in den Blättern als Subklassen der `ConcreteContextCondition` angeordnet, während zur logischen Strukturierung innere Knoten (`CompositeContextCondition`) dienen.

Weiterhin besitzen alle Kontextbedingungen drei Attribute: `enabled`, `errorLevel` und `checkedProperty`. Ersteres dient dazu Bedingungen zu aktivieren bzw. zu deaktivieren, wobei innere Knoten ein Setzen des Attributes transitiv nach unten weiterleiten. Der default-Wert ist hierbei `true`. Das Attribut `errorLevel` beschreibt die Fehlerstufe der Prüfung und kann

die Werte `Fehler`, `Warnung` und `Info` annehmen, wobei als default der Wert `Fehler` gesetzt ist. Die Weiterleitung erfolgt in gleicher Weise wie bei der Aktivierung. Das dritte Attribut `checkedProperty` wird mit einer deskriptiven Beschreibung der geprüften Kontextbedingung gesetzt. Zusätzlich sollte der Inhalt als Konstante verfügbar sein, da er für die Identifizierung der Bedingung bei Aktivierung oder beim Setzen der Fehlerstufe verwendet wird. Für die Speicherung der Konstanten empfiehlt sich eine Konstantendatei, d.h. eine spezielle Klasse, die Konstanten für alle möglichen prüfbaren Eigenschaften beinhaltet. Die Kontextbedingungen verweisen dann in der `checkedProperty` lediglich auf die entsprechende Konstante der Konstantendatei.

Konfiguration der Prüfungen

Die Konfiguration der Prüfungen erfolgt in zwei Schritten: im ersten Schritt werden alle Kontextbedingungen instanziiert und gemäß der Composite-Struktur zusammengefasst. Die Instanziierung sollte dabei in einer Fabrik-Methode gekapselt werden. Im zweiten Schritt werden anschließend die Kontextbedingungen konfiguriert. Die Konfiguration umfasst dabei sowohl Aktivierung/Deaktivierung als auch die Zuweisung einer Fehlerstufe und sollte ebenfalls in einer geeigneten Methode gekapselt werden. Innerhalb dieser Methode wird dabei eine `Map` erstellt, welche als Schlüssel die Konstante der `checkedProperty` der jeweiligen Kontextbedingung und als Wert die Fehlerstufe ausweist. Abbildung 8.3 zeigt sowohl die Instanziierung als auch die Konfiguration am Beispiel von Java.

Einbindung der Prüfungen

Die Einbindung der Prüfungen in das vorgegebene Framework erfolgt gemäß Abbildung 8.4. Zentral ist auch hier wie bei allen anderen Komponenten ein Registrierungsmechanismus, wobei der Sprachentwickler dafür eine Subklasse von `CoCoClient` in seiner Sprachkomponente zur Verfügung stellt. Die Klasse `CoCoClient` ist dabei ein Visitor, der zusätzliche Informationen vom im Framework vorgegeben `CoCoCheckWorkflow` zur Kontextbedingungsprüfung enthält. Dabei handelt es sich unter anderem um den Zugang zur Symboltabelle, einer Komponente für einheitlichen Fehlermeldungen sowie alle aktivierten Kontextbedingungen mit entsprechendem Fehlerstatus. Innerhalb des `CoCoClient` werden dann die einzelnen AST-Klassen besucht, erforderliche Daten aus der Symboltabelle ausgelesen und an alle einzelnen Kontextbedingungen weitergeleitet. Durch diese zentrale Datenaufbereitung werden wiederholte Berechnungen innerhalb der konkreten Kontextbedingungsprüfungen vermieden.

Wie alle anderen Komponenten können die Kontextbedingungen auf allen Ebenen der Composite-Struktur von Sprachen registriert werden. Hierdurch können einerseits existierende Bedingungen wiederverwendet werden, andererseits ist es so möglich, zusätzliche Bedingungen bei der Kombination von Sprachen einzubinden.

Im Gegensatz dazu werden die Konfigurationen nicht in den Sprachen registriert, sondern bei der Instanziierung des Gesamtwerkzeuges als Parameter angegeben. Der Grund hierfür besteht darin, dass durch diese Vorgehensweise eine flexible Verwendung einer festen Sprachkombination mit unterschiedlichen Konfigurationen der Kontextbedingungsprüfungen ermöglicht wird. Nichtsdestotrotz können Konfigurationen innerhalb einer geeigneten Fabrik-Methode für alle

Java-Quellcode «hw»

```
1 //Instanziierung und Zusammenfassung
2 public AbstractContextCondition createCoCo(){
3   CompositeContextCondition all=
4      new CompositeContextCondition(CoCoConstants.ALL);
5   CompositeContextCondition inh=
6      new CompositeContextCondition(CoCoConstants.INHERITANCE);
7   all.add(inh);
8
9   inh.add(new SuperTypeNotFinalCoCo());
10  //...
11  return all;
12 }
13
14 //Erstellung der Konfiguration
15 public Map<String, Level> createConfig(){
16   Map<String, Level> config=new HashMap<String, Level>();
17   config.put(CoCoConstants.INHERITANCE, Level.ERROR);
18   //Schlüssel ist selber Wert wie checkedProperty von SuperTypeNotFinalCoCo
19   config.put(CoCoConstants.SUPER_TYPE_NOT_FINAL, Level.WARNING);
20   return config;
21 }
22
```

Abbildung 8.3.: Beispiel für Instanziierung und Konfiguration von Kontextbedingungen

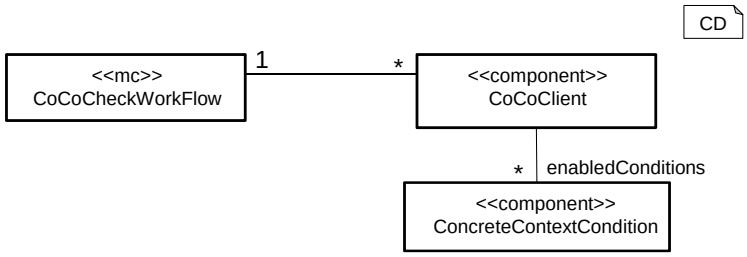


Abbildung 8.4.: Einbindung der Kontextbedingungsüberprüfungen

beteiligten Sprachen hinterlegt und bei Sprachkombination durch Vereinigung zu einer Gesamtkonfiguration zusammengefasst werden. Hierdurch werden existierende Konfigurationen ohne Anpassungen wiederverwendet, gleichzeitig ist eine Anpassung aufgrund der konkreten Sprachkombination möglich.

Kombination und Erweiterung der Prüfungen

Durch die verwendete Composite-Struktur können die Prüfungen für verschiedene Sprachen auf einfache Weise kombiniert werden. Hierzu wird ein neuer Wurzelknoten eingeführt, dessen unmittelbare Kinder die Wurzelknoten der beteiligten Sprachen sind. Auch Erweiterungen

aufgrund einer speziellen Sprachkombination oder anderer Rahmenbedingungen werden ermöglicht. In diesen Fällen wird innerhalb eines Baumes der geeignete Vaterknoten gesucht und an dieser Stelle die neuen Prüfungen als Kinder eingeführt.

8.4. Einsatz des MontiCore-Attributgrammatiksystems

Während der Prüfung von Kontextbedingungen erfolgen häufig komplexe Berechnungen, die sich an der hierarchischen Struktur des ASTs orientieren. Hierbei werden Informationen an verschiedenen Stellen des AST berechnet und aufgrund von definierten Regeln weitergeleitet. Ein Beispiel für solche Berechnungen und deren Verwendung für Kontextbedingungsprüfungen ist die Typisierung von Expressions in Programmiersprachen. So muss für den Java-Ausdruck

```
String s = someMethod() + (3+5);
```

berechnet werden, ob der Typ der rechten Seite der Zuweisung zuweisungskompatibel zum Typ auf der linken Seite ist. Für Berechnung des Typs auf der rechten Seite wird dabei die hierarchische Struktur des AST verwendet. Abbildung 8.5 zeigt die dabei entstehenden Informationsflüsse.

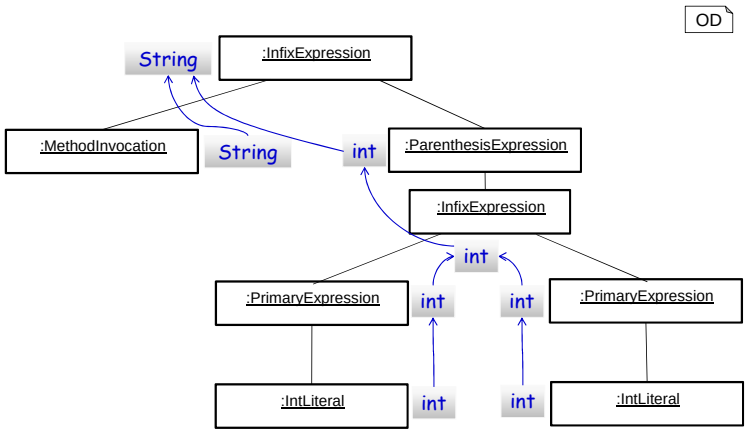


Abbildung 8.5.: Informationsfluss am Beispiel

In dieser Abbildung ist eine verkürzte Variante des ASTs der rechten Seite der Zuweisung abgebildet, wobei aus Übersichtsgründen einige Zwischenknoten ausgelassen wurden. Zusätzlich ist jedem Knoten eine Typisierung gemäß der Java-Regeln [GJS05] zugewiesen. Dabei entsteht diese Typisierung wie folgt: im ersten Schritt wird allen Blättern ein Typ zugewiesen. Bei den int-Literalen ist die Typisierung unmittelbar ersichtlich, beim Methodenaufruf muss zusätzlich eine Auflösung anhand der Symboltabellen erfolgen und somit der Rückgabetyp berechnet werden. Im Anschluss wird der Typ der inneren Knoten aufgrund der Typen der untergeordneten Knoten berechnet. In einigen Fällen kann der Typ dabei übernommen werden, an manchen Stellen (z.B. beim Wurzelknoten des Beispiels) wird der Typ aufgrund einer Regel berechnet.

Zur Umsetzung solcher Berechnungen eignen sich Attributgrammatiken [Knu68]. Hier werden wie bereits in Abschnitt 7.2.1 vorgestellt, Regeln einer Grammatik mit Vorschriften zur Berechnung der Attribute versehen. Attributwerte für konkrete ASTs werden dann automatisch anhand dieser Vorschriften berechnet. Aufgrund technischer Gegebenheiten und insbesondere aufgrund der Kompositionalitätsanforderungen ist die Vorgehensweise innerhalb des MontiCore-Frameworks jedoch verschieden von den in der Literatur beschriebenen Standardansätzen. Daher wird im Folgenden der MontiCore-Ansatz genauer vorgestellt.

8.4.1. Das MontiCore-Attributgrammatik-System

Das MontiCore-Attributgrammatik-System wurde im Wesentlichen innerhalb von [Ste08] entwickelt. Dafür wurde das MontiCore-Grammatikformat um die Möglichkeit erweitert, Attribute zu spezifizieren. Abbildung 8.6 zeigt zwei einfache Attributdefinitionen.

MontiCore-Grammatik «hw»

```
1 package mc.example;
2
3 grammar AttributeExample{
4
5   //Definition der Grammatikregeln
6   ...
7
8   //Definition der Attribute
9   concept Attributes{
10     syn value1: /int;
11     inh value2: /String;
12  }
13 }
```

Abbildung 8.6.: Beispiel für Attributdefinitionen

In dieser Abbildung wird in den Zeilen 9-12 das für Attributdefinitionen verantwortliche Konzept `Attributes` als Erweiterung des MontiCore-Grammatikformates verwendet. Dabei werden zwei Attribute eingeführt: `value1` und `value2`. Zusätzlich wird durch die Angabe `syn` definiert, dass `value1` ein synthetisiertes Attribut vom Typ `int` und `value2` ein inheritiertes Attribut vom Typ `String` ist. Der Schrägstrich vor den Typnamen kennzeichnet wie überall innerhalb des Grammatikformates einen externen Java-Typ, d.h. einen Typ, der nicht aus der Grammatik generiert wird.

Ausgehend von dieser Definition wird eine Klasse `Konnektor` generiert, die einen Zugriff auf die Attributwerte erlaubt. Diese Klasse enthält dabei für jedes definierte Attribut eine `get`-Methode, im obigen Beispiel entstehen somit die Methoden `getValue1` und `getValue2`. Zusätzlich ist diese Klasse intern mit einem Speicher `Storage` verbunden, der Attributwerte zur Vermeidung wiederholter Berechnungen zwischenspeichert. Der Zugriff auf die Attributwerte erfolgt demnach prinzipiell über den `Konnektor`, welcher die Anfrage an den Speicher weiterleitet. Dieser entscheidet dann, ob der Wert neu berechnet werden muss oder bereits vor-

handen ist.

Wie in Abbildung 8.6 ersichtlich ist, werden innerhalb der Grammatik keine Regeln für die Berechnung der Attributwerte definiert. Die Berechnungen werden stattdessen innerhalb einer Java-Klasse formuliert, welche ein vorgegebenes Interface implementieren muss. Dabei unterscheidet sich das vorgegebene Interface je nach Art des Attributes: bei synthetisierten Attributen wird der Berechnungsmethode der aktuelle Knoten übergeben, bei inheriten Attributen der Vaterknoten. Im ersten Fall liefert die Berechnungsmethode unmittelbar den berechneten Wert zurück, der dann automatisch im Speicher hinterlegt wird. Im zweiten Fall muss der Wert für jeden Kindknoten berechnet und explizit im Speicher hinterlegt werden.

Einbindung der Berechnungen

Die letzte benötigte Angabe für das Attributgrammatik-Framework besteht in der Information, wie die vom Sprachentwickler programmierte Berechnungsklasse heißt. Diese Information wird nicht in der MontiCore-Grammatik angegeben, sondern in der Sprachdatei. Abbildung 8.7 zeigt ein Beispiel.

MontiCore-Sprachdatei «hw»

```
1 concept Attributes{
2
3   wert{
4      mc.example.AttributeExample.value1 = /mc.example.Value1Calculator
5   }
6
7 }
```

Abbildung 8.7.: Angabe der Berechnungsklasse

In dieser Abbildung wird in Zeile 3 ein Attribut mit dem Namen `wert` eingeführt, dessen Bedeutung im nächsten Abschnitt näher vorgestellt wird. Innerhalb des Blocks der Attributdefinition wird spezifiziert, dass für das Attribut `value1` der Grammatik `mc.example.AttributeExample` aus Abbildung 8.6 die Berechnungsklasse `mc.example.Value1Calculator` verwendet wird.

Aus dieser Definition wird wie bereits bei der Variante innerhalb der Grammatik ein `Konnektor` generiert. Dieser dient auch hier als Schnittstelle für Benutzeranfragen und hat einen internen Verweis auf das `Storage`. Zusätzlich instanziiert der `Konnektor` die Berechnungsklasse und setzt diese Instanz beim Speicher, damit dieser bei Bedarf die konkreten Berechnungen anstoßen kann.

Kombination von Berechnungen

Das in Abbildung 8.7 definierte Attribut `wert` wurde ohne Berechnungsklasse definiert, da der eigentliche Zweck dieses neuen Attributes lediglich die Unterstützung der Spracheinbettung ist: hierbei kann es vorkommen, dass die beteiligten Sprachen Attribute unterschiedlichen Namens

definieren, wobei der Zweck der Attribute jedoch gleich ist. Im obigen Beispiel ist es so vorstellbar, dass in die Ausgangssprache eine weitere Sprache mit einem Attribut `wert1` eingebettet wird. Um beide Attribute zu vereinen, muss die Attributdefinition der Sprachdatei wie in Abbildung 8.8 angepasst werden.

MontiCore-Sprachdatei «hw»

Glue

```
1 concept Attributes{
2
3   wert{
4      mc.example.AttributeExample.value1 = /mc.example.Value1Calculator
5      mc.example.Grammar2.wert1 = /mc.example.Wert1Calculator
6   }
7
8 }
```

Abbildung 8.8.: Kombination von Berechnungen

Diese Abbildung ist dabei eine Erweiterung von Abbildung 8.7, wobei lediglich Zeile 5 hinzugefügt wurde. Diese Zeile definiert dabei die Berechnungsklasse für das Attribut `wert1` der eingebetteten Grammatik.

Insgesamt entstehen durch diese Zusammenlegung der Einzelattribute mehrere Effekte: wird beispielsweise das Attribut `wert1` für einen Knoten der ersten Grammatik abgefragt, leitet ein interner Mechanismus die Anfrage auf das Attribut `value1` um. Hierdurch können Algorithmen, welche für die beteiligten Teilsprachen entwickelt wurden und deren Attributierung verwenden, unverändert übernommen werden. Als zweiter Effekt entsteht ein neues Attribut `wert`, welches einheitlich für die gesamte Sprache definiert ist. Dadurch können Algorithmen, die von der konkreten Sprachkombination abhängen, konsistent auf ein Attribut zugreifen. Die Zusammensetzung der Attributierung und die Umleitung auf die einzelnen Teilattribute werden dabei vor dem Entwickler gekapselt.

8.4.2. Informationsaustausch

Durch die Möglichkeit, bestehende Attribute unter Angabe eines neuen Attributnamens zu vereinen, ist bereits eine Teilproblematik bezüglich der Einbettung gelöst worden. Ein weiteres Problem besteht jedoch in der Typisierung der Attribute: bei der Einbettung von Sprachen kann es hierbei vorkommen, dass zu vereinende Attribute unterschiedlich typisiert sind. Als Beispiel dient hierfür die Einbettung von Java-Blöcken in Statecharts und die Kombination der attributbasierten Pretty-Printer aus Abbildung 8.9.

Im oberen Teil der Abbildung sind hierbei die Klassen dargestellt, die für die Aufnahme des Ergebnisses eines Pretty-Printers dienen, wobei die linke Klasse für Statecharts und die rechte Klasse für Java verantwortlich ist. Im zweiten Teil der Abbildung ist eine Berechnungsmethode für den Code einer Transition in Statecharts dargestellt, der dritte Abschnitt zeigt die Berechnungsmethode für Java-Blöcke.

Anhand der Abbildung ist erkennbar, dass die Attribute für beide Sprachen unterschiedlich benannt sind: für Statecharts heißt das Attribut `code` - erkennbar an der Verwendung der Me-

Java-Quellcode «hw»

```
1 class SCPrinter extends SynthesizedAttributeCalculation{
2
3   public SCCode calc(ASTTransition s){
4     SCCode ret = new SCCode(s.getFrom() + "-");
5     ret.add(getConnector().getCode(s.getAction()));
6     ret.add(new SCCode(">" + s.getTo()+";"));
7     return ret;
8   }
9 }
```

Java-Quellcode «hw»

```
1 class JavaPrinter extends SynthesizedAttributeCalculation{
2
3   public JavaCode calc(ASTBlockStatement s){
4     JavaCode ret = new JavaCode("{");
5     ret.append(getConnector().getPrinting(s.getBody()));
6     ret.add(new JavaCode("}"));
7     return ret;
8   }
9 }
```

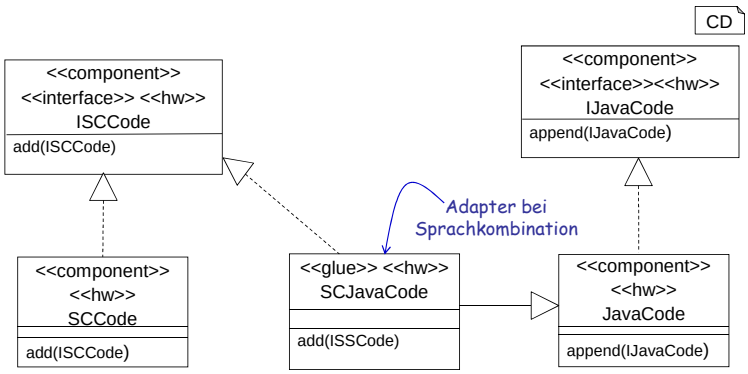
Abbildung 8.9.: Beispiel für Prettyprinter in Form von Attributberechnungen

thode `getCode` in Zeile 5 der mittleren Abbildung. Für Java heißt das entsprechende Attribut `printing` gemäß des Methodenaufrufs `getPrinting` in Zeile 5 des untersten Abschnitts. Diese unterschiedliche Benennung hat aufgrund der im vorigen Abschnitt beschriebenen Namensumsetzung innerhalb des Frameworks jedoch keine Auswirkung: der Aufruf von `getCode` aus den Statecharts heraus wird umgeleitet an die Berechnung von `printing` der Sprache Java.

Trotz der Namensumsetzung kommt es im beschriebenen System zu unerwünschtem Verhalten: die Transition erwartet beim Drucken der Aktion ein Objekt des Typs `SCCode`, der Java-Printer liefert jedoch ein `JavaCode`-Objekt. Da das Framework intern aus technischen Gründen ungetypt arbeitet, entstehen jedoch keine Compile-Fehler, vielmehr wird intern versucht, das `JavaCode`-Objekt als `SCCode`-Objekt aufzufassen. Aufgrund der verwendeten Strukturen ist dies jedoch nicht möglich und der Druck des Java-Anteils ist nicht erfolgreich.

Als Lösungsstrategie empfiehlt sich hierfür die bereits bei der Kombination von Visitoren eingesetzte Architektur aus Abschnitt 6.2.2. Dementsprechend werden Interfaces für die Rückga-

betypen der Berechnungsmethode gebildet und Fabriken zur Instanziierung verwendet. Da die Rückgabetypen der Berechnungsmethoden auf die Interfaces verändert werden müssen, muss auch der Typ des Attributs in der Grammatik auf das Interface umgestellt werden. Zusätzlich werden die Fabriken bei der Kombination von Sprachen so instrumentalisiert, dass sie einen geeigneten Adapter instanziieren und zurückliefern. Abbildung 8.10 fasst das korrekte Vorgehen zusammen.



Java-Quellcode «hw»

```
1 class SCPrinter extends SynthesizedAttributeCalculation{
2
3   //Verwendung des Interfaces als Rückgabetyp
4   public ISCCode calc(ASTTransition s){
5     //Instanziierung per Fabrik
6     ISCCode ret = SCFactory.create(s.getFrom() + "-");
7     ret.add(getConnector().getCode(s.getAction()));
8     ret.add(SCFactory.create(">" + s.getTo()+";"));
9     return ret;
10  }
11 }
```

Java-Quellcode «hw»

```
1 class JavaPrinter extends SynthesizedAttributeCalculation{
2
3   public IJavaCode calc(ASTBlockStatement s){
4     IJavaCode ret = JavaFactory.create("{");
5     ret.append(getConnector().getPrinting(s.getBody()));
6     ret.add(JavaFactory.create("}"));
7     return ret;
8   }
9 }
```

Abbildung 8.10.: Beispiel kompositionaler Attributberechnungen

8.4.3. Zugriff auf zentrale Strukturen

Während der Attributberechnungen werden oftmals zentrale Strukturen des Frameworks benötigt. Hierzu gehören unter anderem der `Resolver` zum Auflösen von Bezeichnern oder die Komponenten zur einheitlichen Fehlermeldung. Um auf diese Komponenten zuzugreifen, empfiehlt sich die Verwendung globaler Attribute. Hierbei werden Attribute innerhalb der Grammatik wie gewohnt definiert, die Art des Attributes wird jedoch auf `global` festgesetzt. Abbildung 8.11 zeigt die Definition für den `Resolver`.

MontiCore-Grammatik «hw»

```
1 concept Attributes{
2   //Setzen des globalen Attributs "resolver" vom Typ mc.Resolver
3   global resolver: /mc.Resolver
4 }
```

Abbildung 8.11.: Angabe der Berechnungsklasse

Da die globalen Attribute einheitliche Werte für alle Knoten eines ASTs besitzen, erfolgt die Berechnung nicht in einer speziellen Berechnungsklasse, sondern kann an beliebiger Stelle implementiert werden. Bei der Instanziierung der Konnektoren wird dann der Attributwert per entsprechender `set`-Methode gesetzt, der Zugriff erfolgt in gleicher Weise wie bei inheriten und synthetisierten Attributen.

8.5. Umsetzung der Szenarien

An dieser Stelle sind alle benötigten Grundlagen für ein Verständnis der Arbeitsweisen der Kontextbedingungsprüfungen vorgestellt worden. In den nächsten Teilabschnitten werden diese auf die verschiedenen Szenarien zur Sprachkombination angewandt.

8.5.1. Sprachvereinigung

Bei der Sprachvereinigung werden die beteiligten Sprachen unter Bildung einer Sprachfamilie im Werkzeug registriert. Hierdurch werden die in den Sprachen vorhandenen Kontextbedingungen automatisch vereint, indem ein neuer Wurzelknoten der Composite-Struktur erstellt wird und die Wurzelknoten der Kontextbedingungen der beteiligten Sprachen als Kindknoten des neuen Wurzelknotens gesetzt werden. Zusätzlich ist es möglich, auf der Ebene der neuen Sprachfamilie weitere Kontextbedingungen zu registrieren, welche dann automatisch als Kind des neuen Wurzelknotens gesetzt werden.

Die in Fabriken hinterlegten Konfigurationen der Kontextbedingungen können weiterhin wiederverwendet werden. Hierzu eignet sich die Erstellung einer neuen Fabrik für die neu gebildete Sprachfamilie. Diese Fabrik liefert dann eine Gesamtkonfiguration unter Einbeziehung der Fabriken der Teilsprachen. Hierdurch wird einerseits die Wiederverwendung der existierenden Fabriken erreicht, andererseits kann auch die neu erstellte Fabrik mit dem gleichen Ansatz wiederverwendet werden. Dies ermöglicht die einfache Bildung einer weiteren Sprachfamilie unter

Einbeziehung der aktuellen Familie. Des Weiteren können auf diese Weise die aufgrund der Bildung der Sprachfamilie hinzugekommen Kontextbedingungen innerhalb der Fabrik-Methoden konfiguriert werden. Tabelle 8.12 fasst die zu entwickelnden Komponenten zusammen.

Komponente	Aufgabe	Registrierungsort
`Abstract-Context-Condition`	Eventuelle zusätzliche Kontextbedingungen aufgrund spezieller Sprachkombination.	`LanguageFamily`
Fabrik zur Erstellung der Konfiguration	Erstellung einer Gesamtkonfiguration für alle Kontextbedingungen. Eventuell Konfiguration der zusätzlichen Bedingungen.	Keine explizite Registrierung, sondern lediglich Parametrisierung des Werkzeuges mit dem Rückgabewert der Fabrikmethode.

Tabelle 8.12.: Zu entwickelnde Komponenten für die Sprachvereinigung

Sowohl bezüglich der Einbindung der Kontextbedingungen durch den `CoCoClient`, als auch bezüglich der Artefakte der attributierten Grammatik können bei der Sprachvereinigung keine unerwünschten Effekte entstehen. Beide Komponenten sind auf der Ebene der einzelnen Sprachen registriert, das Framework wählt automatisch anhand der Sprache des aktuellen Modells die korrekten Komponenten aus.

8.5.2. Nichtterminal-Erweiterung

Bei der Nichtterminal-Erweiterung wird eine neue `ModelingLanguage` durch Vereinigung der Teilsprachen gebildet. Hierbei kommt es zu ähnlichen Effekten wie bei der Sprachvereinigung: die Kontextbedingungen werden unter Bildung eines neuen Wurzelknotens vereint, wobei auch hier neue Kontextbedingungen aufgrund der Sprachkombination registriert werden können. Die Konfigurationen der Kontextbedingungen sind wiederum in einer Fabrik unter Einbeziehung der vorhandenen Fabriken zu vereinen. Zusätzlich sind in dieser neuen Fabrik die hinzugekommenen Kontextbedingungen konfigurierbar.

Die `CoCoClients` werden innerhalb des Werkzeuges als Visitoren für das gegebene Modell eingesetzt. Da der Visitormechanismus auf Kompositionalität ausgelegt ist, wird die unveränderte Übernahme der Visitoren ermöglicht. Insbesondere besucht dabei jeder `CoCoClient` nur die Knoten aus der für ihn vorgesehenen Sprache wodurch unerwünschte Überschneidungen ausgeschlossen werden.

Bezüglich der Attributierung der Grammatik unterstützen die vorhandenen Kompositionalitätsmechanismen die Nichtterminal-Erweiterung. Hierbei kommen vor allem die vorgestellten Konzepte des Attributgrammatik-Systems zum Tragen: zum einen können Attribute unterschiedlichen Namens unter Einführung eines neuen Attributes in der Sprachdatei vereint werden. Zum anderen ermöglicht das spezifische Design der Typisierung der Attribute einen Austausch der Berechnugsergebnisse über Sprachgrenzen hinweg. Hierbei muss lediglich ein Adapter zur In-

tegration der Rückgabetypen der Berechnungen entwickelt werden, von den Fabriken sind anschließend Instanzen dieser Adaptoren zurückzuliefern. Insgesamt sind daher die in Tabelle 8.13 dargestellten Artefakte zu entwickeln.

Komponente	Aufgabe	Registrierungsort
`Abstract-Context-Condition`	Eventuelle zusätzliche Kontextbedingungen aufgrund spezieller Sprachkombination.	`ModelingLanguage`
Fabrik zur Erstellung der Konfiguration	Erstellung einer Gesamtkonfiguration für alle Kontextbedingungen. Eventuell Konfiguration der zusätzlichen Bedingungen.	Keine explizite Registrierung, sondern lediglich Parametrisierung des Werkzeuges mit dem Rückgabewert der Fabrikmethode.
Adapter für Attributierungen	Integration der Rückgabetypen von Attributberechnungen gemäß Abbildung 8.10.	Keine explizite Registrierung.
Fabriken (nur Anpassung)	Umstellung der Fabriken auf Instanzierung der Adaptoren.	Keine explizite Registrierung.

Tabelle 8.13.: Zu entwickelnde Komponenten für die Nichtterminal-Erweiterung

8.5.3. Nichtterminal-Ersetzung

Da die Nichtterminal-Ersetzung bezüglich der Composite-Struktur von Sprachen in gleicher Weise wie die Nichtterminal-Erweiterung umgesetzt wird, ist die Unterstützung seitens der Infrastruktur für Kontextbedingungen ebenso gegeben. Insbesondere werden die Prüfungen und die Konfigurationen durch die gleichen Techniken unterstützt, zusätzlich können auch hier neue Prüfungen und Konfigurationen hierfür eingebunden werden.

Der wesentliche Unterschied zur Nichtterminal-Erweiterung besteht darin, dass die vom ersetzten Nichtterminal aufgespannten Teil-ASTs nicht mehr in der veränderten Version auftauchen können. Hierdurch entfallen auch Kontextbedingungen, die die Korrektheit für solche Teilbereiche prüfen. Da der Mechanismus auf Visitor-Basis beruht, werden die entsprechenden Methoden jedoch nicht aufgerufen. Für die Berechnung der Attribute gilt Ähnliches: auch diese Berechnungen sind innerhalb von Methoden gekapselt, deren Parametertypen den jeweiligen AST-Knotenklassen entsprechen. Daher werden diese Methoden nicht aufgerufen, sondern die Methoden der eingesetzten neuen Sprache. Wie bereits bei der Nichtterminal-Erweiterung unterstützen die Möglichkeit der Einführung neuer Attribute in der Sprachdatei und das Design der Attributtypisierung das Szenario der Nichtterminal-Ersetzung. Dementsprechend gilt Tabelle 8.13 auch für die Nichtterminal-Ersetzung.

8.5.4. Einbettung einer einzelnen Sprache und Einbettung von Sprachalternativen

Sowohl bei der Einbettung einer einzelnen Sprache als auch bei der Einbettung von Sprachalternativen wird eine neue `ModelingLanguage` durch Vereinigung der beteiligten Sprachen gebildet. Auch hier werden die Kontextbedingungen anhand eines neuen Wurzelknotens vereint, das Einfügen neuer Prüfungen ist hierbei ebenfalls möglich. Die Konfigurationen werden durch die Bildung einer neuen Fabrik unter Einbeziehung der ursprünglichen Fabriken zusammengefasst, neue Kontextbedingungen können zusätzlich konfiguriert werden.

In beiden Szenarien werden die `CoCoClients` in einer Initialisierungsphase instanziiert und beim `CoCoCheckWorkflow` registriert. Dies hat zur Folge, dass für jede Eingabe alle `CoCoClients` aktiv sind. Insbesondere bei der Einbettung von Sprachalternativen sind auch diejenigen aktiviert, die im eigentlichen Sinne nicht benötigt werden, da eine andere eingebettete Sprache vom Benutzer gewählt wurde. Wie bei der Nichtterminal-Erweiterung bereits diskutiert wurde, hat diese Situation aufgrund des Visitor-Mechanismus jedoch keine negativen Auswirkungen auf die Prüfungen. Die gleiche Argumentation gilt für die Attributgrammatiken, da auch hier nur diejenigen Berechnungen verwendet werden, deren Sprachanteile im AST vorkommen. Zusätzlich werden beide Szenarien durch die Möglichkeit der Einführung neuer Attribute in der Sprachdatei und das Design der Attributtypisierung unterstützt. Daher gilt auch in diesen Szenarien Tabelle 8.13.

8.5.5. Entwicklung einer erweiterbaren Sprache

Bei der Entwicklung einer erweiterbaren Sprache entstehen zusätzliche Richtlinien, die der Sprachentwickler beachten sollte. Diese stellen dabei eine Fortsetzung der in den vorangegangenen Kapiteln aufgestellten Richtlinien dar.

Richtlinie 13: Verwendung der vorgeschlagenen Architektur für Attributberechnungen

Die in Teilabschnitt 8.4.2 vorgeschlagene Architektur der Typisierung von Attributen dient der Kompositionalität, da verschieden typisierte Attribute in einfacher Art und Weise vereint werden können. Daher empfiehlt es sich, diese Architektur prinzipiell anzuwenden. Dies gilt insbesondere auch dann, wenn eine konkrete Sprachkombination nicht vorhersehbar ist, da der Mehraufwand bei der Verwendung der Architektur sehr gering ist, der Nutzen bei Sprachkombination jedoch den Aufwand um ein Vielfaches übersteigt.

Richtlinie 14: Korrekte Programmierung der Attributberechnungen

In den Abbildungen 8.9 und 8.10 wurden einige Beispiele zur Berechnung von Attributen aufgezeigt. Hierbei wurde das Attribut `printing` für den untergeordneten Knoten des Blockkörpers innerhalb der Attributberechnung für einen Java-Block durch den Aufruf

```
ret.append(getConnector().getPrinting(s.getBody()));
```

abgefragt. Da die Berechnung eines Attributes für alle Knoten einer Sprache meist in einer einzelnen Klasse gekapselt ist, würde der folgende Aufruf zum gleichen Ergebnis führen:

```
ret.append(calc(s.getBody()));
```

Hierbei wurde zur Berechnung im Gegensatz zur ersten Version nicht der Umweg über den Konnektor gewählt, sondern die Methode der eigenen Klasse direkt aufgerufen. Diese Vorgehensweise hat jedoch zwei entscheidende Nachteile: zum einen wird die Speicherung und Wiederverwendung bereits berechneter Attribute innerhalb des Konnektors umgangen und der Wert jedesmal neu berechnet. Zum anderen wird im Falle der Nichtterminal-Erweiterung und der Nichtterminal-Ersetzung immer die Berechnung der aktuellen Klasse verwendet, insbesondere auch für das erweiterte/ersetzte Nichtterminal. Der Grund hierfür ist, dass das erweiternde bzw. ersetzende Nichtterminal Subklasse des originalen Nichtterminals ist und somit die Typisierung der Berechnungsmethode immer noch zutrifft. Dieses Verhalten ist jedoch meist nicht erwünscht, da typischerweise die Berechnungsmethode der erweiternden bzw. ersetzenden Sprache verwendet werden soll. Durch die Verwendung des Konnektors ist dabei stets die Umleitung gesichert, beim direkten Aufruf wird sie hingegen umgangen. Insgesamt empfiehlt es sich daher, bei der Berechnung der Attribute untergeordneter bzw. übergeordneter Knoten den Konnektor zu verwenden.

Richtlinie 15: Robustheit der Attributberechnungen und Kontextbedingungsprüfungen

Sowohl die Attributberechnungen als auch die Kontextbedingungsprüfungen müssen mit inkorrekten Eingaben arbeiten können und sind daher robust gegenüber Fehlern im Modell zu gestalten. Zusätzlich sind hierbei keine impliziten und vor allem keine unnötigen Annahmen über die Struktur des Modells zu machen, da diese sich bei Veränderung der Sprache etwa durch Nichtterminal-Erweiterung oder -Ersetzung ändern kann. Daher sollten in den Attributberechnungen nur Eigenschaften des aktuellen Knotens abgefragt und auf die Traversierung des ASTs verzichtet werden. Für weitergehende Informationen sollte entweder der Konnektor oder die Symboltabelle genutzt werden, da diese anhand der inneren Strukturen von der konkreten Sprachzusammensetzung abstrahieren. Die gleiche Argumentation gilt auch für die Kontextbedingungsprüfungen.

8.6. Verwandte Arbeiten

Die verwandten Arbeiten bezüglich der Kontextbedingungsprüfungen lassen sich in zwei Kategorien unterteilen: zum einen existieren Ansätze für die Formulierung von Constraints, zum anderen gibt es Arbeiten zum Thema attributierte Grammatiken. Dabei bieten die verschiedenen Ansätze unterschiedliche Vor- und Nachteile, sind jedoch im Wesentlichen auf ihre Hauptaufgaben beschränkt und bieten keine oder nur sehr eingeschränkte Unterstützung zur kompositionalen Sprachentwicklung. Für beide Kategorien werden nachfolgend bekannte Stellvertreter aufgeführt.

Zur Formulierung von Constraints für Modelle verschiedener Sprachen lässt sich die Object Constraint Language [OMG10a] einsetzen. So ist es in Verbindung mit der abstrakten Syntax einer Sprache auf Basis von Klassendiagrammen möglich, OCL zur Spezifikation von Kontextbedingungen einzusetzen. Da jedoch zur vollständigen Prüfung die abstrakte Syntax allein nicht ausreicht und zusätzlich Infrastrukturen wie Symboltabellen mit einbezogen werden müssen, ist die OCL speziell für komplexe Sprachen weniger geeignet. Für einfache DSLs lassen sich jedoch effizient Kontextbedingungen formulieren.

Einen ähnlichen Ansatz zur Formulierung von Constraints verfolgt die Sprache *Check* von *openArchitectureWare* [OAW]. Hierbei werden in OCL-ähnlicher Syntax Constraints definiert, zusätzlich verfügt diese Sprache über spezielle Mechanismen für die Prüfung von Kontextbedingungen. Unter anderem ist es hier möglich, lesbare Fehlermeldungen im Falle von Bedingungsverletzungen vorzugeben. Da jedoch auch *Check* eine spezielle DSL zur Formulierung von Kontextbedingungen darstellt, sind die Einschränkungen ähnlich wie bei der OCL.

Die *Epsilon Validation Language (EVL)* des Epsilon-Frameworks [KPP06b] verwendet zunächst auch eine OCL-ähnliche Notation zur Formulierung von Constraints. Um jedoch die Einschränkungen auszugleichen, können hier Java-Anteile mit eingewoben werden [KPP08]. Dadurch werden komplexe Berechnungen oder Zugriffe auf Symboltabellen in Java formuliert, andere Anteile erfolgen in EVL. Hierdurch können beliebig komplexe Prüfungen formuliert werden.

Der in [WS08] beschriebene Ansatz kann neben der Komposition der abstrakten Syntax teilweise auch zur Überprüfung von Kontextbedingungen über Sprachgrenzen hinweg eingesetzt werden. Durch explizite provided und required Interfaces wird dabei beschrieben, welche Anforderungen Teilmodelle an die Umgebung stellen, damit das Gesamtmodell valide wird. Der Ansatz ist jedoch auf einfache Strukturen wie das Vorhandensein bestimmter Klassen im Metamodell beschränkt. Komplexere Kontextbedingungen sind nicht ausdrückbar.

Bezüglich der attributierten Grammatiken lassen verschiedene Erweiterungen der klassischen Variante nennen. So erlauben *Referenced Attribute Grammars (RAGs)* [Hed00] zusätzlich Attribute, deren Werte Knoten des abstrakten Syntaxbaumes sind und somit als Form von Referenzen anstelle von Symboltabelleneinträgen genutzt werden können. Auch *Higher Order Attribute Grammars* [Vog93] erlauben Knoten des Syntaxbaumes als Werte von Attributen. Bei beiden Ansätzen wird versucht, Symboltabellen durch Attribute zu ersetzen, wodurch die Berechnungen und Prüfungen einzig durch Attributgrammatiken ermöglicht werden.

Weiterhin existieren verschiedene Frameworks zur Definition von Sprachen, die zusätzlich über ein Attributgrammatiksystem verfügen. Beispiel hierfür sind unter anderem Lisa [MŽLA99] und JastAdd [EH07]. Diese konzentrieren sich aber auf ihre spezifische Implementierung von Attributgrammatiken und behandeln die Komposition von Sprachen nicht.

8.7. Zusammenfassung

In diesem Kapitel wurde der entwickelte Ansatz zur Prüfung der Kontextbedingungen vorgestellt. Dazu erfolgte zunächst eine Einführung in die Prüfung von Kontextbedingungen. Anschließend wurden die speziellen Anforderungen aufgrund der Eigenschaften von DSLs und der kompositionalen Entwicklung von Sprachen diskutiert. Hierbei zeigte sich, dass sich die entste-

henden Anforderungen deutlich von Anforderungen für allgemeine Programmiersprachen unterscheiden. Insbesondere Konfigurierbarkeit, Selektierbarkeit, Kompositionalität und Erweiterbarkeit sind Eigenschaften, die aufgrund des speziellen Fokus von großer Bedeutung sind.

Die anschließend vorgestellte Architektur der Kontextbedingungsprüfungen wurde unter Beachtung der erarbeiteten Anforderungen entwickelt. Durch die Verwendung des Composite-Ansatzes wurden Erweiterbarkeit, Gruppierbarkeit und Konfigurierbarkeit erreicht. Weiterhin ist es auf diese Weise möglich, Kontextbedingungen verschiedener Sprachen wiederzuverwenden und bei Bedarf zu erweitern. Die zusätzliche Datenaufbereitung vermeidet wiederholte Berechnungen und steigert somit die Effizienz des Ansatzes.

Als weiterer Teil der Kontextbedingungsprüfungen wurde das MontiCore-Attributgrammatiksystem vorgestellt. Hierbei ist es möglich, Berechnungen für einzelne Sprachen zu erstellen und insbesondere bei Einbettung wiederzuverwenden. Dabei wurde aufgezeigt, wie anhand einer speziellen Architektur Informationsaustausch über Sprachgrenzen hinweg erfolgen kann. Der Aufwand beim Einsatz dieser Architektur ist um ein Vielfaches kleiner als der Ertrag, welcher bei der Kombination konkreter Sprachen entsteht.

Die abschließende Diskussion der einzelnen Szenarien zeigt auf, wie die zur Verfügung gestellten Mechanismen auf konkrete Problemstellungen angewandt werden können. Dabei zeigte sich, das sich die Vorgehensweisen in den einzelnen Szenarien kaum unterscheiden. Dadurch entsteht der Vorteil für den Sprachentwickler, dass er für alle Anwendungen gleiche Mechanismen anwendet und nicht für jede neue Problemstellung verschiedenartige Techniken erlernen und verstehen muss.

Insgesamt bietet das entwickelte System eine breite Unterstützung für kompositionale Kontextbedingungsprüfungen. Sowohl die einzelnen Bedingungen als auch die Berechnungen für die Attributierung der Grammatik werden separat entwickelt und im Falle von Sprachkombinationen wiederverwendet. Zusätzlich werden sowohl die spezifischen Anforderungen der DSL-Welt als auch die konkreten Szenarien der kompositionalen Sprachentwicklung unterstützt.

Teil III.

Weiterführende Konzepte und Anwendungen

Kapitel 9.

Kompositionale Entwicklung von Werkzeugen

Die Erstellung sprachspezifischer Werkzeuge wie Editoren ist ein wichtiger Teil der Sprachentwicklung. Durch eine gute werkzeugtechnische Unterstützung entsteht nicht nur Komfort für den Sprachnutzer, vielmehr kann auch die Effizienz bei der Verwendung der Sprache erheblich gesteigert werden [KKV08].

Die Entwicklung sprachspezifischer Werkzeuge ist jedoch oft mit erheblichem Aufwand verbunden [KKV09]. Zwar bieten erweiterbare IDEs wie Eclipse [Ecl, GB03] eine gute Basis für die Entwicklung eigener Werkzeuge, die Kosten für die Einarbeitung in deren Erweiterungsmechanismen und die dazugehörigen APIs sind dennoch sehr hoch. Aus diesem Grunde wird insbesondere bei domänenspezifischen Sprachen mit geringer Nutzergruppe oftmals auf eine Werkzeugentwicklung verzichtet [Kle07b]. Um dies zu vermeiden, ist die Vereinfachung der Werkzeugentwicklung ein wesentliches Ziel dieses Kapitels.

Das Hauptziel besteht deshalb auch hier in der Unterstützung der Kompositionalität. Hierbei sollen Werkzeuge für Einzelsprachen zunächst unabhängig voneinander entwickelt und bei Sprachkombination mit geringem Aufwand kombiniert werden können. Diese Vorgehensweise bildet dabei einen weiteren wesentlichen Aspekt der Kompositionalität in der Sprachentwicklung.

Insgesamt gliedert sich dieses Kapitel wie folgt auf:

- Zunächst erfolgt eine grundlegende Einführung in die Eclipse-IDE, deren Erweiterungsmechanismen und die zur Verfügung gestellten Funktionen eines sprachspezifischen Werkzeugs.
- Im zweiten Teilabschnitt wird erläutert, wie Werkzeuge im entwickelten Ansatz spezifiziert werden und wie das Generat unter Beachtung der Kompositionalität aufgebaut ist.
- Der dritte Teilabschnitt untersucht die Szenarien kompositionaler Sprachentwicklung.
- Abschließend werden verwandte Arbeiten vorgestellt und der Ansatz zusammengefasst.

9.1. Eclipse und sprachspezifische Plugins

Eclipse ist eine IDE (*Integrated Development Environment*) für viele verschiedene Anwendungen und Sprachen. Diese reichen von Modellierungssprachen wie der UML über Programmier-

sprachen wie Java oder C++ bis hin zu domänenspezifischen Sprachen, die auch vom Anwender selbst entwickelt werden können. Aufgrund seiner Plattformunabhängigkeit, der Verfügbarkeit vieler Erweiterungen und seines modularen Aufbaus hat sich Eclipse in den letzten Jahren zu einer der meistverwendeten IDEs entwickelt und wird daher auch als Basis für die Generierung von sprachspezifischen Werkzeugen innerhalb dieser Arbeit verwendet. Bevor jedoch die Werkzeuggenerierung näher diskutiert wird, erfolgt zunächst eine Einführung in die relevanten Aspekte der Eclipse-IDE.

9.1.1. Aufbau einer Workbench

Die Eclipse-Workbench bildet die Gesamtheit der gegenüber dem Benutzer dargestellten Teilfunktionalitäten der IDE. Hierfür ist die Workbench aus vielen Einzelteilen aufgebaut, wobei die eingebunden Einzelteile abhängig von der konkreten Anwendung sind. Da die generierten Werkzeuge auf die Entwicklung mit einer bestimmten Sprache abzielen, wird als Beispiel die Ansicht des *Java Development Toolkits (JDT)* verwendet. Hierbei sind die einzelnen Teilaspekte auf die Verwendung von Java als Programmiersprache spezialisiert. Abbildung 9.1 zeigt die Workbench während der Entwicklung in einem Java-Projekt.

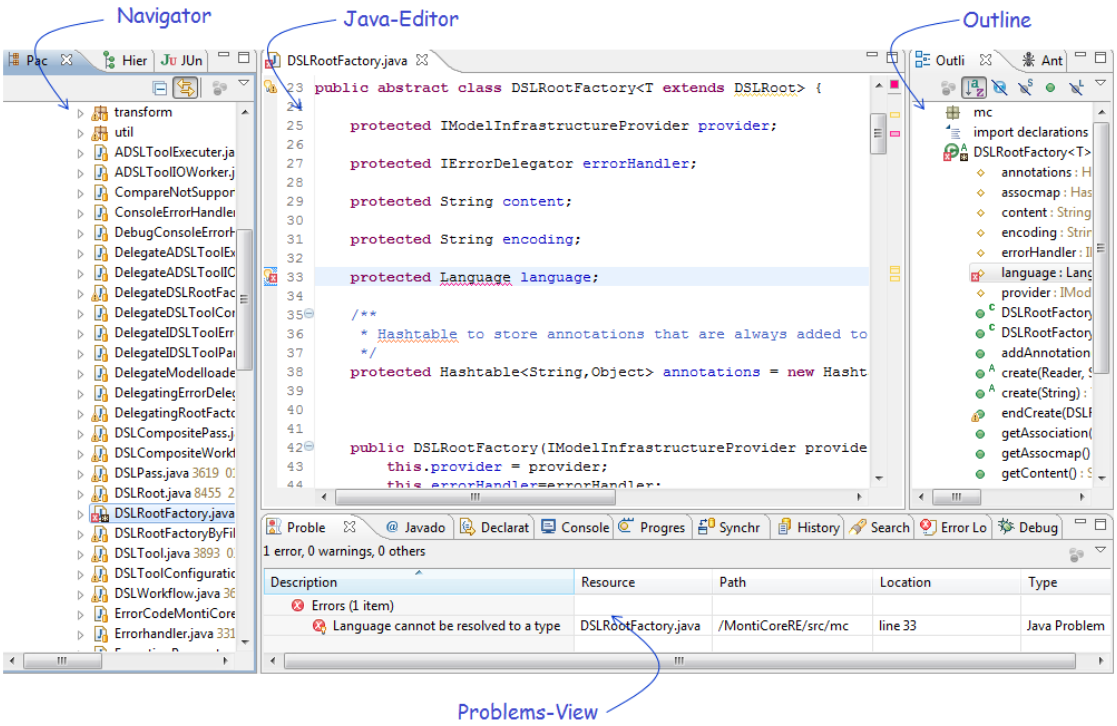


Abbildung 9.1.: Exemplarischer Aufbau der JDT-Workbench

Die in dieser Abbildung dargestellten Teilfunktionalitäten sind:

- Ein Navigator, der existierende Projekte mit deren Inhalt darstellt.
- Der Java-Editor, welcher die aktuell bearbeitete Klasse beinhaltet.
- Eine Outline, die den Inhalt des Editors hierarchisch darstellt.
- Eine Problems-View mit eventuellen Fehlermeldungen.

Die dargestellten Elemente bieten sprachabhängige Funktionalitäten, die wesentlich zum Komfort und zur Effizienzsteigerung bei der Eingabe beitragen. Deshalb stehen diese Funktionalitäten für fast jede in Eclipse integrierte Sprache zur Verfügung und sollten auch für domänenspezifische Sprachen entwickelt werden. Die dabei verwendeten Eclipse-Mechanismen werden in den nächsten Teilabschnitten näher vorgestellt.

9.1.2. Innere Architektur

Eclipse unterliegt einem streng modularen Aufbau. Grundlage hierfür besteht in einem minimalen Kern, der lediglich Basisfunktionalitäten zur Verfügung stellt. Zu diesen Basisfunktionalitäten gehört unter anderem ein System, welches die Erweiterbarkeit der IDE durch Plugins unterstützt. Alle anderen Elemente und Funktionen sind durch solche Plugins realisiert und eingebunden.

Bei der Erstellung eines Plugins sind dabei von Eclipse vorgegebene *Extension Points* zu verwenden. Diese Extension-Points beschreiben dabei alle nötigen Informationen, die von Eclipse zur Einbindung des Plugins in die Workbench benötigt werden. Im Beispiel der sprachspezifischen Editoren umfasst ein solcher Extension Point unter anderem die Klasse des Editors, die Dateiendung der Dateien, für die der Editor verwendet werden soll und eine eindeutige ID des Plugins. Diese Angaben erfolgen in der Konfigurationsdatei `plugin.xml` des Plugins (siehe Abbildung 9.2), welche vom Plattformkern geladen und interpretiert wird. Zusätzlich zur Konfigurationsdatei sind vom Pluginentwickler verschiedene Klassen zu entwickeln, welche das spezifische Verhalten des Plugins implementieren und von Eclipse vorgegebene Interfaces oder Basisklassen implementieren bzw. erweitern.

Da der Aufbau der Eclipse-Plattform modular und hierarchisch ist, können Plugins nicht nur Extension Points nutzen, sondern zusätzliche Extension Points definieren. Hierdurch kann eine gemeinsame Basisfunktionalität zur Verfügung gestellt werden, welche anschließend von verschiedenen spezifischen Plugins gemeinsam genutzt werden kann.

9.1.3. Bereitgestellte Funktionalitäten

Die Plugins einer mittels MontiCore entwickelten Sprache umfassen die in Abbildung 9.1 dargestellten Funktionalitäten, zusätzlich werden jedoch auch andere Möglichkeiten zur Unterstützung des Sprachanwenders zur Verfügung gestellt. Insgesamt umfassen die Plugins folgende Funktionen:

Konfigurationsdatei

```
<plugin>
  <!-- genutzter Extension Point -->
  <extension point="org.eclipse.ui.editors">
    <editor
      <!-- Hauptklasse des Editors -->
      class="mc.automaton.AutomatonEditor"
      <!-- Dateiendung -->
      extensions="aut"
      <!-- Icon -->
      icon="automaton.gif"
      <!-- Eindeutige ID -->
      id="AutomatonDSLEditor"
      <!-- Name des Editors -->
      name="AutomatonDSLEditor"/>
  </extension>
</plugin>
```

Abbildung 9.2.: Beispiel für eine Konfigurationsdatei plugin.xml

- Ein sprachspezifischer Editor mit unterschiedlichem Syntaxhighlighting für Keywords und Code-Regionen.
- Konfigurierbare Unterstützung für das Zusammen- bzw. Auseinanderklappen (*Folding*) von bestimmten Abschnitten des Modells.
- Eine konfigurierbare Outline mit hierarchischer Zusammenfassung des Modells und selektierbaren Elementen, deren Selektion den Cursor auf den entsprechenden Abschnitt im Modell bewegt.
- Unterstützung für Fehlermeldungen in der Problems-View, deren Selektion die entsprechende Datei öffnet und zum relevanten Modellabschnitt springt.
- Unterstützung für Einbringung von Menüpunkten in das Kontextmenü des Editors.
- Unterstützung für Einbringung von Menüpunkten in das Kontextmenü des Navigators.
- Unterstützung für die Formatierung von Modellen.

In einem Ansatz, der die Eclipse-Funktionalitäten direkt verwendet, würden sich folgende Aufgaben für den Sprachentwickler ergeben:

1. Verständnis der Eclipse-IDE und dessen Erweiterungsmechanismen.
2. Verständnis aller nötigen Extension Points und der von Eclipse vorgegebenen Bibliotheken.
3. Implementierung aller nötigen Elemente unter Beachtung der Kompositionalität.

4. Aktualisierung der Implementierungen bei Änderung der DSL.

5. Aktualisierung der Implementierungen bei Änderung von Eclipse.

Da diese Aufgaben mit erheblichem Aufwand verbunden sind, wird ein hauptsächlich generativer Ansatz verwendet, der bei Bedarf um handprogrammierte Elemente erweitert werden kann. Die Möglichkeiten zur Spezifikation der Editoren und insbesondere die kompositionale Arbeitsweise des Ansatzes werden im Folgenden näher vorgestellt.

9.2. Erstellung von Werkzeugen

Der verwendete Ansatz zur Erstellung sprachspezifischer Werkzeuge bedient sich zweier wesentlicher Mechanismen: zum einen wurde das MontiCore-Grammatikformat um ein Konzept zur Spezifikation von Werkzeugen erweitert, zum anderen wird eine einfache Bibliothek zur Verfügung gestellt, die vom Sprachentwickler als Ausgangsbasis zur Implementierung eigener Funktionalitäten genutzt werden kann.

Aus dem erweiterten Grammatikformat werden hierbei Komponenten generiert, die alle benötigten Extension Points und Basisklassen von Eclipse implementieren. Die zur Verfügung gestellte Bibliothek beinhaltet vor allem Basisklassen und Interfaces, deren zu implementierenden Methoden alle benötigten Informationen für eine spezifische Aufgabe in Form von Parametern beinhalten. Die Ergebnisse der Berechnungen werden intern in Eclipse integriert. Der gewählte Ansatz bietet dabei mehrere Vorteile:

1. Durch die Verwendung des erweiterten Grammatikformates und der zu implementierenden Schnittstellen bzw. Basisklassen wird die interne Arbeitsweise von Eclipse vollständig vor dem Sprachentwickler verborgen. Hierdurch wird der Einarbeitungsaufwand erheblich reduziert.

2. Viele Komponenten eines Werkzeuges sind nicht sprachabhängig und können vollständig implementiert vorgegeben werden. Auch minimal sprachabhängige Komponenten können parametrisiert vorgegeben werden, sodass sich der Aufwand auch hier stark reduziert.

3. Da die Eclipse-Architektur nicht auf Kompositionalität von Sprachen ausgelegt ist, wurden zusätzliche Komponenten entwickelt, welche den Kompositionalitätsaspekt in die Plugins einbringen. Hierzu wurde eine spezielle Architektur entworfen, welche sowohl von den Generaten als auch von der zur Verfügung gestellten Bibliothek verwendet wird. Hierdurch ergibt sich zusätzlich eine einheitliche und standardisierte Architektur aller sprachspezifischen Werkzeuge, wodurch die Kompositionalität zusätzlich unterstützt wird.

4. Durch einen generativen Ansatz und die Vorgabe einer gewissen Abstraktion von Eclipse innerhalb der vorgegebenen Bibliotheken kann leicht auf Änderungen reagiert werden. So ist es beispielsweise möglich, die Plugins bei Sprachänderungen durch kleine Anpassungen an der Spezifikation der Werkzeuge in der Grammatik neu zu generieren, wodurch sich der Aufwand für den Sprachentwickler minimiert. Bei Änderungen von Eclipse erfolgt eine Anpassung des Generators bzw. die Anpassungen der inneren Architektur der

bereitgestellten Bibliotheken, hier muss der Sprachentwickler lediglich die neue Version des Generators oder der Bibliotheken verwenden und gegebenenfalls neu generieren.

9.2.1. Entwicklung eines sprachspezifischen Werkzeugs

Zur Entwicklung eines sprachspezifischen Werkzeugs wurde das Grammatikformat um ein zusätzliches Konzept erweitert. Im Folgenden werden die in diesem Konzept spezifizierbaren Eigenschaften anhand des Java-Editors aus den Abbildungen 9.3 und 9.4 vorgestellt.

MontiCore-Grammatik «hw»

```
1 concept editorattributes{
2
3   //Definition der Schlüsselwörter
4   keywords: abstract, double, int, super, assert, else, ...
5
6   //Definition des Foldings
7   foldable: MethodDeclaration;
8
9   //Definition der Outline: Nichtterminal, Icon, anzuzeigender Text
10  //Anzuzeigender Text ist das Wort "class" gefolgt vom Klassennamen als
11  //Attribut des Nichtterminals
12  segment: ClassDeclaration ("icon/class.gif") show: "class" name;
13 }
```

Abbildung 9.3.: Beispiel der Editorspezifikation in der Grammatik

MontiCore-Sprachdatei «hw»

```
1 concept editor {
2    //Hauptklasse des Java-Tools
3    tool: "mc.javadsl.JavaDSLTool";
4
5    //Dateiendung
6    fileextension: java;
7
8    //auszuführende Workflows beim Einlesen eines Modells
9    syntheses: parse, check;
10
11   //Menüpunkt im Editor
12   menuitem: Generate JavaDoc ("mc.javadsl.GenJavaDocAction");
13
14   //Menüpunkt im Navigator
15   navigatoritem: Organize Imports ("mc.javadsl.OrganizeImportsAction");
16 }
```

Abbildung 9.4.: Auszug aus der Editorspezifikation in der Sprachdatei

Standardfunktionalitäten.

Alle generierten Editoren verfügen über Standardfunktionalitäten wie Copy und Paste oder Suchen und Ersetzen. Hierfür sind keine speziellen Angaben des Sprachentwicklers erforderlich.

Syntaxhighlighting.

Das Syntaxhighlighting umfasst zwei unterschiedliche Aspekte: zum einen wird das Modell in Kommentar- und Coderegionen unterteilt, zum anderen können Schlüsselwörter hervorgehoben dargestellt werden. Die Einteilung in verschiedene Regionen wird vom generierten Plugin automatisch aufgrund der im Optionsblock einer Grammatik konfigurierbaren Kommentarelemente (siehe [GKR$^+$06]) vorgenommen. Die Schlüsselwörter einer Sprache werden in einer Kommaseparierten Liste angegeben (Abbildung 9.3, Zeilen 3 und 4). Im generierten Editor werden Kommentare in Grün, Code in Schwarz und Schlüsselwörter in Magenta markiert.

Folding.

Zusammenklappbare Coderegionen können anhand der Struktur der Grammatik identifiziert werden. Hierzu definiert der Sprachentwickler in einer Komma-separierten Liste die Nichtterminale, deren entsprechende Code-Regionen zusammenklappbar sein sollen (Abbildung 9.3, Zeilen 6 und 7). Im generierten Editor wird anschließend nach Vorkommen dieser Nichtterminale gesucht. Neben den entsprechenden Codeabschnitten werden kleine Icons angezeigt, deren Selektion den Code zusammen- bzw. auseinanderklappt. Zusammengeklappte Regionen werden dann durch die jeweils erste Zeile gefolgt von zwei Punkten dargestellt.

Outline.

Die Vorgehensweise für die Erstellung von Elementen (sog. *Segmente*) für die Outline orientiert sich ebenfalls an der Struktur der Grammatik bzw. eines konkreten ASTs. Zunächst gibt der Sprachentwickler an, für welche Nichtterminale Einträge in die Outline erfolgen sollen (Abbildung 9.3, Zeilen 9 bis 12). Anschließend wird definiert, welches Icon für den Eintrag verwendet und welcher Text angezeigt werden soll. Der anzuzeigende Text kann dabei aus festen String-Elementen, Attributabfragen und Methodenaufrufen zusammengesetzt sein. Attributabfragen und Methodenaufrufe werden dabei auf dem jeweiligen Knoten im AST ausgeführt. Die hierarchische Struktur der generierten Outline ergibt sich aus der Hierarchie des AST: Outline-Elemente für übergeordnete AST-Konten sind hierarchisch über Outline-Elementen untergeordneter AST-Knoten platziert.

Fehlermeldungen.

Fehlermeldungen in der Problems-View können zwei Ursachen haben: zum einen kann der Parser aufgrund falscher Eingaben Fehlermeldungen produzieren, zum anderen können nachgelagerte Kontextbedingungsprüfungen Eingaben als fehlerhaft identifizieren. Da sowohl der Parser als auch die Kontextbedingungsprüfungen die von MontiCore vorgegebene Schnittstelle für Fehlermeldungen nutzen, wurde diese Schnittstelle als Ausgangsbasis für die Anbindung der

Problems-View genutzt. Die MontiCore-Fehlermeldungen werden intern in das Eclipse-Format übersetzt und an die entsprechende Schnittstelle in Eclipse weitergeleitet. Der Sprachentwickler muss lediglich angeben, welche Workflows beim Einlesen eines Modells ausgeführt werden sollen (z.B. Parsen, Kontextbedingungsprüfungen), die Workflows selbst können unverändert wiederverwendet werden. Die explizite Angabe der zu verwendenden Workflows ist dabei nötig, da es durchaus Situationen gibt, in denen der Sprachentwickler lediglich eine Teilmenge aller registrierten Workflows beim Einlesen durch den Editor ausführen möchte. Aufgrund der Abhängigkeit der gewählten Workflows von der Sprachkombination erfolgt die Angabe in der Sprachdatei.

Menüpunkte im Editor.

Menüpunkte im Kontextmenü des generierten Editors können Funktionalitäten abhängig vom Editorinhalt implementieren. Beispiele hierfür sind Formatierung des Inhalts, Codegenerierung oder Refactorings. Zur Implementierung einer solchen Funktionalität stellt die zur Verfügung gestellte Bibliothek das Interface `IEditorAction` zur Verfügung. Dieses Interface beinhaltet dabei eine einzige Methode `run`, die als Parameter den aktuellen Editor, die Position des Cursors im Editor und den eventuell markierten Teilabschnitt des Editorinhaltes besitzt. Über den Editor können weitere Informationen wie Dateiname, Dateiort oder der AST abgefragt werden, Cursorposition und markierte Teilabschnitt dienen der Implementierung von positions- oder markierungsabhängigen Funktionen wie Refactorings. Seitens des Sprachentwicklers muss lediglich das vorgegebene Interface implementiert werden, zusätzlich ist die implementierende Klasse unter Angabe eines Namens für den Menüpunkt in der Sprachdatei anzugeben (Abbildung 9.4, Zeilen 11 und 12). Da es zusätzlich möglich ist, in der Sprachdatei einen Pretty-Printer anzugeben, wird hierdurch automatisch ein Menüpunkt zur Formatierung des Modells unter Verwendung des Pretty-Printers generiert.

Menüpunkte im Navigator.

Im Gegensatz zu den Kontextmenüpunkten des Editors können die Menüpunkte des Navigators von mehreren Dateien abhängen. Auch hier implementiert der Sprachentwickler ein vorgegebenes Interface `INavigatorAction`, wobei dessen `run`-Methode als Parameter die selektierten Dateien besitzt. Die implementierende Klasse wird ebenfalls in der Sprachdatei unter Angabe des Namens für den Menüpunkt spezifiziert (Abbildung 9.4, Zeilen 14 und 15).

9.2.2. Struktur eines Plugins

Neben der Bereitstellung von Komfortfunktionalitäten sind die generierten Werkzeuge auf die Kompositionalität von Sprachen ausgelegt. Eclipse selbst bietet hierfür keine Unterstützung an, weswegen aufbauend auf den gegebenen Bibliotheken eine zusätzliche Infrastruktur entwickelt wurde, die speziell zur Umsetzung der Kompositionalitätsanforderungen dient.

Als Ausgangsbasis wurde hierfür zunächst eine Bibliothek entwickelt, die auf sprachunabhängiger Basis Eclipse und MontiCore verbindet. Diese Bibliothek enthält einerseits Standardfunktionalitäten, die automatisch in jedem Editor zur Verfügung stehen. Andererseits sind hierin

abstrakte Basisklassen definiert, die mit den Angaben für eine konkrete Sprache parametrisiert werden können. Die konkrete Parametrisierung entsteht dabei anhand der aus der Editorspezifikation gewonnenen Informationen. Hierbei wird ausgehend vom Spezifikationsanteil der Grammatik zunächst ein `FragmentHandler` generiert. Abbildung 9.5 zeigt dessen Struktur.

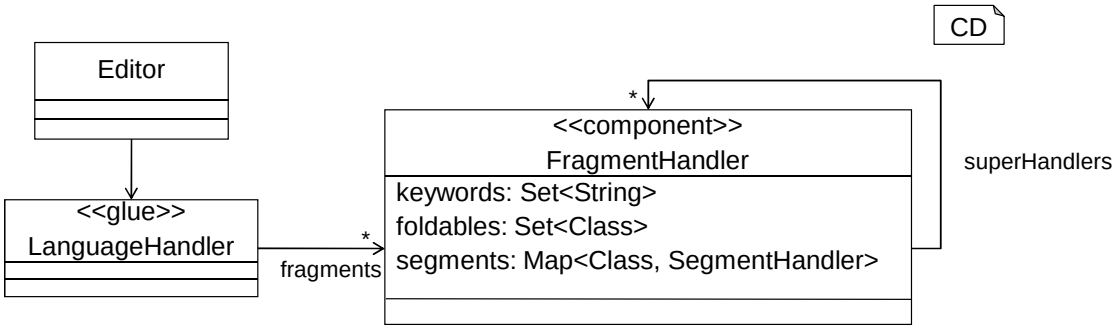


Abbildung 9.5.: Aufbau eines Plugins

Innerhalb des `FragmentHandlers` sind alle Informationen gekapselt, die für eine einzelne Grammatik relevant sind. Hierzu gehören die Schlüsselwörter, die zusammenklappbaren Regionen und die Anteile der Outline. Die Schlüsselwörter werden dabei als einfacher String hinterlegt, die Regionen und Segmente der Outline werden anhand der AST-Klasse des entsprechenden Nichtterminals gespeichert, um eine spätere Identifikation im AST zu ermöglichen. Für die Segmente werden zusätzliche Anteile generiert (`SegmentHandler`), die den AST-Knoten in ein Segment mit Text und Icon umwandeln.

Weiterhin ist die hierarchische Anordnung der `FragmentHandlers` anhand der Assoziation `superHandlers` zu erkennen. Diese Hierarchie wird zur Umsetzung der Grammatikvererbung in MontiCore verwendet: für jede Grammatik wird ein eigener `FragmentHandler` generiert. Falls die Grammatik eine oder mehrere Supergrammatiken besitzt, wird zusätzlich ein Verweis auf die `FragmentHandler` dieser Supergrammatiken erstellt. Hierdurch wird die unabhängige Erstellung der Plugins erreicht und die Änderungen an den Editoren von übergeordneten Grammatiken automatisch in allen Editoren von Subgrammatiken reflektiert. Zusätzlich wird durch dieses Vorgehen eine wesentliche Eigenschaft der Vererbung in der Editorspezifikation umgesetzt: auch hierbei wird in Subgrammatiken lediglich das Delta zur Obergrammatik spezifiziert, das Gesamtverhalten ergibt sich immer aus der Summe aller an der Grammatikhierarchie beteiligten Spezifikationen.

Neben der Umsetzung der Grammatikvererbung wird in Abbildung 9.5 zusätzlich aufgezeigt, wie mit Spracheinbettung verfahren wird. Hierbei wird ein `LanguageHandler` aus der Sprachdatei generiert, der alle durch Einbettung eingebundenen Editoranteile referenziert. Durch diese Vorgehensweise wird eine separate Generierung und Kompilierung der Anteile erreicht, insbesondere werden Änderungen an einzelnen Sprachen bzw. Editoren unmittelbar in allen Werkzeugen, die auf der entsprechenden Sprache aufbauen, reflektiert.

Die aus den Sprachdateien gewonnenen Informationen für Kontextmenü- und Navigatormenü-Einträge, für die Dateiendungen der Modelle oder das für das zu startenden Tool und dessen Workflows zum Einlesen von Modellen werden an verschiedenen Punkten des generierten Werk-

zeugs eingebunden. Der Ort der Einbindung bestimmt sich dabei aus den von Eclipse vorgegeben Strukturen: Menüpunkte im Editor werden in der Editorklasse angegeben, Menüpunkte für den Navigator hingegen in der Konfigurationsdatei. Die Dateiendung wird ebenfalls in der Konfigurationsdatei verwendet, das Tool und die Workflows finden wiederum im generierten Editor Verwendung.

Zusätzlich zu den dargestellten Klassen werden viele vor allem Eclipse-spezifische Komponenten generiert. Hierzu gehören die Konfigurationsdatei, die Manifestdatei und weitere Klassen, die für die Funktion des Editors nötig sind. All diese Komponenten sind jedoch für den Sprachentwickler nicht von Interesse, da die Angaben in der Grammatik bzw. den Sprachdateien alle vom Entwickler anzugebenden Informationen enthalten. Hierdurch müssen sich Sprachentwickler zur Erstellung eines Plugins nicht in Eclipse und die von MontiCore vorgegebenen Komponenten zur Eclipse-Anbindung einarbeiten, sondern lediglich die einfach gehaltene Spezifikationssprache für Editoren verwenden.

Zur Auslieferung und Integration der Editoren in Eclipse sind die generierten Dateien inklusive der Anteile für konkrete und abstrakte Syntax, der Symboltabellen und Kontextbedingungen und weiterer Sprachkomponenten wie Generatoren als `jar`-File zu verpacken und in den `plugin`-Ordner von Eclipse zu kopieren. Im Anschluss besitzt der Sprachnutzer ein voll funktionsfähiges Eclipse-Plugin für seine Sprache. Dieses Plugin kann neben weiteren Plugins für andere Sprachen installiert werden, wodurch eine Basis für die Unterstützung der Sprachvereinigung gegeben ist.

9.2.3. Umsetzung der Sprachaggregation und Einordnung in die Sprachhierarchie

Im vorangegangen Abschnitt wurde die Umsetzung der Kompositionsmechanismen Sprachvererbung und Spracheinbettung bezüglich der Werkzeuggenerierung vorgestellt. Im Gegensatz zu diesen Mechanismen wird die Sprachaggregation nicht innerhalb eines Plugins realisiert, sondern über die Verknüpfung der Sprachen. Abbildung 9.6 zeigt die prinzipielle Vorgehensweise.

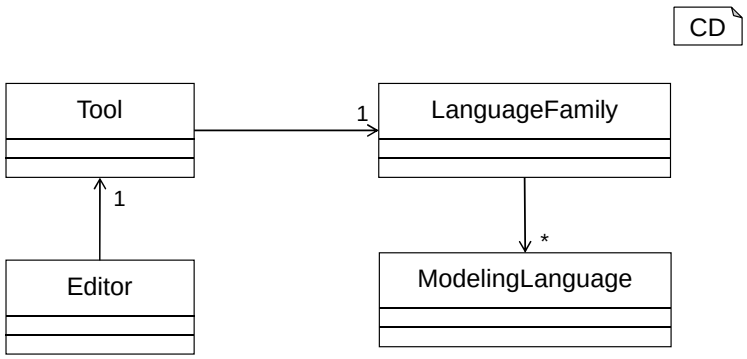


Abbildung 9.6.: Umsetzung der Sprachaggregation

Ausgehend für die Umsetzung der Sprachaggregation ist die Existenz einer Sprachfamilie und eines `Tools`, welches mit dieser Sprachfamilie parametrisiert ist (siehe Abschnitt 4.6.1).

Die Bildung dieser Sprachfamilie und des `Tools` richtet sich dabei nach den in den vorangegangenen Kapiteln beschriebenen Vorgehensweisen und ändert sich für die Einbindung in die Editor-Strukturen nicht.

Durch die Spezifikation des Editors, im Speziellen aufgrund der Angabe des `Tools` in der Sprachdatei gemäß Abbildung 9.4, wird der generierte Editor mit dem `Tool` für die gesamte Sprachfamilie parametrisiert. Dementsprechend wäre es möglich, auch Modelle anderer Sprachen einzulesen. Aufgrund der Angabe der Dateiendung wird der Editor jedoch fest mit der gegebenen Sprache verbunden, wodurch dieser auch nur für die gewünschte Modellart geöffnet wird. Nichtsdestotrotz ist es intern möglich, Modelle anderer Sprachen einzulesen. Hierdurch können beispielsweise Modellschnittstellen erstellt und eingelesen und somit sprachübergreifende Kontextbedingungen umgesetzt werden. Intern ist daher die Sprachaggregation weiterhin umgesetzt, die Einbindung in die Editor-Strukturen hat keine Auswirkungen auf die anderen Aspekte (Syntaxen, Symboltabellen, Kontextbedingungen etc.) der Sprachentwicklung. Die entsprechenden Komponenten können unverändert übernommen werden.

Bei der Erstellung einer Sprachfamilie ist es natürlich wünschenswert, Werkzeugunterstützung für alle an der Familie beteiligten Sprachen zu erhalten. Die Umsetzung erfolgt hierbei in beschriebener Weise, bei der Kombination ist jedoch ein zusätzlicher Arbeitsschritt notwendig. Begründet ist dieser Arbeitsschritt in der von der Sprachkombination unabhängigen Entwicklung der Einzelsprachen bzw. Einzeleditoren: diese können aufgrund der Unabhängigkeit in ihrer Spezifikation kein `Tool` verwenden, welches alle Sprachen der Sprachfamilie kennt. Daher sind bei der Entscheidung, welche Sprachen an der Sprachfamilie beteiligt sind, die unterliegenden `Tools` der Editoren mit anzupassen. Da diese jedoch ihre Sprachfamilie per Fabrik erstellen, ist der Aufwand auf wenige Zeilen beschränkt.

Einordnung in die Sprachhierarchie

Alle in den vorigen Kapiteln besprochenen Aspekte der Sprachentwicklung wurden durch spezielle Registrierungsmechanismen in die Sprachhierarchie gemäß Abschnitt 4.6.1 eingeordnet. Die Editoren verfolgen jedoch einen anderen Ansatz: die Editorkomponenten werden an keiner Stelle der Sprachhierarchie angemeldet, sondern lediglich mit der Sprache aufgrund der Angabe des `Tools` verbunden. Dementsprechend existiert eine Abhängigkeit von den Editoren hin zu den anderen Komponenten, in die Rückrichtung existiert keine Abhängigkeit. Hierdurch bleiben die Grundkomponenten Syntaxen, Symboltabellen, Kontextbedingungen und die darauf aufbauenden Elemente wie Codegeneratoren unabhängig von Eclipse und können somit separat ausgeliefert und verwendet werden. Dadurch können diese Anteile auch in Verbindung mit anderen IDEs eingesetzt werden, zusätzlich wird die Verwendung auf Kommandozeilenbasis ohne Einbindung der Eclipse-Funktionalitäten ermöglicht.

Zur Sicherung der korrekten Abhängigkeiten empfiehlt es sich für Sprachentwickler, die Projektstruktur wie in Abbildung 9.7 anzuordnen.

Grundlegend hierbei sind die von MontiCore vorgegebenen Bibliotheken für die Sprachentwicklung bzw. die Editorentwicklung. Beide sind separat gehalten, wobei auch hier eine Abhängigkeit vom Editorframework zum Framework für Basiskomponenten (Syntaxen, Symboltabellen, Kontextbedingungen etc.) besteht. Ebenso sollten die vom Nutzer erstellten Projekte angeordnet sein, wobei sich die jeweiligen Abhängigkeiten aufgrund der Funktion des Projektes

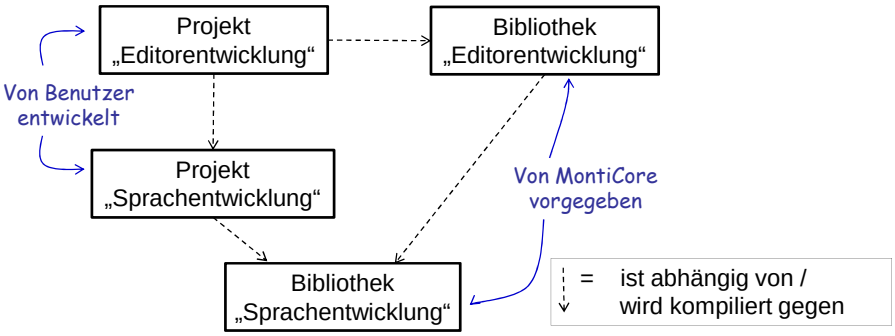


Abbildung 9.7.: Vorgeschlagene Projektstruktur bei der Editorerstellung

ergeben. Durch diese Anordnung wird sichergestellt, dass die Basiskomponenten keine Abhängigkeiten zu den Editoren und somit zu Eclipse haben. Die transitive Verbindung vom Editorprojekt zum Framework für Basiskomponenten bleibt dabei aufgrund der inneren Arbeitsweise des Classpaths von Java ohne negative Auswirkungen.

9.3. Umsetzung der Szenarien

9.3.1. Sprachvereinigung

Bei der Sprachvereinigung sind zwei wesentliche Aspekte relevant: zum einen müssen die vorhandenen Editoren gleichzeitig in Eclipse installierbar sein. Zum anderen müssen die den Editoren unterliegenden Sprachbestandteile, insbesondere die Symboltabellen und Kontextbedingungsprüfungen miteinander arbeiten können.

Der erste Punkt wird dabei bereits von Eclipse übernommen. Hier ist es möglich, beliebig viele Plugins für verschiedene Sprachen zu installieren. Die Editoren der Sprache werden aufgrund der Dateiendung identifiziert, wodurch stets der richtige Editor für jedes Modell genutzt wird. Die Editoren der verschiedenen Sprachen beeinflussen sich somit gegenseitig nur, wenn sie für die gleiche Dateiendung entwickelt wurden. In solchem Fall muss die Dateiendung in der Sprachdatei angepasst werden.

Der zweite Punkt - die Verbindung der unterliegenden Sprachbestandteile - wird wie in Abschnitt 9.2.3 behandelt. Hierbei werden in den existierenden `Tools` erweiterte Sprachfamilien registriert, wodurch diese anschließend intern auch Modelle anderer Sprachen verarbeiten können. Die Registrierung gemäß Tabelle 9.8 bleibt daher die einzige Aufgabe für den Sprachentwickler.

Komponente	Aufgabe	Registrierungsort
`LanguageFamily`	Unterstützung der gesamten Sprachfamilie.	`Tool`

Tabelle 9.8.: Zu entwickelnde Komponenten für die Sprachvereinigung

9.3.2. Nichtterminal-Erweiterung und Nichtterminal-Ersetzung

Sowohl bei der Nichtterminal-Erweiterung als auch bei der Nichtterminal-Ersetzung existieren prinzipiell zwei unterschiedliche Lösungsstrategien (siehe Abschnitte 5.4 und 5.5): erstens konnte eine gemeinsame Grammatik durch Verwendung der Mehrfachvererbung erstellt werden, zweitens bestand eine Lösungsstrategie in der Erstellung einer Subgrammatik einer Sprache mit anschließender Einbettung der zweiten Sprache. Bezüglich der Editoren ergeben sich für diese Fälle unterschiedliche Konsequenzen.

Verwendung der Mehrfachvererbung

Die Mehrfachvererbung auf Grammatikebene reflektiert sich auch auf Editorebene: die Eigenschaften des Editors der erbenden Grammatik ergeben sich somit aus den Eigenschaften der Editoren der Supergrammatiken. Der hierfür verwendete Mechanismus wurde bereits in Abschnitt 9.2.2 vorgestellt. Hierbei wurde für die Subgrammatik ein neuer `FragmentHandler` generiert, der mit den `FragmentHandlern` der Supergrammatiken verbunden ist und somit deren Funktionalitäten ohne erneute Generierung/Kompilierung wiederverwendet.

Bezüglich der Spezifikationen innerhalb der Sprachdateien wird im Falle der Mehrfachvererbung eine neue Sprachdatei erstellt, aus denen die Komponenten generiert werden. Diese Vorgehensweise wird deshalb verwendet, da die Angaben in der Sprachdatei sich typischerweise für jede Sprache ändern, insbesondere auch für eine neue Sprache, die aus existierenden Sprachen per Mehrfachvererbung entstanden ist. Nichtsdestotrotz können hier existierende Artefakte (z.B. Einträge im Kontextmenü des Editors oder im Navigator) erneut angegeben und somit wiederverwendet werden.

Verwendung der Einbettung

Bei der Umsetzung durch Einbettung werden die Sprachen durch die Erstellung einer neuen Sprachdatei vereint. Wie bereits bei der Mehrfachvererbung sind auch hier die Editoreigenschaften zu spezifizieren, da diese sich von denen der existierenden Sprachen unterscheiden.

Die Angaben in den Grammatiken werden auch hier wiederverwendet. Gemäß Abschnitt 9.2.2 wird hierfür ein neuer `LanguageHandler` gebildet, der die einzelnen `FragmentHandler` der Sprachen einbindet. Hierdurch kann auf eine erneute Spezifikation verzichtet werden, eine erneute Generierung bzw. Kompilierung findet nicht statt.

Sowohl bei der Verwendung der Mehrfachvererbung als auch bei Verwendung der Einbettung ist vom Sprachentwickler lediglich eine neue Sprachdatei zu schreiben. Tabelle 9.9 fasst diese Aufgabe zusammen.

9.3.3. Einbettung einer einzelnen Sprache und Einbettung von Sprachalternativen

Bei beiden Szenarien werden die gleichen Mechanismen angewandt wie bei der Umsetzung der Nichtterminal-Erweiterung und der Nichtterminal-Ersetzung durch Einbettung. Hier wird die konkrete Zusammensetzung in der Sprachdatei definiert, wobei neue Angaben bezüglich der dortigen Editorspezifikationsanteile erfolgen.

Komponente	Aufgabe	Registrierungsort
Sprachdatei	Spezifikation der relevanten Editoranteile unter Wiederverwendung der Klassen für Menüeinträge.	Keine explizite Registrierung.

Tabelle 9.9.: Zu entwickelnde Komponenten für Nichtterminal-Erweiterung und -Ersetzung

Bei der Einbettung von Sprachalternativen kann es zu keinen ungewünschten Ergebnissen bezüglich der Outlines und der Foldings kommen. Hauptgrund hierfür ist, dass sowohl die Elemente der Outline als auch die zusammenklappbaren Code-Regionen anhand eines Visitors auf AST-Basis aufgebaut werden. Daher werden diese Elemente stets korrekt für die verwendete eingebettete Sprache ermittelt.

Dennoch kann es vor allem bei der Einbettung von Sprachalternativen vorkommen, dass beispielsweise Menüpunkte des Editors und Menüpunkte im Navigator nur für eine bestimmte Sprachkombination gültig sind. In diesen Fällen sollte eine Subklasse der Aktionsausführungsklassen erstellt werden, wobei diese zunächst die konkrete Sprachkombination anhand eines Visitors prüft. Falls dabei eine unerwünschte Sprachkombination ermittelt wurde, kann auf die Ausführung der Aktion verzichtet werden, andernfalls wird die entsprechende Methode der Superklasse aufgerufen und somit wiederverwendet. Im Falle des Verzichts sollte zusätzlich eine entsprechende Meldung an den Benutzer erfolgen. Insgesamt ergeben sich die in Tabelle 9.10 zusammengefassten Aufgaben für den Sprachentwickler.

Komponente	Aufgabe	Registrierungsort
Sprachdatei	Spezifikation der relevanten Editoranteile unter Wiederverwendung der Klassen für Menüeinträge.	Keine explizite Registrierung.
Klassen der Aktionen für Menüeinträge	Analyse der Sprachzusammensetzung vor Ausführung der wiederverwendeten Aktion. Rückmeldung an den Benutzer, falls Aktion nicht ausgeführt wird.	Sprachdatei.

Tabelle 9.10.: Zu entwickelnde Komponenten für die Einbettung einer einzelnen Sprache und die Einbettung von Sprachalternativen

9.3.4. Entwicklung einer erweiterbaren Sprache

Bei der Entwicklung einer erweiterbaren Sprachen kommen bezüglich der Werkzeugentwicklung die Vorteile des generativen Ansatzes zum Tragen. Der Sprachentwickler spezifiziert das Werkzeug in einer vorgegebenen Sprache, woraus MontiCore die entsprechenden Anteile automatisch generiert. Durch dieses Vorgehen bleibt der Spielraum für den Sprachentwickler be-

schränkt, insbesondere kann dieser kaum Fehler bezüglich Kompositionalität oder Erweiterbarkeit seiner Sprache und des begleitenden Werkzeuges machen. Daher ist in diesem Kapitel nur eine Richtlinie identifizierbar, wobei auch diese eine Ergänzung der Richtlinien vorangegangener Kapitel darstellt.

Richtlinie 16: Verwendung der empfohlenen Projektstruktur

Die in Abbildung 9.7 dargestellte Projektstruktur dient einerseits der Verwendbarkeit des Sprachwerkzeugs auf Kommandozeilenbasis, andererseits ermöglicht sie die Einbindung in andere IDEs. Der Aufwand der Trennung ist dabei minimal: wird die Sprache in Eclipse entwickelt, müssen sich die entsprechenden Projekte lediglich geeignet referenzieren. Werden die Projekte durch build-Scripte wie *ant* oder *make* erstellt, ist der Java-Classpath entsprechend zu gestalten.

9.4. Verwandte Arbeiten

Bereits in den achtziger und neunziger Jahren wurde intensiv an der Generierung sprachspezifischer Werkzeuge geforscht. Bekannte Ansätze sind unter anderem PSG [BS86], Mentor [DHKL84] oder IPSEN [Nag96, KS97]. Diese Werkzeuge sind jedoch nicht in heutige IDEs integriert und behandeln die kompositionale Entwicklung von Sprachen nicht.

Das zur Zeit noch in der Startphase befindliche Eclipse-IMP Project [IMP] zielt unter anderem auf die Unterstützung bei der Entwicklung von sprachspezifischen IDEs ab. Geplante Funktionen sind die Erstellung von syntaxgetriebenen Editoren mit verschiedenen Komfortfunktionalitäten, die Erstellung von Parsern mittels Parsergeneratoren oder auch verschiedene Refactoring-Mechanismen. Die kompositionale Entwicklung von Sprachen bzw. Werkzeugen ist derzeit nicht geplant.

Auf IMP aufbauend wurde in [KKV08] und [KKV09] ein Werkzeug diskutiert, welches das SDF [HHKR89] als Generalized-Scannerless-LR-Parser-Generator verwendet. Die bisher erreichten Merkmale auf dem Gebiet der Werkzeuggenerierung sind mit denen des in dieser Arbeit beschriebenen Ansatzes vergleichbar, die Kompositionalität des Parse-Algorithmus wurde in [KKV08] und [KKV09] jedoch nicht konsequent auf die Werkzeuggenerierung übertragen.

Auch xText als Teil von openArchitectureWare [OAW] bietet Funktionalitäten zur Werkzeuggenerierung in Form von Eclipse-Plugins. Das zur Zeit in der Erprobungsphase befindliche Konzept der Sprachmodularisierung in Form von Vererbung ist bisher nur für die konkrete und abstrakte Syntax verfügbar. Die Werkzeuggenerierung ist nicht darauf abgestimmt.

Ein auf EMF basierender Ansatz zur Erstellung von Editoren für DSLs wird in [GS08] beschrieben. Hierbei wird zu einer Grammatik eine weitere Datei mit Informationen zum Syntaxhighlighting, Folding oder zum Aufbau einer Outline erstellt. Aus dieser Datei werden verschiedene Eclipse-spezifische Klassen generiert, zusätzlich muss der Entwickler einige Anteile des Editors selbst implementieren, eine vollständige Generierung wird nicht unterstützt. Sprachkompositionalität wird in [GS08] nicht behandelt.

9.5. Zusammenfassung

In diesem Kapitel wurde der Ansatz zur kompositionalen Entwicklung von Werkzeugen vorgestellt. Hierbei definiert der Sprachentwickler in der Grammatik bzw. in der Sprachdatei die gewünschten Eigenschaften des Editors. Diese reichen von Syntaxhighlighting und einer Outline zur hierarchischen Darstellung der Eingabe über zusammenklappbare Coderegionen bis hin zu Menüpunkten im Kontextmenü bzw. Navigator. Aus dieser Spezifikation werden anschließend vom MontiCore-Framework Eclipse-Plugins generiert, die durch einfaches Verpacken in einer `jar`-Datei mit anschließendem Einfügen in den `plugins`-Ordner von Eclipse installiert werden können. Der Sprachentwickler muss sowohl während der Definition der Werkzeuge als auch bei der Auslieferung keine tiefergehenden Kenntnisse von Eclipse und dessen Erweiterungsmechanismen besitzen. Zusätzlich beschränkt sich der Aufwand bei der Erstellung von Editoren auf wenige Zeilen Spezifikation, wodurch sich die Investition auch bei Sprachen mit geringer Nutzerzahl rentiert.

Neben den Komfortfunktionalitäten bieten die generierten Plugins eine Unterstützung der kompositionalen Sprachentwicklung. Hierbei werden alle drei Mechanismen - Sprachvererbung, Einbettung und Aggregation - unterstützt, wobei für jede Sprache separat Plugin-Anteile generiert und in einem weiteren Schritt kombiniert und somit wiederverwendet werden. Der bei der Kombination entstehende Entwicklungsaufwand für den Sprachentwickler beschränkt sich dabei auf eine eventuelle Neuerstellung der Sprachdatei. Durch die Unterstützung der Kompositionsmechanismen konnten anhand des vorgestellten Ansatzes alle Szenarien der Sprachkomposition umgesetzt werden.

Kapitel 10.

Fallstudie

Zur Überprüfung der Anwendbarkeit des Ansatzes wurde eine Fallstudie durchgeführt, die verschiedene Szenarien der kompositionalen Sprachentwicklung beinhaltet. Die verwendeten Einzelsprachen wurden dabei nicht allein vom Autor der vorliegenden Arbeit verfasst, vielmehr sind diese in verschiedenen anderen Arbeiten entwickelt worden (siehe Abschnitte 10.1.1 bis 10.1.3). Die Einbindung von weiteren Entwicklern diente dabei der Überprüfung der Verständlichkeit und Anwendbarkeit des entwickelten Systems zur kompositionalen Sprachentwicklung. Weiterhin konnte durch die Verwendung existierender Sprachen sichergestellt werden, dass diese ohne Kenntnis der letztendlichen Sprachkombinationen entwickelt worden sind.

Insgesamt verfügt die Fallstudie über eine recht komplexe Interaktion von verschiedenen Teilsprachen. Diese Interaktionen beziehen sich dabei auf alle Artefakte der Sprachentwicklung (Syntaxen, Symboltabellen, Kontextbedingungen und Editoren). Aufgrund des Umfangs der Studie werden in diesem Kapitel nur die wichtigsten Elemente der beteiligten Sprachen und der Sprachkombinationen diskutiert. Insgesamt beinhaltet dieses Kapitel:

- Eine Vorstellung der beteiligten Einzelsprachen.
- Eine Beschreibung der daraus entstandenen Sprachkombinationen.
- Die wesentlichen Elemente der Realisierung der Fallstudie.
- Eine Diskussion der wichtigsten Ergebnisse.

10.1. Beteiligte Sprachen

10.1.1. Statecharts

Für die Fallstudie wurden Statecharts als Teil der UML/P [Rum04b, Rum04a] eingesetzt, wobei die Umsetzung der Sprache auf MontiCore-Basis in der Dissertation [Sch11] erfolgte. Da in dieser Dissertation die gesamte Sprachfamilie UML/P implementiert wurde und die verschiedenen Sprachen viele Gemeinsamkeiten aufweisen, wurde eine spezielle Spracharchitektur verwendet. Abbildung 10.1 zeigt diese Architektur.

Wie in dieser Abbildung ersichtlich ist, bauen die Statecharts auf einer Hierarchie von Basisgrammatiken auf. Die Grammatik *Literals* definiert hierbei Literale wie Character und String sowie numerische und boolesche Literale. Die Grammatik *Types* definiert die in den Sprachen der UML/P verwendbaren Ausdrücke für Typen. Diese reichen von grundlegenden Datentypen

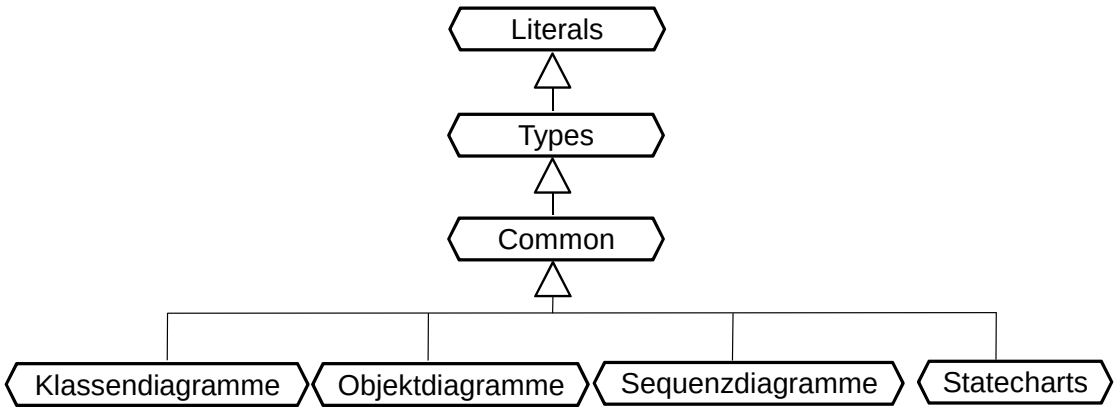


Abbildung 10.1.: Aufbau der Sprachen der UML/P

wie `int` oder `boolean` über Arraytypen unter Verwendung der eckigen Klammern bis hin zu generischen Typen wie etwa `List<String>`. Die Grammatik *Common* stellt zusätzliche Basiselemente wie Modifikatoren, Stereotypen und Kardinalitätsangaben zur Verfügung. Abgeleitet von diesen Basisgrammatiken sind die einzelnen Teilgrammatiken der UML/P definiert.

Die wesentlichen Kernelemente der verwendeten Statecharts sind hierarchische Zustände und Transitionen. Sowohl Zustände als auch Transitionen können mit Aktionen und Invarianten versehen werden. Hierzu verfügt die Grammatik über zwei externe Nichtterminale `Statement` und `Invariant`[1], wobei ersteres für Aktionen verwendet wird, zweiteres findet bei Invarianten der Zustände bzw. Transitionen Anwendung. Zusätzlich zu diesen Elementen bieten die Statecharts weitere Elemente, die im Kontext der Fallstudie jedoch keine Anwendung finden. Anhang E zeigt die komplette Grammatik auf.

10.1.2. OCL

Die OCL ist ebenso wie die Statecharts Teil der UML/P. Die in der MontiCore-Sprachbibliothek zur Verfügung stehende Version wurde in [PS07] erstellt und in [Hel10] und [Sch11] weiterentwickelt. Da auch die OCL Teil der UML/P ist, baut auch diese Sprache auf den grundlegenden Grammatiken *Literals*, *Types* und *Common* aus Abbildung 10.1 auf.

Die verwendete Variante der OCL hat einen sehr großen Sprachumfang, daher werden an dieser Stelle nur die für die Fallstudie relevanten Teile aufgeführt. Hierbei handelt es sich zum einen um Invarianten, zum zweiten sind Vorbedingungen verwendet worden. Die Invarianten können beispielsweise eingesetzt werden, um Zustandsinvarianten in Statecharts zu definieren. Vorbedingungen können unter anderem in der Definition von Methoden in Klassendiagrammen verwendet werden. Für eine detaillierte Diskussion der Einsatzgebiete der OCL-Bestandteile wird auf [Rum04b, Rum04a] verwiesen. Die Gesamtgrammatik der OCL findet sich zur Referenz in Anhang F.

[1] Das externe Nichtterminal `Invariant` wird aus der Grammatik Common geerbt.

10.1.3. Java

Die objektorientierte Programmiersprache Java ist ebenfalls in der MontiCore-Sprachbibliothek verfügbar und baut auf der Grammatik der Typen aus Abbildung 10.1 auf. Bei der genutzten Variante handelt es sich um die ab Version 1.5 gültige Sprachversion gemäß [GJS05], deren MontiCore-Grammatik in [FM07] entwickelt wurde. Zusätzlich zur Grammatik stehen Symboltabellen, Kontextbedingungen und Editoren zur Verfügung, welche hauptsächlich vom Autor der vorliegenden Arbeit entwickelt wurden.

Das Einsatzgebiet der Sprache Java ist vielfältig. Einerseits kann sie wie gewohnt als einzige Sprache zur Programmierung verschiedenster Systeme verwendet werden. Andererseits ist es sinnvoll, auch Teile der Sprache innerhalb anderer Sprachen wiederzuverwenden. So können etwa Statements zur Spezifikation von Methodenrümpfen in Klassendiagrammen und Expressions zur Definition von Invarianten in verschiedenen Sprachen dienen. Insgesamt bietet die MontiCore-Sprachbibliothek ausgehend von der Java-Grammatik daher die Möglichkeit, auch Teile der Sprache zu verwenden. Die vollständige Grammatik ist in Anhang G aufgeführt.

10.2. Sprachkombinationen

Das in der Fallstudie entwickelte System besteht aus zwei Kernsprachen, die wiederum aus mehreren Teilsprachen unter Nutzung verschiedener Szenarien zusammengesetzt sind. In der ersten Sprache wurden dabei die Teilsprachen Statecharts, OCL-Expressions, Java-Expressions und Java-Statements kombiniert, die zweite Sprache umfasst Java im Verbund mit OCL-Vorbedingungen. Zur Unterscheidung der Sprachkombinationen von den Teilsprachen wird die erste Sprache mit SC/JO und die zweite Sprache mit Java/O gekennzeichnet. Im Folgenden werden diese Sprachen und deren Kombinationsarten näher vorgestellt.

10.2.1. SC/JO

Die Sprache SC/JO verwendet als Ausgangsbasis die Statecharts gemäß Abschnitt 10.1.1. Die beiden externen Nichtterminale für Aktionen und Invarianten wurden dabei durch Spracheinbettung wie folgt besetzt:

1. Aktionen werden durch Einbettung von Java-Statements ersetzt.

2. Invarianten können wahlweise durch OCL-Expressions oder durch Java-Expressions ersetzt werden.

Abbildung 10.2 zeigt ein exemplarisches Statechart unter Verwendung der beschriebenen Sprachkombination.

In dieser Abbildung sind dabei die spezifischen Kombinationen der beteiligten Sprachen zu erkennen. Bei den Invarianten der Zustände wurde in Zeile 18 eine OCL-Expression verwendet, alternativ ist auch die Verwendung von Java-Expressions gemäß Zeile 20 möglich. Die Auswahl der Sprache erfolgt dabei vom Nutzer: wird das Präfix `ocl:` verwendet, wird auf OCL-

SC/JO

```
1  package sc;
2
3  import jv.User;
4  import jv.Password;
5  import jv.LoginEvent;
6  import jv.Login;
7
8  //das Statechart Login ist der
9  //gleichnamigen Klasse aus dem Paket jv zugeordnet
10 statechart Login jv.Login{
11
12     initial state NotLoggedIn;
13
14     state CheckPasswd;
15
16     state LoggedIn{
17        //Verwendung der OCL-Expressions als Invariante
18        ocl: [user.isLoggedIn];
19        //alternativ wäre auch Java möglich
20        //java: [user.isLoggedIn()];
21
22        //Verwendung von Java-Statements als Aktionssprachen
23        entry / {System.out.println("Login successful!");}
24     }
25
26    NotLoggedIn -> CheckPasswd: receive(LoginEvent evt);
27
28     //Definition weiterer Zustände und Transitionen
29     //...
30 }
```

Abbildung 10.2.: Beispiel für eine Statechart in der Sprache SC/JO

Expressions umgeleitet, das Präfix `java:` selektiert Java-Expressions [2]. Zusätzlich zu den Invarianten sind in Zeile 23 Java-Blockstatements als Aktionssprache beim Betreten eines Zustands verwendet worden.

Insgesamt beschränkt sich die Sprachkombination nicht auf den rein syntaktischen Anteil. So wurde bei der Definition der OCL-Invariante in Zeile 18 auf eine Variable `user` zugegriffen. Diese Variable ist dabei in der dem Statechart unterliegenden Klasse `jv.Login` (siehe Zeilen 8-10) enthalten. Da diese Variable in Java definiert wurde und außerdem Java und OCL unabhängig voneinander entwickelt worden sind, muss sie explizit nach OCL konvertiert werden. Anschließend muss auch die Variable `isLoggedIn` in eine OCL-Variable umgewandelt werden, wodurch der Ausdruck in Zeile 18 aus OCL-Sicht korrekt wird.

[2] Eine Festlegung einer Default-Sprache gemäß Abschnitt 5.7 wäre ebenfalls möglich. Um das Beispiel übersichtlich zu lassen, wurde darauf verzichtet.

Anders verhält es sich bei der Verwendung von Java als Sprache für die Invariante. Hier muss keine explizite Konvertierung vorgenommen werden, da sowohl die Variablen-Definition als auch die Nutzung der Variable in Java erfolgt. Lediglich die Auflösungslogik muss dahingehend verändert werden, dass der dem Statechart unterliegende Typ als Suchbereich für die Auflösung von Elementen in Invarianten verwendet wird.

Neben diesen offensichtlichen Sprachkombinationen ist in der Abbildung eine weitere Sprache eingebunden. Hierbei sind zwei verschiedene Regionen im Modell von Interesse: die import-Anweisungen ab Zeile 3 und der Java-Ausdruck der Aktion in Zeile 23.

Die import-Anweisungen sind in der Sprache Statecharts definiert worden und dienen dort zum Import der Schnittstellen anderer Statecharts. Hierdurch können deren Zustände als Ursprung und Ziel von Transitionen verwenden werden. Innerhalb der Sprache SC/JO wurde die Bedeutung der import-Anweisungen jedoch erweitert. Neben dem Importieren von Statecharts können hierbei zusätzlich Java-Klassen importiert werden, welche anschließend im eingebetteten Java-Anteil verfügbar sind. Dementsprechend ist im gegebenen Beispiel zusätzlich die Sprache Java involviert.

Ähnliches gilt für die Aktion in Zuständen bzw. die Java-Invarianten: hier können Ausdrücke angegeben werden, die reine Java-Klassen verwenden (z.B. die Klasse `System` in Zeile 23). Hierbei werden demnach die eingebetteten Sprachen Java-Expressions und Java-Blockstatements mit der Sprache Java kombiniert.

Zusammenfassend wurden in der Sprache SC/JO folgende Szenarien verwendet[3]:

- **Entwicklung einer erweiterbaren Sprache.** Alle beteiligten Sprachen wurden unabhängig voneinander unter Verwendung der in dieser Arbeit entwickelten Infrastruktur und unter Beachtung der verschiedenen Richtlinien zur Entwicklung erweiterbarer Sprachen erstellt.

- **Einbettung einer einzelnen Sprache.** Dieses Szenario wurde bei der Einbettung von Java-Statements als Aktionssprache verwendet. Hierbei wurde das externe Nichtterminal `Statement` der Statechart-Grammatik durch das Nichtterminal `Blockstatement` der Java-Grammatik ersetzt.

- **Einbettung von Sprachalternativen.** Die Möglichkeit zur Auswahl der Sprache für die Invarianten verwendet die Einbettung von Sprachalternativen. Dabei wurde das externe Nichtterminal `Invariant` der Obergrammatik Common durch das Nichtterminal `Expression` von Java bzw. das Nichtterminal `OCLExpression` der OCL ersetzt.

- **Sprachvereinigung.** Der Import von Java-Klassen in die Sprache SC/JO entspricht dem Szenario der Sprachvereinigung. Aus Sicht der Grammatiken erfolgten hierbei keine Änderungen bzw. Ersetzungen.

Zusätzlich wurde die Sprache SC/JO so entwickelt, dass die Kombination nicht nur die konkrete und abstrakte Syntax beinhaltet. Vielmehr wurden auch die Symboltabellen kombiniert (z.B. beim Importieren von Java-Klassen in Statecharts) und die Kontextbedingungen wiederverwendet (z.B. bei der Prüfung der Korrektheit von Invarianten). Zusätzlich wurden auch die Editoranteile kombiniert. Abbildung 10.3 zeigt den resultierenden Editor.

[3] Für eine detaillierte Diskussion der Umsetzungen wird auf Abschnitt 10.3 verwiesen.

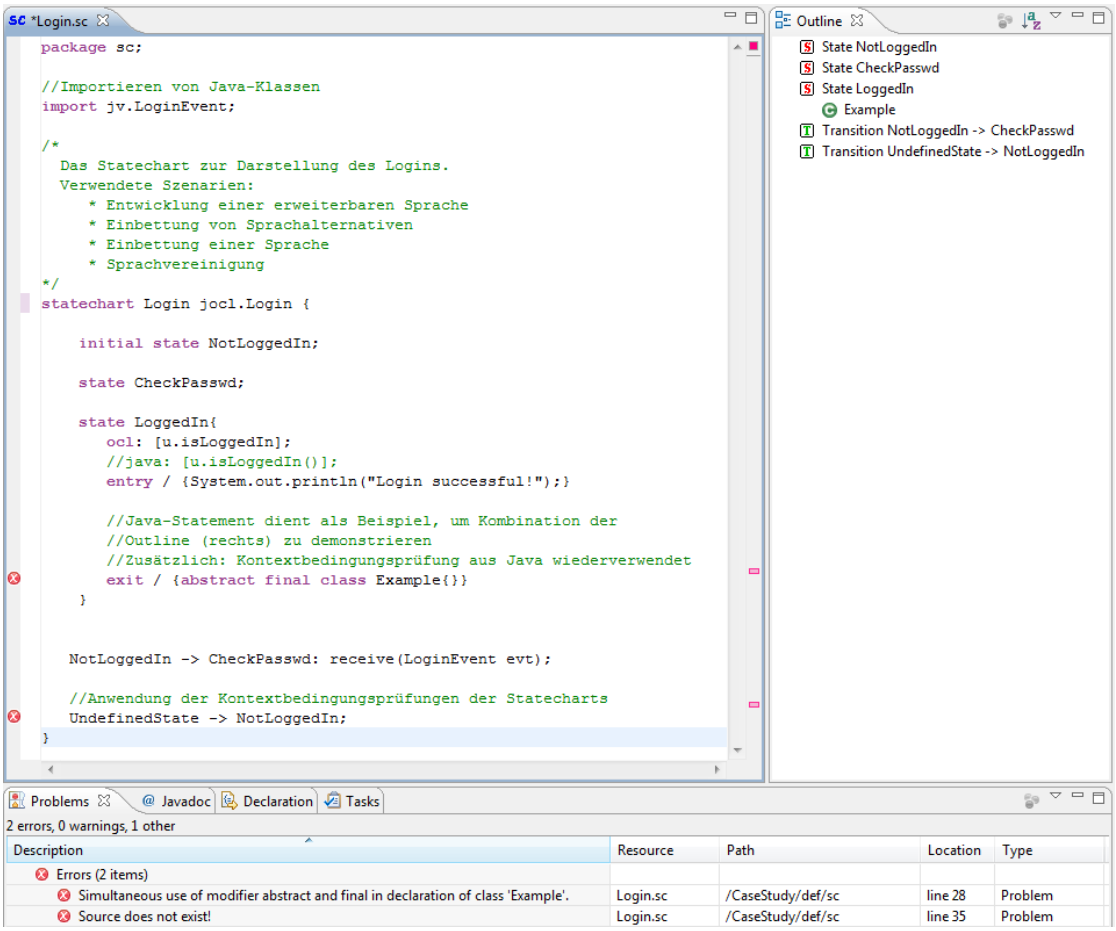


Abbildung 10.3.: Screenshot des Editors der Sprache SC/JO

10.2.2. Java/O

Als Ausgangsbasis für die Sprache Java/O wurde Java 1.5 gemäß Abschnitt 10.1.3 verwendet. Von syntaktischer Seite wurde Java dahingehend verändert, dass OCL-Vorbedingungen vor Methoden angegeben werden können. Abbildung 10.4 zeigt ein Beispiel.

In dieser Abbildung wird in Zeile 11 dargestellt, wie OCL-Vorbedingungen für Methoden eingesetzt werden. Neben der rein syntaktischen Kombination der Sprachen sind auch hier verschiedene weitergehende Sprachkombinationen aufgezeigt:

- Zeilen 3 bis 6: Die Bedeutung der Java-Importe wurde erweitert. So wird in Zeile 3 das Statechart aus Abbildung 10.2 importiert. Des Weiteren können sowohl reine Java-Klassen als auch Instanzen der neuen Sprache Java/O importiert werden.
- Zeilen 11 und 12: Der Methodenparameter `u` aus Java wird als OCL-Variable interpretiert. Zusätzlich wird der Java-Typ `User` nach OCL konvertiert, wodurch eine Abfrage der Variable `isLoggedIn` in OCL ermöglicht wird.

Java/O

```
1  package jv;
2
3  import sc.Login;
4  import jocl.LoginEvent;
5  import jocl.Password;
6  import jocl.User;
7
8  public class Weblogin{
9
10    //Verwendung der OCL-Vorbedingungen
11    pre: u.isLoggedIn==false;
12    public boolean loginIn(User u, Password password){
13
14       //Instanziierung des Statecharts
15       Login log = new Login();
16
17       //Senden eines Events
18       log.receive(new LoginEvent(u, password));
19
20       //Abfrage des Zustandes
21       boolean res=log.isIn_LoggedIn();
22       u.setLoggedIn(res);
23       return res;
24    }
25 }
```

Abbildung 10.4.: Beispiel für die Sprache Java/O

- Zeile 15: Hier wird das importierte Statechart der Sprache SC/JO instanziiert. Dieses Statechart wird dabei in Java als Klasse mit parameterlosem Konstruktor dargestellt.
- Zeile 18: Bei der Interpretation von Statecharts in Java wird implizit angenommen, dass jedes Statechart eine Methode `receive` besitzt. Diese Methode hat genau einen Parameter vom Typ `Event`, wobei dieser Typ in Java programmiert ist. Die Klasse `LoginEvent` ist ebenfalls in Java programmiert und Subklasse von `Event`. Hierdurch ist der Aufruf in Zeile 18 korrekt.
- Zeile 21: Um den Zustand eines Statecharts abfragen zu können, verfügt dieses aus Java-Sicht über boolesche Methoden, deren Namen sich aus dem Präfix `isIn_` gefolgt vom jeweiligen Zustandsnamen ergeben.

Auch bei der Sprache Java/O wurden demnach verschiedene Interaktionen über die reine syntaktische Kombination hinaus realisiert. Hierbei wurden die verschiedenen Symboltabelleneinträge ineinander übersetzt, Kontextbedingungen sprachübergreifend geprüft und ein Editor generiert. Insgesamt sind hierbei folgende Szenarien zur Anwendung gekommen[4]:

[4]Eine detailliertere Beschreibung der Umsetzungen befindet sich in Abschnitt 10.3.

- **Entwicklung einer erweiterbaren Sprache.** Auch hier wurden alle beteiligten Sprachen unabhängig voneinander unter Verwendung der vorgegebenen Infrastrukturen und der Richtlinien zur Entwicklung erweiterbarer Sprachen implementiert.
- **Nichtterminal-Ersetzung.** Um die Verwendung der OCL-Vorbedingungen zu ermöglichen, wurde das Nichtterminal `MethodDeclaration` durch ein Nichtterminal überschrieben, welches optionale OCL-Vorbedingungen besitzt. Alternativ hätte die Umsetzung auch durch Nichtterminal-Erweiterung erfolgen können.
- **Sprachvereinigung.** Bei der Interpretation der Statecharts in Java/O wurde das Szenario der Sprachvereinigung verwendet. Aus Sicht der Grammatiken waren hierzu keine Änderungen/Erweiterungen nötig.

10.3. Realisierung

Bei der Realisierung der Fallstudie wurden die beteiligten Sprachen bezüglich aller Aspekte der Sprachentwicklung miteinander verbunden. Hierzu wurde aufbauend auf den existierenden Elementen zusätzlicher Glue Code entwickelt, der zum Verbund der Teilsprachen dient. Im Folgenden werden sowohl die existierenden Einzelkomponenten als auch die wichtigsten Teile der Verbundkomponenten vorgestellt.

10.3.1. Komponentenanordnung

Als grundlegende Komponenten für die Fallstudie dienen zum einen die MontiCore-Bibliotheken, zum anderen die in dieser Arbeit entwickelten Komponenten zur kompositionalen Sprachentwicklung. Aufbauend hierauf sind die verschiedenen Einzelsprachen als Teilkomponenten eingebunden. Abbildung 10.5 zeigt den generellen Aufbau des Gesamtwerkzeuges, wobei die dargestellten Komponenten sowohl Quellcode als auch Grammatiken und den daraus generierten Code enthalten.

Die in dieser Abbildung dargestellten Projekte sind dabei:

- **MC_Runtime**: Laufzeitkomponente des MontiCore Systems. Beinhaltet unter anderem Basisklassen für Lexer, Parser, ASTs, Workflows und Roots.
- **SymtabCoCo_Runtime**: Komponente mit Basisklassen für Symboltabellen und Kontextbedingungen.
- **LiteralsTypes**: Gemeinsame Basiskomponenten aller Sprachen. Beinhaltet die Grammatiken für Literale und Typen und darauf aufbauende Infrastrukturen für Symboltabellen und Kontextbedingungen.
- **Common**: Beinhaltet die Common-Grammatik und Basiselemente für deren Symboltabellen und Kontextbedingungen.
- **Statecharts**: Komponente der Statecharts mit Grammatik, Symboltabellen und Kontextbedingungen.

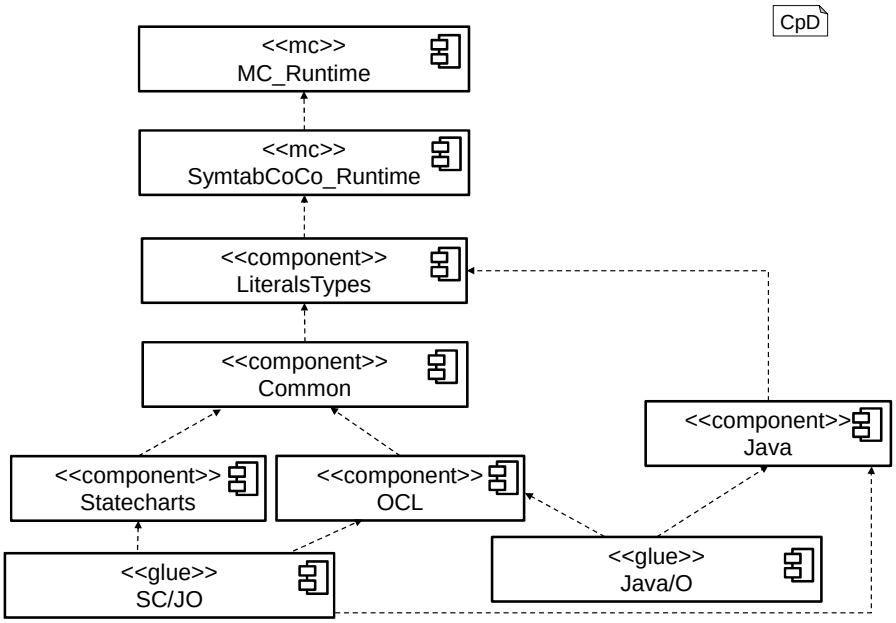


Abbildung 10.5.: Anordnung der Komponenten (ohne Editoren) SC/JO

- **OCL**: Beinhaltet die OCL-Grammatik und deren Symboltabellen und Kontextbedingungen.
- **Java**: Grammatik, Symboltabellen und Kontextbedingungen für Java 1.5.
- **SC/JO** : Beinhaltet die Definition der Kombination der Einzelsprachen zu SC/JO und zusätzliche Elemente zur Integration der Symboltabellen und Kontextbedingungen.
- **Java/O** : Komponente der Sprache Java/O mit Elementen zur Integration von Symboltabellen und Kontextbedingungen von Java und OCL.

Aus Übersichtsgründen wurde auf die Darstellung der Komponenten für Editoren verzichtet. Diese Komponenten bilden dabei jeweils eine Parallelstruktur zu den dargestellten Elementen und importieren die jeweils grammatikbeinhaltende Komponente.

10.3.2. Verbund der Grammatiken

Die beiden Sprachen SC/JO und Java/O wurden bezüglich der Grammatiken auf Basis unterschiedlicher Mechanismen realisiert. Abbildung 10.6 zeigt zunächst die Realisierung der Sprache SC/JO .

In dieser Abbildung sind die zwei wesentlichen Szenarien der kompositionalen Sprachentwicklung zu erkennen: Zunächst wird in Zeile 2 eine einzelne Sprache (Java-Blockstatements) eingebettet. Im Gegensatz dazu wird - erkennbar an den jeweiligen Parametern am Ende - in den Zeilen 5 und 7 die Einbettung von Sprachalternativen angewandt. Die Kombination der

MontiCore-Sprachdatei «hw»

Glue

```
1 //Einbettung von Java-Blockstatements als Aktionssprache
2 javadsl.JavaDSL.BlockStatement statements in sc.Statement;
3
4 //Verwendung der Java-Expressions als Sprache für Invarianten
5 javadsl.JavaDSL.Expression javaInvContent in sc.Invariant(java);
6 //Alternative Verwendung von OCL-Expressions
7 umlp.ocl.OCL.OCLExpression oclInvContent in sc.Invariant(ocl);
```

Abbildung 10.6.: Kombination der Grammatiken für die Sprache SC/JO

Sprachen beschränkt sich daher auf die Erstellung einer neuen Sprachdatei, neue Grammatiken wurden nicht erstellt.

Im Unterschied dazu wurde bei der Erstellung der Sprache Java/O eine neue Grammatik erstellt. Abbildung 10.7 zeigt die Grammatik auf.

MontiCore-Grammatik «hw»

Glue

```
1 grammar Java_OCL extends javadsl.JavaDSL, umlp.ocl.OCL{
2
3   //Überschreiben der Methodendeklaration
4   MethodDeclaration implements MemberDeclaration =
5     (OCLPreStatement)?
6     (Modifiers:Modifier)* TypeParameters ReturnType Name
7     "(" (Parameters:ParameterDecl ("," Parameters:ParameterDecl)*)?  ")"
8     ("throws" Throwables:QualifiedName ("," Throwables:QualifiedName)*)?
9     ( ";" | Block:BlockStatement );
10 }
```

Abbildung 10.7.: Grammatik der Sprache Java/O

Zur Umsetzung der Sprache wurde hierbei das Szenario der Nichtterminal-Ersetzung angewandt. Hierfür wurde eine Grammatik erstellt, die sowohl von der Java-Grammatik als auch von der OCL erbt (Zeile 1). Anschließend wurde das Nichtterminal der Methodendeklaration aus Java durch eine neues Nichtterminal mit optionaler OCL-Vorbedingung ersetzt (Zeilen 4-9).

10.3.3. Symboltabellen, Kontextbedingungen und Editoren

Neben den Sprachdateien und Grammatiken zur Kombination von konkreter und abstrakter Syntax wurden zusätzliche Komponenten in Form von Java-Klassen entwickelt, die zur Integration der Symboltabellen, der Kontextbedingungen und der Editoren dienen. Prinzipiell sind die Vorgehensweisen und Mechanismen bei allen Sprachkombinationen ähnlich, weshalb im Folgenden lediglich stellvertretende Beispiele aufgezeigt werden.

Übersetzung von Statecharts nach Java

Zur Interpretation eines Statecharts als Java-Typ sind Adaptoren entwickelt worden, die die verschiedenen Symboltabelleneinträge übersetzen. Abbildung 10.8 zeigt einen Auszug aus einem Adapter.

Java-Quellcode «hw»

```
1  public class StateChart2JavaTypeAdapter extends JavaTypeEntry {        Glue
2
3    protected StateChartEntry adaptee;
4    JavaTypeEntry bool=JavaTypeEntryCreator.create("boolean");
5
6    public StateChart2JavaTypeAdapter(StateChartEntry adaptee){
7      this.adaptee=adaptee;
8    }
9
10   @Override
11   public Set<MethodEntry> getMethods() {
12     if (methods==null){
13       methods=new HashSet<MethodEntry>();
14       String prefix="isIn_";
15       for (StateEntry e:adaptee.getSubStates()){
16         MethodEntry m=new MethodEntry();
17         m.setName(prefix+e.getName());
18         m.setPublic(true);
19         m.setReturnType(bool);
20         methods.add(m);
21       }
22     }
23     return methods;
24   }
25 }
```

Abbildung 10.8.: Auszug aus dem Adapter zur Übersetzung von Statecharts in Java-Typen

Der entwickelte Adapter entspricht der vorgeschlagenen Struktur aus Abschnitt 7.4.1. Hierbei ist der Adapter Subklasse des Zieleintrags mit einer Referenz auf den Ursprungseintrag (Zeile 1 und 3). Um die Übersetzung von Zuständen zu realisieren, wurde die Methode `getMethods` überschrieben (Zeilen 11-24). Hierin wurde für jeden Zustand eine neue öffentliche boolesche Methode unter Verwendung des Präfixes `isIn_` erstellt. Auf einen expliziten Adapter, welcher Zustände in Methoden übersetzt, konnte dabei verzichtet werden.

Auflösung von OCL-Variablen in Vorbedingungen

Bei der Kombination von OCL und Java müssen Java-Variablen in OCL-Variablen übersetzt werden. So ist im Beispiel aus Abbildung 10.4 im OCL-Teil (Zeile 11) eine Variable `u` verwendet worden, die Parameter der Java-Methode (Zeile 12) ist. Zur Übersetzung von Java nach OCL muss auch hier zunächst ein entsprechender Adapter wie bei der Übersetzung von Statecharts

nach Java entwickelt werden. Zusätzlich ist eine weitere Auflösungskomponente zu implementieren. Abbildung 10.9 zeigt diese Komponente.

Java-Quellcode «hw»

Glue

```
1 public class JavaFields2OCLFieldResolverClient
2                       extends AbstractResolverClient {
3
4   @Override
5   public String getResponsibleKind() {
6     return OCLFieldEntry.KIND;
7   }
8
9   @Override
10  public STEntry resolve(String name, NameSpace nsp){
11    STEntry javaField=resolver.resolve(name, JavaFieldEntry.KIND, nsp);
12    if (javaField!=null){
13      JavaField2OCLFieldAdapter ad=new
14              JavaField2OCLFieldAdapter((JavaFieldEntry) javaField);
15      return ad;
16    }
17    return null;
18  }
19 }
```

Abbildung 10.9.: Auszug aus der Klasse zur Auflösung von OCL-Variablen

Zur Umsetzung wurden dabei die in Abschnitt 7.4.3 vorgestellten Mechanismen zur Wiederverwendung von Auflösungskomponenten angewandt. Hierfür gibt die neue Komponente in Zeile 6 an, OCL-Variablen auflösen zu können. Bei der eigentlichen Suche ab Zeile 10 wird dann der `Resolver` beauftragt, eine gleichnamige Java-Variable aufzulösen. Hierbei werden intern existierende Komponenten der Sprache Java wiederverwendet. Ist die Auflösung erfolgreich, wird der entsprechende Eintrag adaptiert und zurückgeliefert (Zeilen 13 bis 15).

Kombination der Editoren

Zur Kombination der Editoren sind zwei verschiedene Mechanismen angewandt worden: für die Sprache SC/JO wurde eine neue Sprachdatei gemäß Abbildung 9.4 geschrieben. Hierin wurden unter anderem Dateiendung, auszuführende Workflows und die Startklasse des Tools spezifiziert. Für die Sprache Java/O wurde hingegen eine neue Grammatikdatei geschrieben, die per Mehrfachvererbung von der Java- und der OCL-Grammatik mit Editorspezifikationen erbt. Neben dieser Vererbung enthält die Grammatikdatei keinen weiteren Inhalt, sie dient lediglich zur Kombination der Editorattribute.

Ausgehend von Sprach- und Grammatikdatei wurde anschließend automatisch jeweils ein Plugin generiert. Dieses Plugin kombiniert dabei die Funktionalitäten der Editoren der Einzelsprachen und konnte ohne zusätzliche Implementierungen verwendet werden.

10.4. Wichtigste Ergebnisse und Bewertung

Bei der Umsetzung der Fallstudie kamen wesentliche Charakteristika der kompositionalen Sprachentwicklung zum Tragen. Daneben zeigten sich auch weitere wichtige Erkenntnisse bezüglich der Verwendbarkeit des in dieser Arbeit entwickelten Ansatzes. Die wesentlichsten Charakteristika und Erkenntnisse sind im Folgenden zusammengefasst.

Anwendung mehrerer Szenarien

Die Fallstudie zeigt auf, dass der entwickelte Ansatz die kompositionale Entwicklung von Sprachen ermöglicht. Diese Eigenschaft bezieht sich dabei nicht nur auf die Realisierung eines speziellen Szenarios, vielmehr sind in der Fallstudie mehrere Szenarien kompositionaler Sprachentwicklung gleichzeitig angewandt worden. Die Szenarien beeinflussten sich dabei nicht.

Anwendbarkeit und Verständlichkeit

Die verwendeten Teilsprachen Statecharts und OCL sind nicht vom Autor dieser Arbeit geschrieben worden, sondern von verschiedenen anderen Sprachentwicklern [Sch11, FM07, PS07, Hel10, Sch11]. Zusätzlich sind Teile der Sprache SC/JO in der Dissertation [Sch11] entstanden. Die weiteren Anteile am Gesamtsystem wurden spezifisch zur Realisierung der Fallstudie in dieser Arbeit entwickelt.

Insgesamt zeigte sich, dass die von anderen Entwicklern verfassten Einzelsprachen weitestgehend ohne Anleitung eingesetzt werden konnten. Die Teilsprachen konnten ohne Änderungen in das Gesamtsystem übernommen werden. Dies unterstreicht insgesamt die Verständlichkeit und Anwendbarkeit des Ansatzes.

Abstraktion

Die Umsetzung der Sprachkombinationen verlangte kein tiefes Wissen über die konkrete Implementierung der Einzelsprachen, das nötige Verständnis beschränkte sich auf wenige Kombinationspunkte (Teile der Grammatik, Entries in der Symboltabelle und Kontextbedingungen). Durch diese Abstraktion und Unabhängigkeit von der Implementierung konnten die Einzelsprachen leicht eingebunden werden.

Erfassung benötigter Komponenten

Sowohl bei der Umsetzung der Sprache Java/O durch den Autor der vorliegenden Arbeit als auch bei der Realisierung der Sprache SC/JO in [Sch11] konnten die benötigten Komponenten zur Sprachkombination leicht erfasst und implementiert werden. Hierbei zeigte sich, dass die konkreten Kombinationspunkte vor allem für Fremdanwender anhand der Sprachhierarchie auf Composite-Basis leicht zu lokalisieren sind.

Aufwand

Der Aufwand bei der Zusammenstellung des Gesamtsystems war wesentlich geringer als der Aufwand bei der Entwicklung der Einzelsprachen. Die Tabellen 10.10 bis 10.13 geben eine Übersicht über den Entwicklungsaufwand, wobei Tabelle 10.13 den Aufwand nach Art der Komponente aufschlüsselt.

Komponente	LOC Grammatiken	LOC generiert	LOC Java (handgeschrieben)
MC_Runtime	0	0	40100
Editor_Runtime	0	0	2500
SymtabCoCo_Runtime	15	4000	8000

Tabelle 10.10.: Anteile der vorgefertigten MontiCore-Artefakte («mc»)

Komponente	LOC Grammatiken	LOC generiert	LOC Java (handgeschrieben)
LiteralsTypes	400	31600	1640
Common	150	11200	1000
Statecharts	210	29400	3200
OCL	980	53500	8900
Java	1200	92300	16600
LiteralsTypesEditor	10	90	0
CommonEditor	5	70	0
StatechartEditor	15	120	0
OCLEditor	30	530	0
JavaEditor	40	770	0

Tabelle 10.11.: Anteile der einzelnen Sprachen («component»)

Zur Bestimmung der Kennzahlen wurden dabei nur diejenigen Anteile der Komponenten einbezogen, die unmittelbar Verwendung in der Fallstudie fanden. Daher ist die Komponente MC_Runtime nur teilweise einbezogen worden, auch für die Komponenten der Einzelsprachen wurden nicht alle Anteile einbezogen (z.B. Codegeneratoren).

Insgesamt unterstreichen die Kennzahlen den Gewinn bei der Anwendung kompositionaler Sprachentwicklung. Der Anteil für den Entwickler bei der Kombination der Sprachen (625 LOC) gegenüber dem Aufwand bei der Entwicklung der Einzelsprachen (34380 LOC) liegt dabei im unteren einstelligen Prozentbereich (ca. 1,9 % des Gesamtaufwandes). Weiterhin ist hierbei zu beachten, dass insbesondere die Kontextbedingungen für die Sprachen OCL und Java noch nicht vollständig umgesetzt sind. Durch eine vollständige Umsetzung würde sich das Aufwandsver-

Komponente	LOC Grammatiken	LOC generiert	LOC Java (handgeschrieben)
SC/JO	20	190	410
Java/O	30	1000	150
SC/JO Editor	10	380	0
Java/O Editor	5	380	0

Tabelle 10.12.: Anteile zur Sprachkombination («glue»)

Komponente	LOC Grammatiken	LOC generiert	LOC Java (handgeschrieben)
Summe «mc»-Komponenten	15	4000	50600
Summe «component»-Komponenten	3040	219580	31340
Summe «glue»-Komponenten	65	1950	560

Tabelle 10.13.: Aufschlüsselung nach Art der Komponenten

hältnis weiter verbessern.

Zusätzlich ist der durch den generativen Ansatz entstandene Aufwandsgewinn zu nennen. Das Schreiben einer Grammatik ist dabei mit geringerem Aufwand verbunden als die Implementierung eines Parsers und der entsprechenden abstrakten Syntax. Auch die Spezifikation der Editoren im Grammatikformat ist wesentlich schneller und einfacher als die Entwicklung eines handgeschriebenen Editors.

10.5. Zusammenfassung

In diesem Kapitel wurde die Fallstudie beschrieben, die zur Überprüfung der Anwendbarkeit des entwickelten Ansatzes durchgeführt wurde. Um zusätzlich die Verständlichkeit zu überprüfen, wurden hierbei Sprachen verwendet, die teilweise von anderen Entwicklern geschrieben wurden. Diesen Entwicklern waren dabei die in der Fallstudie implementierten Sprachkombinationen und Anwendungsszenarien nicht bekannt.

Insgesamt zeigte sich, dass durch die Verwendung des in dieser Arbeit beschriebenen Ansatzes eine relativ einfache Realisierung der Fallstudie ermöglicht wurde. Der Aufwand bei der Sprachkombination war dabei wesentlich geringer als der Aufwand bei der Entwicklung der Einzelsprachen. Zusammenfassend sind die einfache Anwendbarkeit, die Verständlichkeit für andere Entwickler, der verminderte Entwicklungsaufwand und die Möglichkeit zur gleichzeitigen Verwendung unterschiedlicher Szenarien die wesentlichsten Erkenntnisse der Fallstudie.

Neben der expliziten Fallstudie existieren in der MontiCore-Sprachbibliothek noch eine Viel-

zahl weiterer Sprachen und Sprachkombinationen. Diese verwenden die in dieser Arbeit beschriebenen Mechanismen und dienten als begleitende Studienobjekte zur Implementierung und Verbesserung des Ansatzes. Auch bei diesen Sprachen und Sprachkombinationen zeigten sich ähnliche Ergebnisse wie in der vorliegenden Fallstudie.

Teil IV.

Epilog

Kapitel 11.

Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf mögliche Erweiterungen gegeben.

11.1. Zusammenfassung

In dieser Arbeit wurde die kompositionale Entwicklung domänenspezifischer Sprachen untersucht. Die Hauptmotivation lag dabei in der Erkenntnis, dass vor allem bei der Anwendung von Modellierungstechniken eine Vielzahl von Sprachen verwendet wird, um das System in allen Facetten zu beschreiben. Die konkrete Zusammensetzung der Sprachen variiert dabei je nach Projektkontext, Projektziel oder Nutzerpräferenz. Bisherige Ansätze zur Umsetzung der Kompositionalität von Sprachen konzentrieren sich jedoch nur auf einen oder einige wenige Aspekte der Sprachentwicklung, eine durchgängige Unterstützung existiert bisher nicht. Gerade aber diese Durchgängigkeit ist von zentraler Bedeutung, da es nur so möglich ist, komplette Sprachen in Bibliotheken zu hinterlegen und bei Bedarf unter geringem Aufwand wiederzuverwenden.

Diese bibliotheksartige Vorgehensweise ist dabei keineswegs neu in der Informatik, sondern auch schon Grundidee der komponentenbasierten Softwareentwicklung. Hierbei werden komplexe Systeme aus vorgefertigten Komponenten zusammengesetzt, um somit nicht nur den Aufwand zu reduzieren, sondern gleichzeitig Qualität und Flexibilität zu erhöhen. Auch die Erweiterbarkeit, die bessere Testbarkeit und die erhöhte Verständlichkeit zählen zu den Vorteilen komponentenbasierter Systeme. Diese Vorteile konnten im Verlaufe dieser Arbeit auf die Sprachentwicklung übertragen werden.

Als technische Grundlage für die Umsetzung wurde dabei das MontiCore-Framework gewählt. Die wesentlichen genutzten Merkmale sind dabei die integrierte Definition konkreter und abstrakter Syntax, ein unterstützendes Framework für die Verarbeitung von Modellen und erste Ansätze zur Modularisierung von Sprachen bezüglich der konkreten und abstrakten Syntax. Diese Ansätze wurden während dieser Arbeit konsequent weiterentwickelt und ergänzt.

Die zentrale Fragestellung bei der Untersuchung kompositionaler Sprachentwicklung sollte jedoch nicht auf der technischen Strategie liegen, sondern vielmehr im zu lösenden Problem aus Nutzersicht. Daher wurden in dieser Arbeit konkrete Szenarien identifiziert, anhand derer die verschiedenen Artefakte der Sprachentwicklung und insbesondere deren Kompositionalitätsaspekte untersucht wurden. Durch die Betrachtung verschiedener existierender Sprachkombinationen wurden dabei insgesamt sechs verschiedene Szenarien identifiziert: Sprachvereinigung, Nichtterminal-Erweiterung, Nichtterminal-Ersetzung, Einbettung einer einzelnen Sprache, Ein-

bettung von Sprachalternativen und die Entwicklung einer erweiterbaren Sprache. Durch die Identifizierung dieser Szenarien und der Diskussion der Realisierungsstrategien ist es Sprachnutzern möglich, ihr konkretes Anwendungsproblem zu kategorisieren und anschließend anhand der vorgestellten Vorgehensweisen auf den unterschiedlichen Sprachaspekten zu realisieren. Die wichtigsten Ergebnisse und Vorgehensweisen für die einzelnen Aspekte werden im Folgenden kurz zusammengefasst.

Konkrete Syntax.
Zur Umsetzung der Kompositionalität für die konkrete Syntax wurde ANTLR als verwendeter Parsergenerator so erweitert, dass während des Erkennungsprozesses ein Wechsel von Lexer und Parser ermöglicht wurde. Durch den auf dieser Technik basierenden Mechanismus der Einbettung und die weiteren Mechanismen Sprachvererbung und Sprachaggregation konnten die Szenarien teilweise auf unterschiedliche Art umgesetzt werden. Die Auswahl der verwendeten Strategie hängt dabei von den gegebenen Sprachen und deren Eigenschaften ab.

Abstrakte Syntax.
Auch bei der abstrakten Syntax zeigte sich, dass mehrere Strategien zur Umsetzung der Szenarien verwendet werden können. Die auf der abstrakten Syntax basierenden Visitoren unterstützen dabei alle Strategien und können dank der flexiblen Architektur unterschiedlichste Anforderungen umsetzen.

Symboltabellen.
Der Symboltabellen-Infrastruktur liegt eine speziell auf Kompositionalität ausgelegte Architektur zugrunde. Diese verfügt dabei über zwei Kerneigenschaften: zum einen können Symboltabellen für Sprachen separat entwickelt und unter Einbeziehung von Adaptoren und zusätzlichen Resolvern mit Symboltabellen anderer Sprachen kombiniert werden. Zum anderen ermöglicht die explizite Erstellung von Modellschnittstellen eine effiziente Verarbeitung großer Systeme, da für die Korrektheitsprüfung eines Modells lediglich die Schnittstellen anderer Modelle einbezogen werden müssen. Auch die Symboltabellen unterstützen alle Szenarien.

Kontextbedingungsprüfungen.
Der entwickelte Ansatz zur Prüfung von Kontextbedingungen beachtet nicht nur die Kompositionalität, sondern bezieht auch unterschiedlichste Anforderungen speziell von Modellierungssprachen mit ein. Diese Anforderungen beinhalten unter anderem die Konfigurierbarkeit, die Erweiterbarkeit und die Selektierbarkeit in Verbund mit einer möglichen Gruppierung. Durch den Einsatz der speziellen Symboltabellen-Infrastruktur können in den meisten Fällen existierende Kontextbedingungsprüfungen ohne Änderungen wiederverwendet werden, da die Adaption der Symboltabelleneinträge bereits die Vermittlung zwischen den Sprachen ermöglicht.

Sprachspezifische Werkzeuge.
Dieser wichtige, doch aufgrund des hohen Aufwandes oftmals vernachlässigte Aspekt wird durch die Verwendung eines generativen Ansatzes wesentlich erleichtert. Sprachentwickler spezifizieren im erweiterten Grammatikformat lediglich die gewünschten Eigenschaften der Werk-

zeuge, der Generator erstellt hieraus automatisch ein auslieferbares Eclipse-Plugin. Hierdurch wird der Aufwand für den Sprachentwickler auf ein Minimum reduziert, da einerseits eine Einarbeitung in die umfangreiche Eclipse-API entfällt und andererseits die recht häufigen Änderungen in dieser API im Generator umgesetzt und somit automatisch in allen Sprachen reflektiert werden. Die Kompositionalität wird auch bezüglich der sprachspezifischen Werkzeuge unterstützt.

Nicht nur in der Fallstudie, sondern vielmehr auch in vielen anderen Sprachen und Sprachkombinationen, welche den in dieser Arbeit beschriebenen Ansatz anwenden, zeigen sich die wesentlichen Merkmale des entwickelten Systems. Hierzu zählen unter anderem die einfache Anwendbarkeit, die Verständlichkeit für andere Entwickler, der reduzierte Aufwand und die Möglichkeit zur gleichzeitigen Umsetzung unterschiedlicher Szenarien.

Insgesamt zeigt sich, dass die kompositionale Entwicklung von Sprachen für alle behandelten Aspekte umgesetzt werden konnte. Dabei ermöglichen die Umsetzungen aufgrund der gemeinsam verwendeten Szenarien eine nahtlose Integration der Kompositionalitätsmechanismen für die konkrete Syntax, für die abstrakte Syntax, für die Symboltabellen, für Kontextbedingungsprüfungen und für sprachspezifische Werkzeuge. Dadurch können komplette und vor allem qualitätsgesicherte Sprachen in Bibliotheken hinterlegt und in verschiedensten Projekten mit anderen Sprachen kombiniert und somit wiederverwendet werden. Der Aufwand zur Kombination ist dabei minimal gegenüber dem Aufwand bei der Entwicklung der einzelnen Sprachen. Diese Flexibilität unterstützt dabei letztendlich insbesondere die Entwickler komplexer Systeme, die anhand aktueller Techniken und Methodiken von einer agilen Verwendung unterschiedlicher Sprachen profitieren.

11.2. Ausblick

Eine Weiterentwicklung der Ansätze dieser Arbeit ist in vielen Richtungen denkbar. Im Folgenden werden drei grundlegende Ideen beschrieben.

Codegenerierung.
Ein wesentlicher Aspekt bei der Entwicklung von Sprachen ist die Entwicklung eines Codegenerators, der Instanzen der Eingabesprache in ausführbaren Code umwandelt. In Bezug auf die Kompositionalität sollte es auch hier Zielsetzung sein, Codegeneratoren für alle involvierten Sprachen separat zu entwickeln und diese in einem späteren Schritt zu kombinieren.

Eine wesentliche Voraussetzung zur Umsetzung einer kompositionalen Codegenerierung ist dabei die Möglichkeit, Informationen über Sprachgrenzen hinweg auszutauschen, da die Generierung von Code für ein gegebenes Modell oftmals stark von Eigenschaften anderer Elemente in womöglich anderen Sprachen abhängt. So ist beispielsweise die Umsetzung von Statecharts in Java auf verschiedene Weise möglich (State-Pattern, Zustände als Enum-Werte oder int-Konstanten etc.), der die Statecharts aufrufende oder den aktuellen Zustand auswertende Code muss die konkrete Umsetzungsweise beachten. Daher müssen die entsprechenden Codegeneratoren dieses Wissen austauschen, wobei eine zu starke Kopplung zu vermeiden ist. Als Grun-

didee zum Austausch dieser Informationen zwischen Codegeneratoren können die Strukturen und Vorgehensweisen beim Informationsaustausch zwischen Visitoren und Attributberechnungen aus dieser Arbeit dienen.

Kompositionale Transformationen und Refactorings.
Neben der Codegenerierung sind auch generelle Transformationen sowie Refactorings ein wichtiger Aspekt der Sprachentwicklung. In Bezug auf die Kompositionalität gelten auch hier ähnliche Randbedingungen: Transformationen/Refactorings müssen für alle beteiligten Sprachen separat entwickelt und anschließend kombiniert werden. Zur Kombination ist der Austausch von Informationen zwingend notwendig, da eine Transformation in einem Modell einer Sprache Transformationen in Modellen anderer Sprachen nach sich ziehen kann.

Als technische Grundidee zur Umsetzung bietet sich die Refactoring-Engine von Eclipse an. Diese verfügt über ein Observer-Mechanismus, bei der sich Clients über erfolgte Refactorings informieren lassen können. Diese Clients können anschließend die Änderungen in ihren Gebieten/Modellen nachziehen, wodurch auch sprachübergreifende Refactorings umgesetzt werden können. Leider ist der Eclipse-Mechanismus auf einfache Refactorings beschränkt, eine Verallgemeinerung und Ausweitung wäre hier notwendig.

Co-Evolution von Sprachen und Modellen.
Der in dieser Arbeit entwickelte Ansatz ermöglicht die unmittelbare Reflektion von Änderungen an einer Sprache in allen Sprachkombinationen, an denen die Eingangssprache beteiligt ist. Hierdurch sind die Sprachkombinationen stets auf demselben aktuellen Stand wie die beteiligten Einzelsprachen.

Als Folge dieser unmittelbaren Reflektion kann es aber zu Situationen kommen, in denen die Modelle der kombinierten Sprache invalide werden, da sie nicht mehr konform zur geänderten Eingangssprache sind. Dieser Problematik kann mit einer Co-Evolution von Sprachen und Modellen begegnet werden, wobei nicht nur die geänderten Sprachen veröffentlicht werden, sondern gleichzeitig entsprechende Refactorings zur Änderung der Modelle. Diese können dann neben der Änderung der Sprache als weitere Maßnahme auf die Modelle der Sprachkombinationen angewandt werden.

Literaturverzeichnis

[Ada91] Stephen R. Adams. *Modular Grammars for Programming Language Prototyping*. Doktorarbeit, University of Southhampton, 1991.

[Aßm03] U. Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[ASU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ber03] Hans Bergsten. *JavaServer pages*. O'Reilly Media, 2003.

[BHD+01] Mark van den Brand, Jan Heering, Arie van Deursen, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction (CC) 2001*, LNCS 2102. Springer, 2001.

[Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin-Cummings, 1987.

[BS86] Rolf Bahlke, Gregor Snelting. The PSG System: From Formal Language Definitions To Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8:547–576, 1986.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[BV07] Martin Bravenboer, Eelco Visser. Designing Syntax Embeddings and Assimilations for Language Libraries. In *4th International Workshop on Software Language Engineering*, 2007.

[BVVV05] Martin Bravenboer, Rob Vermaas, Jurgen J. Vinju, Eelco Visser. Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax. In *Proc. of International Conference on Generative Programming and Component Engineering (GPCE) 2005*, LNCS 3676. Springer, 2005.

[Car97] Luca Cardelli. *Type Systems*, Kapitel 103. CRC Press, Boca Raton, FL, 1997.

[CCL06] Ivica Crnkovic, Michel Chaudron, Stig Larsson. Component-based development process and component lifecycle. In *ICSEA '06: Proceedings of the International*

Conference on Software Engineering Advances, Seite 44, Washington, DC, USA, 2006. IEEE Computer Society.

[CJK07] Steve Cook, Gareth Jones, Stuart Kent. *Domain Specific Development with Visual Studio DSL Tools*. Addison-Wesley Longman, 2007.

[Cle07] Thomas Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. Doktorarbeit, Vrije Universteit Brussel, 2007.

[Cre94] Regis Crelier. *Separate Compilation and Module Extension*. Doktorarbeit, Swiss Federal Institute of Technology, Zurich, 1994.

[CT04] Keith D. Cooper, Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[DDZ08] Jürgen Dingel, Zinovy Diskin, A. Zito. Understanding and Improving UML Package Merge. *Software and Systems Modeling*, 7(4):443–467, October 2008.

[DHKL84] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, Bernard Lang. Programming Environments Based on Structured Editors: The MENTOR Experience. In *Interactive Programming Environments*. McGraw-Hill, 1984.

[dJNNKV09] Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats, Eelco Visser. Natural and Flexible Error Recovery for Generated Parsers. In *SLE*, Seiten 204–223, 2009.

[DKV00] Arie van Deursen, Paul Klint, Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[DRRS09] Michael Dukaczewski, Dirk Reiss, Bernhard Rumpe, Mark Stein. MontiWeb - Modular Development of Web Information Systems. In M. Rossi, J. Sprinkle, J. Gray, J.-P. Tolvanen (Editoren), *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, 2009.

[DW99] Sesmond D'Souza, Alan Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[Ear70] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13:2:94–102, 1970.

[EBN] EBNF Spezifikation ISO/IEC 14977:1996, `www.iso.org/iso/catalogue_detail.htmc?snumber=26153`.

[Ecl] Eclipse Website `http://www.eclipse.org`.

[EH07] Torbjörn Ekman, Görel Hedin. The JastAdd System - Modular Extensible Compiler Construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[ES06] Matthew Emerson, Janos Sztipanovits. Techniques for Metamodel Composition. In *OOPSLA – 6th Workshop on Domain Specific Modeling*, Seiten 123–139, 2006.

[FM07] Christoph Ficek, Fabian May. Umsetzung der Java 5 Grammatik für MontiCore. Studienarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2007.

[Fre] Freemarker Website `http://freemarker.org/`.

[fS06] International Organization for Standardization. Iso/iec 9075-14:2006, 2006.

[GB03] E. Gamma, K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[GC03] Joseph D. Gradecki, Jim Cole. *Mastering Apache Velocity*. Wiley, 2003.

[GE99] Ralf Güting, Martin Erwig. *Übersetzerbau. Techniken, Werkzeuge, Anwendungen*. Springer-Verlag, 1999.

[GH98] Etienne Gagnon, Laurie Hendren. SableCC – an Object-Oriented Compiler Framework. In *Proceedings of TOOLS 1998*, 1998.

[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 Auflage, 2005.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Technischer Bericht Informatik-Bericht 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 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, Seiten 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Proceedings of Modellierung 2006 (LNI P-82)*, 2006.

[GMP03] Blaise Genest, Anca Muscholl, Doron Peled. Message Sequence Charts. In *Lectures on Concurrency and Petri Nets*, Seiten 537–558, 2003.

[Gou01] John Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall, November 2001.

[Grö10] H. Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten*. Doktorarbeit, RWTH Aachen, 2010.

[Graa] Graphical Modeling Framework Website. `www.eclipse.org/gmf/`.

[Grab] Graphviz Website, `www.graphviz.org`.

[Gro03] Christian Grothoff. Walkabout Revisited: The Runabout. In *ECOOP*, Seiten 103–125, 2003.

[Gro05] Hans-Gerhard Gross. *Component-based Software Tessting with UML*. Springer Verlag, 2005.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Upper Saddle River, NJ, 2009.

[GS08] Miguel Garcia, Paul Sentosa. Generation of Eclipse-based IDEs for Custom DSLs. Technischer Bericht, Software Systems Institute, Technische Universität Hamburg-Harburg, 2008.

[Hau97] Juha Hautamäki. A Survey of Frameworks. Technischer Bericht A-1997-3, Department of Computer Science, University of Tampere, 1997.

[HC01] George T. Heineman, William T. Councill (Editoren). *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Hed00] Görel Hedin. Reference Attributed Grammars. *Informatica (Slovenia)*, 24(3), 2000.

[Hel10] Ulrich Helker. Integration der Object Constraint Language in die UML/P. Diplomarbeit, Lehrstuhl Informatik 3 (Software Engineering), RWTH Aachen, 2010.

[HHKR89] Jan Heering, P. R. H. Hendriks, Paul Klint, Jan Rekers. The Syntax Definition Formalism SDF - Reference Manual. *Sigplan Notices*, 24(11):43–75, 1989.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, R. F. Paige (Editoren), *Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, LNCS 4530, Seiten 99–113, Haifa, Israel, June 2007. Springer.

[HKR+08] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. Scaling-Up Model-based Development for Large Heterogeneous Systems with Compositional Modelling. In *15th Monterey Workshop*, Budapest, Hungary, 2008.

[HKRR11] Arne Haber, Thomas Kutz, Jan Oliver Ringert, Bernhard Rumpe. MontiArc 1.1.3 - Architectural Modeling Of Interactive Distributed Systems. Technischer Bericht, RWTH Aachen University, 2011. (to appear).

[HR04] David Harel, Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[HTM] HTML Spezifikation, `http://www.w3.org/html/wg/html5/`.

[Hud98] Paul Hudak. Modular Domain Specific Languages and Tools. In *Fifth International Conference on Software Reuse*, Seiten 134–142, 1998.

[IMP] IMP - The Eclipse IDE Meta-tooling Platform Website. `http://www.eclipse.org/imp/`.

[JB06] Frederic Jouault, Jean Bezivin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (LNCS 4037)*, Seiten 171–185, 2006.

[JF88] Ralph E. Johnson, Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[KdJNNV09] Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, Eelco Visser. Providing Rapid Feedback in Generated Modular Language Environments: Adding Error Recovery to Scannerless Generalized-LR Parsing. In *OOPSLA*, Seiten 445–464, 2009.

[KKV08] Lennart C. L. Kats, Karl Trygve Kalleberg, Eelco Visser. Generating Editors for Embedded Languages. Integrating SGLR into IMP. In A. Johnstone, J. Vinju (Editoren), *Language Descriptions, Tools, and Applications (LDTA'08)*, Electronic Notes in Computer Science, Budapest, Hungary, April 2008. Elsevier.

[KKV09] Lennart C. L. Kats, Karl Trygve Kalleberg, Eelco Visser. Domain-Specific Languages for Composable Editor Plugins. In *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, April 2009.

[Kle07a] Anneke Kleppe. A Language Description is More than a Metamodel. In *Proceedings of Fourth International Workshop on Software Language Engineering, 1 Oct 2007, Nashville, USA.*, 2007.

[Kle07b] Anneke Kleppe. Towards the Generation of a Text-Based IDE from a Language Metamodel. In *ECMDA-FA*, Seiten 114–129, 2007.

[Knu68] Donald F. Knuth. Semantics of Context-Free Languages. *Mathematical systems theory*, 12:127–145, 1968.

[Knu79] Donald E. Knuth. *TEX and METAFONT: New Directions in Typesetting*. American Mathematical Society, Boston, MA, USA, 1979.

[KPKP06] Dimitrios Kolovos, Richard Paige, Tim Kelly, Fiona Polack. Requirements for Domain-Specific Languages. In *1st ECOOP Workshop on Domain-Specific Program Development (DSPD)*, Nantes, France, July 2006.

[KPP06a] Dimitrios S. Kolovos, Richard F. Paige, Fiona Polack. Merging Models with the Epsilon Merging Language (EML). In *MoDELS*, Seiten 215–229, 2006.

[KPP06b] Dimitrios S. Kolovos, Richard F. Paige, Fiona A. C. Polack. Eclipse Development Tools for Epsilon. In *In Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.

[KPP08] 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, Seiten 69–73, New York, NY, USA, 2008. ACM.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Doktorarbeit, RWTH Aachen University, 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*, Seiten 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*, volume 11 of *Lecture Notes in Business Information Processing*. Springer-Verlag 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, September 2010.

[KS97] Peter Klein, Andy Schürr. Constructing SDEs with the IPSEN Meta Environment. In *SEE '97: Proceedings of the 8th International Conference on Software Engineering Environments (SEE '97)*, Seite 2, Washington, DC, USA, 1997. IEEE Computer Society.

[KT08] Steven Kelly, Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[Läm01] Ralf Lämmel. Grammar Adaptation. In *Proceedings of Formal Methods Europe (FME) 2001 (LNCS 2021)*, Seiten 550–570. Springer-Verlag, 2001.

[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.

[Lou02] Kenneth C. Louden. *Programming Languages: Principles and Practice*. Brooks/-Cole Publishing Company, 2002.

[MBB06] Erik Meijer, Brian Beckman, Gavin Bierman. LINQ: Reconciling Objects, Relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, Seiten 706–706, New York, NY, USA, 2006. ACM.

[McI68] Douglas McIlroy. Mass-Produced Software Components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, Seiten 88–98, 1968.

[Met] MetaCase Website `http://www.metacase.com`.

[MHS05] Marjan Mernik, Jan Heering, Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[MŽLA99] Marjan Mernik, Viljem Žumer, Mitja Lenič, Enis Avdičaušević. Implementation of Multiple Attribute Grammar Inheritance in the Tool LISA. *SIGPLAN Not.*, 34(6):68–75, 1999.

[Nag96] Manfred Nagl (Editor). *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*. Springer, 1996.

[NMFR10] C. Pinkernell S. Plesser N. M. Fisch, T. Kurpick, B. Rumpe. The Energy Navigator - A Web based Platform for Functional Quality Mangement in Buildings. In *Proceedings of the 10th International Conference for Enhanced Building Operations (ICEBO 10)*, Kuwait City, Kuwait, October 2010.

[OAW] OpenArchitectureWare Website `www.openarchitectureware.com/`.

[OMG10a] Object Management Group. Object Constraint Language Version 2.2 (OMG Standard 2010-02-01), 2010.
`http://www.omg.org/spec/OCL/2.2/PDF`.

[OMG10b] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10-05-03), May 2010. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/.

[OMG10c] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[Par72] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first Auflage, May 2007.

[PB03] Rachel A. Pottinger, Philip A. Bernstein. Merging Models Based on Given Correspondences. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, Seiten 862–873, 2003.

[PJ98] Jens Palsberg, C. Barry Jay. The Essence of the Visitor Pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC, Vienna, Austria*, Seiten 9–15, Los Alamitos, California, August 1998. IEEE.

[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] Claas Pinkernell, Sören Schulze. Integration der Object Constraint Language in die UML/P. Studienarbeit, Institut für Software Systems Engineering, Technische Universität Brauschweig, 2007.

[RBB06] Andreas Rausch, Manfred Broy, Klaus Bergner. *Das V-Modell XT. Grundlagen, Methodik und Anwendungen*. Springer, 2006.

[RNS04] T. Ritter, B. Neubauer, F. Stoinski. *CORBA Komponenten. Effektives Software-Design und Programmierung.* Springer-Verlag, 2004.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, Technische Universität München, 1996.

[Rum04a] Bernhard Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer, 2004.

[Rum04b] Bernhard Rumpe. *Modellierung mit UML.* Springer, 2004.

[SC89] Daniel J. Salomon, Gordon V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. *SIGPLAN Not.*, 24(7):170–178, 1989.

[SCC06] Charles Simonyi, Magnus Christerson, Shane Clifford. Intentional Software. In *OOPSLA*, Seiten 451–464, 2006.

[Sch11] Martin Schindler. *Eine Werkzeuginfrastruktur zur Agilen Entwicklung mit der UML/P*. Doktorarbeit, RWTH Aachen, 2011. to appear.

[Spi01] Diomidis Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.

[Ste08] Jens Stephani. Erweiterung von MontiCore zur Verwendung von Attributen. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

[Str00] Bjarne Stroustrup. *Die C++-Programmiersprache*. Addison Wesley, 2000.

[SVB02] Thorsten Sturm, Jesco von Voss, Marko Boger. Generating Code from UML with Velocity Templates. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, Seiten 150–161, London, UK, 2002. Springer-Verlag.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Vis97] Eelco Visser. Scannerless Generalized-LR Parsing. Technischer Bericht P9707, University of Amsterdam, 1997.

[Vog93] Harald H. Vogt. *Higher order Attribute Grammars*. Doktorarbeit, University of Utrecht, 1993.

[VWH09] Eric Van Wyk, Mats Per Erik Heimdahl. Flexibility in Modeling Languages and Tools: a Call to Arms. *Journal on Software Tools for Technology Transfer*, 11(3):203–215, 2009.

[Win00] Andreas Winter. *Referenz-Metaschema für visuelle Modellierungssprachen*. Doktorarbeit, Wiesbaden, 2000. Dissertation. Institut für Informatik, Universität Koblenz-Landau.

[Win02] Shuly Wintner. Modular Context-Free Grammars. *Grammars*, 5(1):41–63, 2002.

[Wit07] Steffen Witt. Entwurf und Implementierung einer erweiterbaren Qualitätssicherungsschnittstelle für Codierungsrichtlinien im automobilen Forschungsumfeld für das Framework MontiCore am Beispiel der Sprache C++. Master's thesis, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2007.

[WMBK02] Eric van Wyk, Oege de Moor, Kevin Backhouse, Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of the 11th International Conference on Compiler Construction 2002*, Seiten 128–142, London, UK, 2002. Springer-Verlag.

[WS07] Eric R. Van Wyk, August C. Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, Seiten 63–72, New York, NY, USA, 2007. ACM.

[WS08] Ingo Weisemöller, Andy Schürr. Formal Definition of MOF 2.0 Metamodel Components and Composition. In *MoDELS*, Seiten 386–400, 2008.

Teil V.

Anhänge

Anhang A.

Glossar

Die folgenden Definitionen sind während dieser Arbeit entstanden oder basieren auf [Kra10] und [Grö10].

Abstrakter Syntaxbaum (AST)
: Der abstrakte Syntaxbaum ist die innere Repräsentationsform eines →Modells. Dabei werden die AST-Klassen vom →Parser während der →syntaktischen Analyse gemäß der Zusammensetzung des aktuellen →Modells instanziiert und von folgenden Teilen des →Compilers als Grundlage verwendet.

Abstrakte Syntax
: Die abstrakte Syntax beschreibt die innere Struktur einer Sprache und legt somit die Menge aller möglichen →abstrakten Syntaxbäume fest.

Adapter
: Ein Adapter ist ein grundlegendes Design-Pattern gemäß [GHJV95]. Es wird eingesetzt, um die Schnittstelle einer Klasse in eine andere Schnittstelle zu übersetzen und somit diese Klasse in nicht vorgesehenen Kontexten zu verwenden.

Attributierte Grammatik
: Attributierte Grammatiken [Knu68] bilden ein Mittel, um Attributflüsse innerhalb von →abstrakten Syntaxbäumen zu beschreiben. Hierbei werden Grammatiksymbole (→Terminale und →Nichtterminale) um sogenannte Attribute ergänzt und entsprechende Berechnungsvorschriften an den Grammatikregeln angegeben.

Compiler
: Ein Compiler ist eine Software, die Modelle einer Quellsprache einliest und in Modelle einer Zielsprache umwandelt. Der Compilierungsvorgang ist dabei in folgende Phasen gegliedert:

  - →Lexikalische Analyse durch den →Lexer
  - →Syntaktische Analyse durch den →Parser
  - Aufbau einer →Symboltabelle
  - Prüfung von →Kontextbedingungen (oft auch als semantische Analyse bezeichnet)
  - Generierung einer oder mehrerer Dateien in der Zielsprache durch den →Generator

Compilezeit
Die Compilezeit ist der Zeitpunkt, zu dem ein →Modell der Quellsprache durch den →Compiler in Modelle der Zielsprache übersetzt wird.

Domäne
Eine Domäne ist ein Fachgebiet, in dem Domänenexperten mit spezialisiertem Fachwissen arbeiten.

Domänenspezifische Sprache (DSL)
Eine DSL ist eine für eine →Domäne speziell entworfene Sprache und enthält daher domänenspezifische Konzepte. Daneben charakterisieren sich DSLs durch ihren eingeschränkten Sprachumfang und die fehlende Berechnungsuniversalität.

Einbettung
Einbettung ist ein →Mechanismus der kompositionaler Sprachentwicklung. Hierbei definiert eine Sprache ein externes →Nichtterminal, das zur →Konfigurations- oder →Laufzeit durch ein Nichtterminal einer anderen Sprache ersetzt wird.

Einbettung einer einzelnen Sprache
Die Einbettung einer einzelnen Sprache ist ein →Szenario der kompositionalen Sprachentwicklung. Hierbei wird zur →Konfigurationszeit einmalig festgelegt, wodurch ein →externes Nichtterminal einer Sprache ersetzt wird.

Einbettung von Sprachalternativen
Die Einbettung von Sprachalternativen ist ein →Szenario der kompositionalen Sprachentwicklung. Hierbei wird im Gegensatz zur →Einbettung einer einzelnen Sprache erst zur →Laufzeit, d.h. während der Erstellung des →Modells festgelegt, wodurch ein →externes Nichtterminal einer Sprache ersetzt wird. Demnach existieren für die Ersetzung Alternativen.

Entry
Ein Entry ist ein Eintrag der →Symboltabelle für ein konkretes Element des →Modells. Entries haben dabei eine für die →Sorte des Elementes spezialisierte Form, durch die sie in der Lage sind, spezifische Informationen aufzunehmen und wiederzugeben.

Entwicklung einer erweiterbaren Sprache
Die Entwicklung einer erweiterbaren Sprache ist eines der →Szenarien der kompositionalen Sprachentwicklung. Hierbei wird die Sprache von vornherein so gestaltet, dass sie in späteren Schritten leicht und auf verschiedene Art und Weise mit anderen Sprachen kombiniert werden kann.

Entwicklungszeit
Die Entwicklungszeit bezeichnet die Zeitspanne, in der eine Software oder eine Sprache erstellt wird.

Externes Nichtterminal
Ein externes Nichtterminal besitzt keinen Produktionskörper und wird durch den Mechanismus der →Einbettung zur →Konfigurations- oder →Laufzeit durch ein anderes Nichtterminal ersetzt.

Fabrik
Eine Fabrik ist ein grundlegendes Design-Pattern gemäß [GHJV95]. Die Fabrik ist dabei die einzige Stelle, in der ein Objekt einer Klasse instanziiert wird. Diese Instanziierung erfolgt in einer Fabrikmethode, die von anderen Teilen des Codes aufgerufen wird.

Generator
Der Generator ist Teil des →Compilers und erstellt aus einem →abstrakten Syntaxbaum Modelle der Zielsprache.

Grammatik
Grammatiken bilden ein Mittel zur Definition der →konkreten Syntax einer Sprache. Sie bestehen aus →Terminalen, →Nichtterminalen und →Produktionen. Grammatiken im MontiCore-Format werden auch als →MontiCore-Grammatiken bezeichnet.

Grammatikfragment
Ein Grammatikfragment ist eine Grammatik mit →externen Nichtterminalen.

Grammatikvererbung
Die Grammatikvererbung ist ein →Mechanismus zur kompositionalen Sprachentwicklung. Hierbei können Grammatiken von ein oder mehreren Obergrammatiken erben, wodurch deren Nichtterminale und Terminale in der erbenden Grammatik sichtbar und somit verwendbar sind.

Konfigurationszeit
Die Konfigurationszeit ist der Zeitpunkt, zu dem ein Programm konfiguriert wird und beginnt meist mit dem Start des Programms. Im Anschluss an die Konfigurationszeit wird zur →Laufzeit übergegangen.

Kompositionsoperator
Der Begriff „Kompositionsoperator“ kann in einen syntaktischen und einen semantischen Operator unterschieden werden. Der syntaktische Operator beschreibt, wie auf syntaktischer Ebene aus zwei Eingabemodellen ein komponiertes →Modell erstellt werden kann. Der semantische Operator definiert hingegen, wie die →Semantiken der Eingabemodelle komponiert werden.

Kompositionalität
Als Kompositionalität wird das Prinzip verstanden, wonach sich die →Semantik eines Kompositums aus den Semantiken der Einzelteile und der Semantik des →Kompositionsoperators zusammensetzt. Aus technischer Sicht bezeichnet Kompositionalität die Eigenschaft meist komplexer Softwaresysteme, sich aus einzelnen, vorgefertigten Bausteinen unter Einbeziehung weniger zusätzlicher Artefakte zusammenzusetzen.

Konkrete Syntax
Die konkrete Syntax bezeichnet die gegenüber dem Benutzer sichtbare Repräsentationsform einer Sprache. In dieser Arbeit wird eine textuelle konkrete Syntax verwendet, die durch eine →MontiCore-Grammatik definiert wird.

Kontextbedingung
Kontextbedingungen sind Bedingungen, welche die Menge der gültigen →Modelle weiter einschränken. Die Korrektheitsprüfung geht dabei typischerweise über die Möglichkeiten kontextfreier →Grammatiken hinaus.

Laufzeit
Die Laufzeit bezeichnet die Zeitspanne, in der ein konfiguriertes Programm ausgeführt wird. Sie schließt sich an die →Konfigurationszeit an und endet mit Beendigung des Programms.

Lexer
Der Lexer ist der Teil des →Compilers, der die →lexikalische Analyse übernimmt.

Lexikalische Analyse
Bei der lexikalischen Analyse wird der Zeichenstrom der Eingabe in →Token überführt.

Mechanismus
Als Mechanismen werden in dieser Arbeit die drei Verfahren →Einbettung, →Grammatikvererbung und →Sprachaggregation als technische Grundlage für die Umsetzung der →Szenarien bezeichnet.

Modell
Ein Modell bezeichnet eine Instanz einer →domänenspezifischen Sprache oder einer →Programmiersprache.

MontiCore-Grammatik
Eine MontiCore-Grammatik bezeichnet eine →Grammatik im MontiCore-Format. Sie spezifiziert die →konkrete und →abstrakte Syntax einer Sprache.

Namespace
Ein Namespace ist ein Abschnitt eines →Modells, in dem Namen und zugehörige →Entries gemeinsam verwaltet werden.

Nichtterminal
Ein Nichtterminal ist Teil einer →Grammatik und wird in MontiCore sowohl zur Spezifikation der →konkreten Syntax als auch als Basis für die Ableitung der →abstrakten Syntax verwendet.

Nichtterminal-Ersetzung
Die Nichtterminal-Ersetzung ist ein →Szenario der kompositionalen Sprachentwicklung. Hierbei wird der Inhalt eines →Nichtterminals durch den Inhalt eines anderen Nichtterminals einer anderen Sprache ersetzt.

Nichtterminal-Erweiterung
Die Nichtterminal-Erweiterung ist ein →Szenario der kompositionalen Sprachentwicklung. Bei ihr besteht das Ziel darin, ein gegebenes Nichtterminal um den Inhalt eines anderen Nichtterminals einer anderen Sprache zu erweitern.

Parser
Der Parser ist Teil eines →Compilers und überführt den durch den →Lexer produzierten Strom von →Token in einen →abstrakten Syntaxbaum.

Pattern
Ein Pattern charakterisiert eine →Tokenklasse in Form von regulären Ausdrücken.

Produktion
Die Produktionen einer →Grammatik sind die Ableitungsvorschriften für die →Nichtterminale.

Programmiersprache
Eine Programmiersprache ist eine berechnungsuniverselle Sprache, die nicht speziell für eine →Domäne entwickelt wurde. Sie enthält daher im Gegensatz zu →domänenspezifischen Sprachen nur allgemeine Konzepte, anhand derer Domänenspezifika nachgebildet werden können (z.B. durch Klassenbildung in objektorientierten Sprachen).

Scope
Der Scope eines →Entries ist der Bereich im →Modell, in dem das Element des Entries durch die Verwendung seines Namens referenziert werden kann, also sichtbar und nicht verdeckt ist.

Semantik
Die Semantik beschreibt die Bedeutung der Sprache, wobei bekannte Ansätze zur Definition denotationale, operationale und translationale Semantiken sind. Dabei verwendet die denotationale Semantik mathematische Konstrukte und erklärt damit die Bedeutung der Sprachelemente, die operationale Semantik verwendet abstrakte Maschinen und erklärt das Verhalten anhand von Zustandsübergängen. Die translationale Semantik hingegen verwendet Transformationen auf eine wohlbekannte Sprache.

Sorte
In jeder Sprache existieren Entitäten verschiedener Sorten, für die ein →Symboltabelleneintrag angelegt wird (z.B. Klassen, Methoden und Variablen in Klassendiagrammen oder Zustände in Statecharts).

Sprachaggregation
Sprachaggregation ist ein →Mechanismus zur kompositionalen Sprachentwicklung. Hierbei werden vormals eigenständige Werkzeuge für verschiedene Sprachen zu einem Gesamtwerkzeug verschmolzen.

Sprachdatei
In der Sprachdatei wird hauptsächlich die →Einbettung einer Sprache in eine andere definiert. Sie bildet somit eine Ergänzung zur →Grammatik.

Sprachbestandteile
Die in dieser Arbeit behandelten Sprachbestandteile sind →konkrete und →abstrakte Syntax, →Symboltabellen, →Kontextbedingungen und →sprachspezifische Werkzeuge. Weitere Bestandteile sind →Semantik und →Generatoren.

Sprachspezifisches Werkzeug
Als sprachspezifisches Werkzeug werden Editoren und ähnliche Werkzeuge bezeichnet, die eine auf die Sprache abgestimmte Funktionalität haben.

Sprachvereinigung
Die Sprachvereinigung ist ein →Szenario der kompositionalen Sprachentwicklung. Hierbei wird davon ausgegangen, dass mindestens zwei vorhandene Sprachen und die dazugehörige Infrastruktur miteinander kombiniert werden, um anschließend ein Werkzeug zu erhalten, das alle Eingangssprachen unterstützt.

Symboltabelle
Eine Symboltabelle ist eine meist hierarchisch gegliederte Datenstruktur zum Speichern und Auflösen von Bezeichnern in einer Sprache. Kernaufgabe besteht im Auflösen von Namen mit dem Ziel, weitere Informationen wie Typ oder Signatur zu diesem Namen zu erhalten.

Symboltabelleneintrag
siehe →Entry.

Syntaktische Analyse
In der syntaktischen Analyse werden die vom →Lexer produzierten →Token durch den →Parser in einen →abstrakten Syntaxbaum überführt.

Szenario
Ein Szenario beschreibt die Intention des Sprachentwicklers bei der Kombination zweier Sprachen. Hierbei wird zwischen →Sprachvereinigung, →Nichtterminal-Erweiterung, →Nichtterminal-Ersetzung, →Einbettung einer einzelnen Sprache, →Einbettung von Sprachalternativen und der →Entwicklung einer erweiterbaren Sprache unterschieden. Zur Realisierung werden die →Mechanismen kompositionaler Sprachentwicklung in unterschiedlichen Kombinationen angewandt.

Terminal
Ein Terminal ist ein atomarer Teil einer →Grammatik und kann daher nicht weiter zerlegt werden.

Token
Die Token sind die Elemente einer →Tokenklasse.

Tokenklasse
Eine Tokenklasse ist eine Menge von Worten, die durch ein spezifisch für die Tokenklasse gültiges →Pattern beschrieben werden.

Visitor

Ein Visitor ist ein grundlegendes Design-Pattern gemäß [GHJV95]. Er wird eingesetzt, um baumartige Strukturen entlang ihres hierarchischen Aufbaus zu traversieren.

Anhang B.

Zeichenerklärungen

In den verschiedenen Kapiteln dieser Arbeit wurden unterschiedliche Notationen in den Abbildungen verwendet. Hierbei wurden soweit möglich, die Diagrammarten der UML verwendet und die entsprechenden Abbildungen mit Markern versehen, welche die verwendete Diagrammart anzeigen. Der generelle Aufbau der Abbildungen ergibt sich dabei wie in Abbildung B.1 dargestellt.

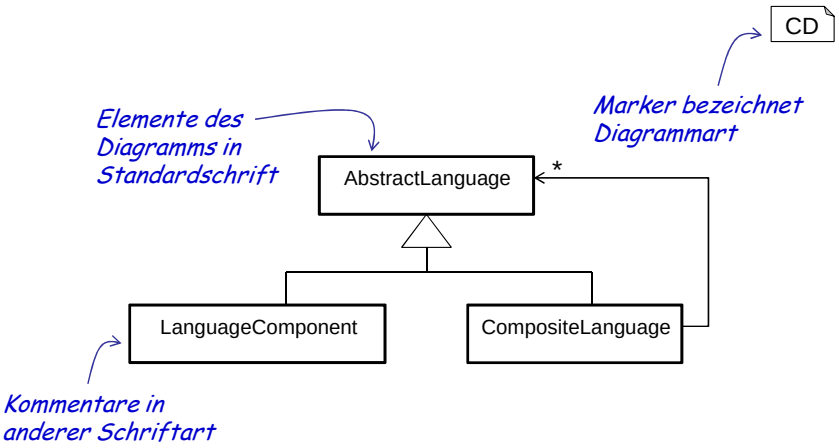


Abbildung B.1.: Aufbau von Abbildungen

Die verwendeten Diagrammarten in dieser Arbeit sind[1]:

Inhalt des Markers	Erklärung
CD	Klassendiagramme
CpD	Komponentendiagramme
OD	Objektdiagramme
SC	Statecharts
SeqD	Sequenzdiagramme

Tabelle B.2.: Verwendete Diagrammarten

[1] Für nähere Erläuterungen der Diagrammarten wird auf [OMG10c] und [OMG10b] verwiesen.

Zusätzlich wurden die unterschiedlichen Elemente innerhalb der Abbildungen, aber auch innerhalb von Code- und Grammatikbeispielen mit Stereotypen versehen. Diese Stereotypen kennzeichnen die Rolle des entsprechenden Elementes innerhalb der Komposition. Es wird zwischen folgenden Rollen unterschieden:

Markierung	Beispiele für markierte Elemente	Bedeutung
«mc»	Klassen in Klassendiagrammen	Basisklassen des MontiCore-Frameworks. Diese sind einmalig vorgegeben und werden für die Erstellung einer neuen Sprache lediglich wiederverwendet.
«component»	Klassen in Klassendiagrammen, Grammatiken, Sprachdateien	Elemente, die für die Entwicklung einer neuen Sprache erstellt bzw. aus einer einzelnen Sprachdefinition generiert wurden. Diese Elemente sind unabhängig von einer späteren Sprachkombination.
«glue»	Klassen in Klassendiagrammen, Grammatiken, Sprachdateien	Elemente, die für die Verbindung verschiedener Sprachen entwickelt bzw. zum Zwecke der Sprachkombination generiert wurden. Diese sind sowohl abhängig von den konkreten Sprachen als auch von der Sprachkombination.

Tabelle B.3.: Notationselemente zur Unterscheidung der Rollen der Artefakte

Daneben wird in vielen Diagrammen und Codefragmenten unterschieden, ob das dargestellte Artefakt vom Entwickler handgeschrieben («hw» - handwritten) ist oder generiert («gen») wurde. Da generierter Code generell nicht modifiziert wird, ist eine eindeutige Unterscheidung immer möglich.

Anhang C.

Abkürzungsverzeichnis

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
ASF	Algebraic Specification Formalism
AST	Abstract Syntax Tree
BNF	Backus-Naur-Form
DSL	Domain Specific Language (Domänenspezifische Sprache)
DSML	Domain Specific Modeling Language
EBNF	Extended Backus-Naur-Form
EMF	Eclipse Modeling Framework
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
GPL	General Purpose Language
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IMP	IDE Meta-tooling Platform
IPSEN	Integrated Project Support ENvironment
JDK	Java Development Kit
LOC	Lines Of Code
MC	MontiCore
MOF	Meta Object Facility
MPS	Meta Programming System
OAW	OpenArchitectureWare
OCL	Object Constraint Language
OMG	Object Management Group
PROGRES	PROgrammed Graph REwriting Systems
PSG	Programming System Generator
SDF	Syntax Definition Formalism
SQL	Structured Query Language
UML	Unified Modeling Language
UML/P	Unified Modeling Language / Programmier-geeignet

Anhang D.

Essenzen der gemeinsamen Obergrammatiken der Sprachen der Fallstudien

D.1. Literals

MontiCore-Grammatik

```
package mc.literals;

grammar Literals {

    options {
        parser lookahead=3
        lexer lookahead=3
        nostring noident nows
    }

    interface Literal;
    interface NumericLiteral extends Literal;

    NullLiteral implements Literal =
        "null";

    BooleanLiteral implements Literal =
        source:["true" | "false"];

    CharLiteral implements Literal =
        source:Char;

    StringLiteral implements Literal =
        source:String;

    IntLiteral implements NumericLiteral =
        source:Num_Int;

    LongLiteral implements NumericLiteral =
        source:Num_Long;

    FloatLiteral implements NumericLiteral =
        source:Num_Float;
```

```
34      DoubleLiteral implements NumericLiteral =
35         source:Num_Double;
36
37
38      token Name
39          options{testLiterals=true;} =
40          ('a'..'z' | 'A'..'Z' | '_' | '$')
41          ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$')*;
42
43      token Char =
44          '\'' ( ESC | ~('\'' | '\n' | '\r' | '\\') ) '\'';
45
46      token String =
47          '"' (ESC | ~('"' | '\\' | '\n' | '\r'))* '"';
48
49      protected token Hex_Digit =
50          ('0'..'9'|'A'..'F'|'a'..'f');
51
52      protected token Float_Suffix =
53          'f'|'F'|'d'|'D';
54
55      protected token Decimal_Exponent =
56          ('e'|'E') ('+'|'-')? ('0'..'9')+;
57
58      protected token Hex_Exponent =
59          ('p'|'P') ('+'|'-')? ('0'..'9')+;
60
61      token Numbers
62          options{testLiterals=true;}
63          {boolean isDecimal=false, isHex=false, isDoubleDot=false;} =
64          ('.'
65            (
66              ('.')
67              |
68              ('.' '.')
69              |
70              (
71                ('0'..'9')+ (Decimal_Exponent)? (t=Float_Suffix)?
72                {
73                  if (t != null && t.getText().toUpperCase().
74                  indexOf('F')>=0){
75                    _ttype = Num_Float;
76                  }else{
77                    _ttype = Num_Double;
78                  }
79                }
80              )?
81            )
82          )
83          |
84
85
86
```

```
87          (
88            ('0' {isDecimal = true; _ttype = Num_Int;}
89              (
90                (('x'|'X') ((Hex_Digit)+ | ('.' (Hex_Digit)+)
91                  | Hex_Exponent)) =>
92                (
93                  ('x'|'X')
94                  (options{warnWhenFollowAmbig=false;}: Hex_Digit)*
95                  {isHex = true;}
96                )
97                |
98                (('0'..'9')+ ('.'|Decimal_Exponent|Float_Suffix)) =>
99                  ('0'..'9')+
100               |
101               ('0'..'7')+
102             )?
103           )
104           |
105           (('1'..'9') ('0'..'9')*  {isDecimal=true; _ttype = Num_Int;})
106         )
107         {
108           if (LA(1)=='.' && LA(2)=='.'){
109             isDoubleDot = true;
110           }else{
111             isDoubleDot = false;
112           }
113         }
114         (
115           ('l'|'L') {_ttype = Num_Long;}
116           |
117           {isDecimal && !isDoubleDot}?
118           (
119             {isHex}?
120             (
121               '.' (Hex_Digit)* Hex_Exponent (t=Float_Suffix)?
122               |
123               Hex_Exponent (t=Float_Suffix)?
124               |
125               t=Float_Suffix
126             )
127             |
128             (
129               '.' ('0'..'9')* (Decimal_Exponent)? (t=Float_Suffix)?
130               |
131               Decimal_Exponent (t=Float_Suffix)?
132               |
133               t=Float_Suffix
134             )
135           )
136           {
137             if (t != null && t.getText().toUpperCase().indexOf('F') >= 0){
138               _ttype = Num_Float;
139             }else{
```

```
140                 _ttype = Num_Double;
141               }
142             }
143         )?;
144     protected token ESC =
145         '\\'
146         (
147           'n' | 'r' | 't' | 'b' | 'f' | '"' | '\'' | '\\' | ('u')+
148           Hex_Digit Hex_Digit Hex_Digit Hex_Digit | '0'..'3'
149           (
150             options{warnWhenFollowAmbig=false;} : '0'..'7'
151             (options{warnWhenFollowAmbig=false;} : '0'..'7')?
152           )?
153           |
154           '4'..'7' (options{warnWhenFollowAmbig=false;} : '0'..'7')?
155         );
156
157     token WS =
158         (
159           ' ' | '\t' | '\f'
160           |
161           (
162             options{generateAmbigWarnings=false;}:
163             "\r\n" | '\r' | '\n'
164           )
165           {newline();}
166         )+
167         {_ttype = Token.SKIP;};
168
169 }
```

D.2. Types

MontiCore-Grammatik «hw»

```
1  package mc.types;
2
3  grammar Types extends mc.literals.Literals {
4
5      concept antlr {
6          parser java {
7              public int ltCounter = 0;
8          }
9      }
10
11     interface Type extends TypeArgument, ReturnType;
12     interface ReferenceType;
13     interface TypeArgument;
14     interface ReturnType;
15
16
```

```
17      QualifiedName =
18          parts:Name (options{greedy=true;}: "." parts:Name)*;
19
20
21      ComplexArrayType: ArrayType implements Type returns Type
22      {int ltLevel = ltCounter;} =
23          ret=ComplexType
24          (
25            {ltCounter==ltLevel}?
26            (
27              astscript{!(ComponentType=ret;);}
28              (
29                options{greedy=true;}: "[" "]"
30                {a.setDimensions(a.getDimensions()+1);}
31              )+
32            )
33            |
34          );
35
36      PrimitiveArrayType: ArrayType implements (PrimitiveType "[")=>Type
37      returns Type =
38          ret=PrimitiveType
39          (
40            options{greedy=true;}:
41            astscript{!(ComponentType=ret;);}
42            (
43              options{greedy=true;}: "[" "]"
44              {a.setDimensions(a.getDimensions()+1);}
45            )+
46          )?;
47
48      VoidType implements ReturnType =
49          "void";
50
51      BooleanType:PrimitiveType =
52          primitive: ["boolean"];
53
54      IntegralType:PrimitiveType =
55          primitive: ["byte" | "short" | "int" | "long" | "char"];
56
57      FloatingPointType:PrimitiveType =
58          primitive: ["float" | "double"];
59
60      NumericType:PrimitiveType =
61          ret=FloatingPointType | ret=IntegralType;
62
63      PrimitiveType implements Type =
64          ret=FloatingPointType | ret=IntegralType | ret=BooleanType;
65
66      ComplexType returns Type =
67          ret=SimpleReferenceType
68          (options{greedy=true;}: "." ret=QualifiedType->[ret])*;
69
```

```
70      SimpleReferenceType implements ReferenceType =
71          Name
72          (options{greedy=true;}: "." Name)*
73          (options{greedy=true;}: TypeArguments)?;
74      QualifiedType [qualification:Type] =
75          Name
76          (options{greedy=true;}: TypeArguments)?;
77
78      TypeArguments {int currentLtLevel = 0;} =
79          {currentLtLevel = ltCounter;}
80          (
81            "<" {ltCounter++;}
82            typeArguments:TypeArgument
83            (options{greedy=true;}:
84              {inputState.guessing !=0 || ltCounter == currentLtLevel + 1}?
85                  "," typeArguments:TypeArgument
86            )*
87            (
88              options{generateAmbigWarnings=false;}:
89              ">" {ltCounter-=1;}
90              | ">>" {ltCounter-=2;}
91              | ">>>" {ltCounter-=3;}
92            )?
93          )
94          {(currentLtLevel != 0) || ltCounter == currentLtLevel}?;
95
96      WildcardType implements TypeArgument =
97          "?" (
98                options{greedy=true;}:
99                ("extends" upperBound:Type) | ("super" lowerBound:Type)
100             )?;
101
102     TypeParameters {int currentLtLevel = 0;} =
103         {currentLtLevel = ltCounter;}
104         (
105           options{greedy=true;}:
106           "<" {ltCounter++;}
107           typeVariableDeclarations:TypeVariableDeclaration
108           (
109             options{greedy=true;}:
110             "," typeVariableDeclarations:TypeVariableDeclaration
111           )*
112           (
113             options{generateAmbigWarnings=false;}:
114             ">" {ltCounter-=1;}
115             | ">>" {ltCounter-=2;}
116             | ">>>" {ltCounter-=3;}
117           )?
118         )
119         {(currentLtLevel != 0) || ltCounter == currentLtLevel}?
120         | ;
121
122
```

```
123     TypeVariableDeclaration =
124         Name
125         (
126           options{generateAmbigWarnings=false;}:
127           "extends" upperBounds:ComplexType
128           ("&" upperBounds:ComplexType)*
129         )?;
130 }
```

D.3. Commons

MontiCore-Grammatik «hw»

```
1  package mc.umlp.common;
2
3  grammar Common extends mc.types.Types {
4
5      options {
6          parser lookahead=3
7          lexer lookahead=7
8      }
9
10     external Invariant;
11
12     Stereotype =
13         "<<" values:StereoValue ("," values:StereoValue)* ">>";
14
15     StereoValue =
16         Name ("=" source:String)?;
17
18     Cardinality =
19         "["
20         (
21           many:["*"] {a.setLowerBound(0);a.setUpperBound(0);}
22           |
23           lowerBoundLit:IntLiteral
24           {
25             a.setLowerBound(a.getLowerBoundLit().getValue());
26             a.setUpperBound(a.getLowerBound());
27           }
28           (
29             ".."
30             (
31               upperBoundLit:IntLiteral
32               ({a.setUpperBound(a.getUpperBoundLit().getValue());})
33               |
34               noUpperLimit:["*"] {a.setUpperBound(0);}
35             )
36           )?
37         )
38         "]";
```

```
39      Completeness =
40          complete:[COMPLETE:"(c)"]
41            | ("(" incomplete:[INCOMPLETE:"..."] ")")
42            | incomplete:[INCOMPLETE:"(...,...)"]
43            | complete:[COMPLETE:"(c,c)"]
44            | incomplete:[INCOMPLETE:"(..., ...)"]
45            | complete:[COMPLETE:"(c, c)"]
46            | rightComplete:[RIGHTCOMPLETE:"(...,c)"]
47            | leftComplete:[LEFTCOMPLETE:"(c,...)"]
48            | rightComplete:[RIGHTCOMPLETE:"(..., c)"]
49            | leftComplete:[LEFTCOMPLETE:"(c, ...)"];
50
51      Modifier =
52          Stereotype?
53          (   Public:["public"]       | Public:[PUBLIC:"+"]
54            | Private:["private"]     | Private:[PRIVATE:"-"]
55            | Protected:["protected"] | Protected:[PROTECTED:"#"]
56            | Final:["final"]
57            | Abstract:["abstract"]
58            | Local:["local"]
59            | Derived:["derived"]     | Derived:[DERIVED:"/"]
60            | Readonly:["readonly"]   | Readonly:[READONLY:"?"]
61            | Static:["static"]
62          )*;
63
64      Invar =
65          (kind:Name ":")?
66          "[" content:Invariant(parameter kind) "]";
67
68  }
```

Anhang E.

Essenz der Statechart-Grammatik

MontiCore-Grammatik «hw»

```
1
2  package mc.umlp.sc;
3
4  grammar SC extends mc.umlp.common.Common {
5
6      options {
7          compilationunit SCDefinition
8      }
9
10     abstract / SCElement;
11     abstract / SCEvent;
12
13
14     external Statement;
15     external Expression;
16
17     SCDefinition =
18         Completeness?
19         Stereotype?
20         "statechart" Name
21         ((QualifiedName "(")=> SCMethod | className:ReferenceType)?
22         "{"
23           (
24             (Completeness? SCModifier "state")=>
25             sCStates:SCState
26             |
27             (Stereotype? QualifiedName "->")=>
28             sCTransitions:SCTransition
29             |
30             SCCode
31           )*
32         "}";
33
34
35     SCMethod =
36         name:QualifiedName "(" (
37           sCParameters:SCParameter ("," sCParameters:SCParameter)*
38         )? ")";
39
40
```

```
41  SCParameter =
42      Type Name;
43
44  SCAction =
45      (((Name ":") | "["=> preCondition:Invar)?
46      (
47        ";"
48        |
49        ("/" Statement
50          (((Name ":") | "["=> postCondition:Invar ";")?)
51      );
52
53  SCDoAction =
54      "do" SCAction;
55
56  SCEntryAction =
57      "entry" SCAction;
58
59  SCExitAction =
60      "exit" SCAction;
61
62  SCModifier =
63      Stereotype?
64      (Initial:["initial"] | Final:["final"] | Local:["local"])*;
65
66
67  SCState astextends SCElement =
68      Completeness?
69      SCModifier
70      "state" Name
71      (
72        ("{"
73          (((Name ":") | "[" )=> Invar ";")?
74          SCEntryAction?
75          SCDoAction?
76          SCExitAction?
77          (
78            (Completeness? SCModifier "state")=>
79            sCStates:SCState
80            |
81            (Stereotype? QualifiedName "->")=>
82            sCTransitions:SCTransition
83            |
84            ("code")=>
85            SCCode
86            |
87            sCInternTransitions:SCInternTransition
88          )*
89        "}")
90        |
91        ";"
92      );
93
```

```
94      SCInternTransition =
95          Stereotype? "->" ":"? SCTransitionBody;
96
97
98      SCTransition astextends SCElement =
99          Stereotype?
100         sourceName:QualifiedName "->" targetName:QualifiedName
101         (
102           (":" SCTransitionBody)
103           |
104           ";"
105         );
106
107
108     SCTransitionBody =
109         (((Name ":") | "["=> preCondition:Invar)?
110         SCEvent?
111         (
112           ("/" Statement
113               (((Name ":") | "["=> postCondition:Invar ";")?)
114           |
115           ";"
116         );
117
118
119     SCMethodOrExceptionCall extends SCEvent =
120         name:QualifiedName (("("=> SCArguments)?;
121
122
123     SCReturnStatement extends SCEvent =
124         "return"
125         (("("=> "("
126           (("..."=> incomplete:[INCOMPLETE:"..."]
127           | Expression)
128         ")")?;
129
130
131     SCArguments =
132         ("(" "..."=> ("(" incomplete:[INCOMPLETE:"..."] ")")
133         | ("(" ")"=> ("(" ")")
134         | ("(" expressions:Expression ("," expressions:Expression)* ")");
135
136
137     SCCode astextends SCElement =
138         "code" Statement;
139
140 }
```

Anhang F.

Essenz der OCL-Grammatik

MontiCore-Grammatik «hw»

```
1  package mc.umlp.ocl;
2
3  grammar OCL extends mc.umlp.common.Common {
4
5      options {
6          compilationunit OCLFile
7          follow OCLEmbedment "]";
8      }
9
10
11     OCLEmbedment =
12         OCLDefinition;
13
14
15     interface OCLDefinition;
16
17
18     OCLFile =
19         Stereotype? prefix:Name Name "{"
20           OCLConstraints:OCLConstraint*
21         "}"
22         EOF;
23
24
25     OCLConstraint implements (Stereotype? "context")=>OCLDefinition =
26         Stereotype?
27         OCLRawConstraint;
28
29
30     interface OCLRawConstraint;
31
32
33     OCLOperationConstraint implements
34      ("context" Stereotype? OCLOperationSignature)=>OCLRawConstraint =
35         "context"
36         Stereotype?
37         OCLOperationSignature
38         OCLLetDeclaration?
39         OCLPreStatement?
40         OCLPostStatement?;
```

```
41  interface OCLOperationSignature;
42
43
44  OCLInvariant implements OCLRawConstraint =
45      OCLClassContext?
46      "inv" Name?
47      OCLParameters? ":"
48      (statements:OCLExpression ";")+;
49
50
51  OCLClassContext =
52      (context:["context"] | Import:["import"])
53      contextDefinitions:OCLContextDefinition
54      ("," contextDefinitions:OCLContextDefinition)*;
55
56
57  OCLContextDefinition =
58      className:QualifiedName Name?;
59
60
61  OCLLetDeclaration =
62      "let" (declarations:OCLDeclaration ";")+;
63
64
65  OCLPreStatement =
66      "pre" Name? ":"
67      (statements:OCLExpression ";")+;
68
69
70  OCLPostStatement =
71      "post" Name? ":"
72      (statements:OCLExpression ";")+;
73
74
75  OCLMethodSignature implements OCLOperationSignature =
76      (
77        options{greedy=true;}:
78        TypeParameters
79      )?
80      ((ReturnType QualifiedName "(")=> ReturnType |)
81      methodName:QualifiedName
82      OCLParameters
83      OCLThrowsClause?;
84
85
86  OCLConstructorSignature implements OCLOperationSignature =
87      "new" ReferenceType
88      OCLParameters
89      OCLThrowsClause?;
90
91
92
93
```

```
94      OCLParameters =
95          "("
96          (
97            params:OCLParameterDeclaration
98            ("," params:OCLParameterDeclaration)*
99          )?
100         ")";
101
102
103     OCLParameterDeclaration =
104         Type Name;
105
106
107     OCLThrowsClause =
108         "throws" throwables:ComplexType
109         ("," throwables:ComplexType)*;
110
111
112     interface OCLComprehensionExpr;
113
114
115     OCLComprehensionVarDeclaratorStyle
116       implements (OCLCollectionVarDeclaration "|")
117       =>OCLComprehensionExpr =
118         generator:OCLCollectionVarDeclaration "|"
119         comprehensionItems:OCLComprehensionItem
120         ("," comprehensionItems:OCLComprehensionItem)*;
121
122
123     OCLComprehensionExpressionStyle
124       implements (OCLNonGreedyEquivalentExpr "|")
125       =>OCLComprehensionExpr =
126         expression:OCLNonGreedyEquivalentExpr "|"
127         comprehensionItems:OCLComprehensionItem
128         ("," comprehensionItems:OCLComprehensionItem)*;
129
130
131     OCLComprehensionEnumerationStyle implements OCLComprehensionExpr =
132         (
133           collectionItems:OCLCollectionItem
134           ("," collectionItems:OCLCollectionItem)*
135         )?;
136
137
138     interface OCLComprehensionItem;
139
140
141     OCLCollectionItem =
142         expression:OCLNonGreedyEquivalentExpr
143         (".." extension:OCLNonGreedyEquivalentExpr)?;
144
145
146
```

```
147   interface OCLDeclaration;
148
149
150   OCLVariableDeclaration
151     implements (Type? Name "=")=>OCLComprehensionItem,
152                (Type? Name "=")=>OCLDeclaration =
153       Type?
154       Name
155       "=" OCLExpression;
156
157
158   OCLMethodDeclaration implements OCLDeclaration =
159       ReturnType?
160       Name
161       OCLParameters
162       "=" OCLExpression;
163
164
165   OCLCollectionVarDeclaration
166     implements ((Name "in")|(Type Name))=>OCLComprehensionItem =
167       (
168         Type
169         Name
170         ("in" expression:OCLNonGreedyEquivalentExpr)?
171       )
172       |
173       (
174         Name
175         "in" expression:OCLNonGreedyEquivalentExpr
176       );
177
178  interface OCLExpression extends
179     OCLDefinition, OCLComprehensionItem;
180
181
182   OCLIfThenElseExpr implements OCLExpression =
183       "if" condition:OCLExpression
184       "then" thenExpression:OCLExpression
185       "else" elseExpression:OCLExpression;
186
187
188
189   OCLTypeIfExpr implements OCLExpression =
190       "typeif" unknownType:Name "instanceof" Type
191       "then" thenExpression:OCLExpression
192       "else" elseExpression:OCLExpression;
193
194
195   OCLForallExpr implements OCLExpression =
196       "forall"
197       declarations:OCLCollectionVarDeclaration
198       ("," declarations:OCLCollectionVarDeclaration)*
199       ":" OCLExpression;
```

```
200   OCLExistsExpr implements OCLExpression =
201       "exists"
202       declarations:OCLCollectionVarDeclaration
203       ("," declarations:OCLCollectionVarDeclaration)*
204       ":" OCLExpression;
205
206
207    OCLAnyExpr implements OCLExpression =
208       "any" OCLExpression;
209
210
211    OCLLetinExpr implements OCLExpression =
212       "let" (declarations:OCLDeclaration ";")+
213       "in" OCLExpression;
214
215
216    OCLIterateExpr implements OCLExpression =
217       "iterate" "{"
218       iterationDeclarator:OCLCollectionVarDeclaration ";"
219       initDeclarator:OCLDeclaration ":"
220       accumulatorName:Name "=" accumulatorValue:OCLExpression
221       "}";
222
223
224    OCLConditionalExpr
225      implements (("?" OCLConditionalExpr ":" OCLConditionalExpr)?
226      )=>OCLExpression returns OCLExpression =
227        ret=OCLEquivalentExpr
228        (
229          astscript{!(Condition=ret;);}
230          "?" thenExpression:OCLConditionalExpr
231          ":" elseExpression:OCLConditionalExpr
232        )?;
233
234
235    OCLNonGreedyEquivalentExpr:OCLInfixExpression
236    returns OCLExpression =
237        (
238          init {comprehensionStack.push(Greedy.OFF);}:
239          ret=OCLImpliesExpr
240          (
241            astscript{!(leftHand=ret;);}
242            operator:["<=>"] rightHand:OCLImpliesExpr
243          )*
244          (init {comprehensionStack.pop();}:)
245        );
246
247
248
249
250
251
252
```

```
253     OCLEquivalentExpr:OCLInfixExpression returns OCLExpression =
254         (
255           init {comprehensionStack.push(Greedy.ON);}:
256           ret=OCLImpliesExpr
257           (
258             astscript{!(leftHand=ret;);}
259             operator:["<=>"] rightHand:OCLImpliesExpr
260           )*
261           (init {comprehensionStack.pop();}:)
262         );
263
264
265     OCLImpliesExpr:OCLInfixExpression returns OCLExpression =
266         ret=OCLDoubleLogicalORExpr
267         (
268           astscript{!(leftHand=ret;);}
269           operator:["implies"] rightHand:OCLDoubleLogicalORExpr
270         )*;
271
272     OCLDoubleLogicalORExpr:OCLInfixExpression returns OCLExpression =
273         ret=OCLDoubleLogicalANDExpr
274         (
275           astscript{!(leftHand=ret;);}
276           operator:["||"] rightHand:OCLDoubleLogicalANDExpr
277         )*;
278
279     OCLDoubleLogicalANDExpr:OCLInfixExpression returns OCLExpression =
280         ret=OCLSingleLogicalORExpr
281         (
282           astscript{!(leftHand=ret;);}
283           operator:["&&"] rightHand:OCLSingleLogicalORExpr
284         )*;
285
286     OCLSingleLogicalORExpr:OCLInfixExpression returns OCLExpression =
287         {comprehensionStack.peek().equals(Greedy.OFF)}?
288         (
289           ret=OCLLogicalXORExpr
290           (
291              options{greedy=false;}:
292              astscript{!(leftHand=ret;);}
293              operator:["|"] rightHand:OCLLogicalXORExpr
294           )*
295         )
296         |
297         (
298           ret=OCLLogicalXORExpr
299           (
300             options{greedy=true;}:
301             astscript{!(leftHand=ret;);}
302             operator:["|"] rightHand:OCLLogicalXORExpr
303           )*
304         );
305
```

```
306     OCLLogicalXORExpr:OCLInfixExpression returns OCLExpression =
307         ret=OCLSingleLogicalANDExpr
308         (
309           astscript{!(leftHand=ret;);}
310           operator:["^"] rightHand:OCLSingleLogicalANDExpr
311         )*;
312
313     OCLSingleLogicalANDExpr:OCLInfixExpression returns OCLExpression =
314         ret=OCLRelationalExpr
315         (
316           astscript{!(leftHand=ret;);}
317           operator:["&"] rightHand:OCLRelationalExpr
318         )*;
319
320     OCLRelationalExpr:OCLInfixExpression returns OCLExpression =
321         ret=OCLElemInExpr
322         (
323           astscript{!(leftHand=ret;);}
324           operator:["==" | "!="] rightHand:OCLElemInExpr
325         )?;
326
327     OCLElemInExpr:OCLInfixExpression returns OCLExpression =
328         ret=OCLCompareExpr
329         (
330           (
331             astscript{!(leftHand=ret;);}
332             operator:["isin"] rightHand:OCLCompareExpr
333           )*
334         |
335           ret=OCLInstanceofExpr->[ret]
336         );
337
338     OCLInstanceofExpr implements
339     OCLExpression [expression:OCLExpression] =
340         "instanceof" Type;
341
342     OCLCompareExpr:OCLInfixExpression returns OCLExpression =
343         ret=OCLShiftExpr
344         (
345           astscript{!(leftHand=ret;);}
346           operator:["<" | "<=" | ">" | ">="] rightHand:OCLShiftExpr
347         )?;
348
349
350     OCLShiftExpr:OCLInfixExpression returns OCLExpression =
351         ret=OCLBinaryPlusMinusExpr
352         (
353           astscript{!(leftHand=ret;);}
354           operator:["<<" | ">>" | ">>>"] rightHand:OCLBinaryPlusMinusExpr
355         )*;
356
357
358
```

```
359 OCLBinaryPlusMinusExpr:OCLInfixExpression returns OCLExpression =
360     ret=OCLBinaryMultDivModExpr
361     (
362       astscript{!(leftHand=ret;);}
363       operator:["+" | "-"] rightHand:OCLBinaryMultDivModExpr
364     )*;
365
366
367 OCLBinaryMultDivModExpr:OCLInfixExpression returns OCLExpression =
368     ret=OCLPrefixExpression
369     (
370       astscript{!(leftHand=ret;);}
371       operator:["*" | "/" | "%"] rightHand:OCLPrefixExpression
372     )*;
373
374
375 OCLPrefixExpression
376    implements ("-" | "+" | "~" | "!"
377        | ("(" Type ")" OCLPrefixExpression)=>
378         "(" Type ")" OCLPrefixExpression | OCLPrimary)
379         =>OCLExpression returns OCLExpression =
380     (
381       (
382         astscript{!();}
383         operator:["-" | "+" | "~" | "!"]
384         expression:OCLPrefixExpression
385       )
386       |
387       ("(" Type ")" OCLPrefixExpression)=>
388       ret=OCLTypeCastExpr
389       |
390       ret=OCLPrimary
391     );
392
393
394 OCLTypeCastExpr  =
395     "(" Type ")" expression:OCLPrefixExpression;
396
397
398
399 interface OCLPrimary;
400
401
402 OCLParenthizedExpr implements OCLPrimary =
403     "(" OCLExpression ")" ("." qualification:OCLPrimary)?;
404
405
406 OCLComprehensionPrimary implements
407 (Name? ( "<" Type ">")? "{")=>OCLPrimary
408  {boolean validCollectionType=false;} =
409     (
410       type=Name
411       {
```

```
412             if (type != null) {
413               if (type.equals("Set")) {
414                 a.setSet(true);
415                 validCollectionType = true;
416               }
417               else if (type.equals("List")) {
418                 a.setList(true);
419                 validCollectionType = true;
420               }
421               else if (type.equals("Collection")) {
422                 a.setCollection(true);
423                 validCollectionType = true;
424               }
425             }
426           }
427         )?
428         {type == null || validCollectionType == true}?
429         (
430            "<" referenceParameter:Type ">"
431         )?
432         "{" expression:OCLComprehensionExpr "}"
433
434         ("." qualification:OCLPrimary)?;
435
436
437     OCLQualifiedPrimary implements OCLPrimary =
438         (
439           this:["this"]
440           |
441           super:["super"]
442           |
443           res:["result"]
444           |
445           prefixIdentifier:Name
446         )
447         ("." qualifications:Name)*
448         (
449           postfixQualification:OCLQualification
450           ("." OCLQualifiedPrimary)?
451         )?;
452
453     interface OCLQualification;
454
455
456     OCLArrayQualification implements OCLQualification =
457         ("[" arguments:OCLExpression "]")+;
458
459
460     OCLArgumentQualification implements OCLQualification =
461         "(" (arguments:OCLExpression
462         ("," arguments:OCLExpression)*)? ")";
463
464
```

```
465     OCLPostfixQualification implements OCLQualification =
466         (atpre:["@pre"] | transitive:["**"])
467         OCLQualification?;
468
469
470     OCLIsnewPrimary implements OCLPrimary =
471         "isnew" "(" OCLExpression ")";
472
473
474
475     OCLDefinedPrimary implements OCLPrimary =
476         "defined" "(" OCLExpression ")";
477
478
479     OCLLiteral implements OCLPrimary =
480         value:Literal;
481
482
483
484     concept antlr {
485         parser java {
486           public enum Greedy { ON, OFF };
487           public java.util.Stack<Greedy> comprehensionStack
488               = new java.util.Stack<Greedy>();
489         }
490     }
491
492 }
```

Anhang G.

Essenz der Java-Grammatik

MontiCore-Grammatik «hw»

```
1  package mc.javadsl;
2
3  grammar JavaDSL extends mc.types.Types {
4
5      options {
6          lexer lookahead=4
7          keywords {"."}
8      }
9
10
11     CompilationUnit =
12         (((Annotation)* "package")=> PackageDeclaration)?
13         (ImportDeclarations:ImportDeclaration)*
14         (TypeDeclarations:TypeDeclaration | ";" )* ;
15
16
17     PackageDeclaration =
18        (Modifiers:Annotation)*
19        "package"
20        Name ( "." Name )* ";" ;
21
22
23     ImportDeclaration =
24         "import" (IsStatic:["static"])?
25         Name (options{greedy=true;}: "." Name)*
26         ("." IsOnDemand:["*"])?  ";"   ;
27
28
29     interface TypeDeclaration extends
30         (TypeDeclaration)=>AnnotationMemberDeclaration,
31         (TypeDeclaration)=>MemberDeclaration;
32
33
34
35     VariableDeclaration =
36         (Modifiers:Modifier)*
37         Type:Type
38         Declarators:VariableDeclarator
39            (options{warnWhenFollowAmbig=false;}: ","
40            Declarators:VariableDeclarator)*;
```

```
41      VariableDeclarationStatement
42      implements (VariableDeclaration ";")=>Statement =
43          Declaration:VariableDeclaration ";";
44
45
46      VariableDeclarationExpression implements
47      (VariableDeclaration)=>Expression =
48          Declaration:VariableDeclaration;
49
50
51      interface Modifier;
52
53
54      PrimitiveModifier implements Modifier =
55          Modifier:["private" | "public" | "protected" | "static" |
56          "transient" | "final" | "abstract" | "native" |
57          "threadsafe" | "synchronized" | "const" | "volatile" |
58          "strictfp"]
59      ;
60
61
62      EnumDeclaration implements (Modifier* "enum")=>TypeDeclaration =
63          (Modifiers:Modifier)* "enum" Name
64          ("implements" ImplementedTypes:Type
65          ("," ImplementedTypes:Type)* )?
66          "{"
67            (EnumConstantDeclarations:EnumConstantDeclaration
68            (options{greedy=true;}:
69              "," EnumConstantDeclarations:EnumConstantDeclaration)*)?
70            (",")?
71            (";" (MemberDeclarations:MemberDeclaration | ";")* )?
72          "}" ;
73
74
75      EnumConstantDeclaration =
76          (Annoations:Annotation)* Name
77          (
78            "("
79              (Arguments: Expression ("," Arguments: Expression)* )?
80             ")"
81          )?
82          (
83            "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}"
84            {a.getMemberDeclarations().set_Existent(true);}
85          )? ;
86
87
88      AnnotationTypeDeclaration implements
89      (Modifier* "@")=>TypeDeclaration =
90          (Modifiers:Modifier)* "@" "interface" Name
91          "{"
92            (MemberDeclarations:AnnotationMemberDeclaration | ";")*
93          "}" ;
```

```
94      interface AnnotationMemberDeclaration;
95      ast AnnotationMemberDeclaration astextends MemberDeclaration;
96
97      AnnotationMethodDeclaration implements
98      ((Modifier)* ReturnType Name "(" ")")=>AnnotationMemberDeclaration =
99          (Modifiers:Modifier)* ReturnType Name "(" ")"
100         ("default" DefaultValue:AnnotationMemberValue )? ";" ;
101
102     interface Annotation extends Modifier;
103     ast Annotation astextends Expression, Modifier;
104
105     MappedMemberAnnotation implements
106     ("@" Name ("." Name)* "(" AnnotationMemberValuePair)=>Annotation =
107         "@" Name (options{greedy=true;}:  "." Name)*
108         "("
109           MemberValues:AnnotationMemberValuePair
110           ("," MemberValues:AnnotationMemberValuePair)*
111         ")" ;
112
113
114     SingleMemberAnnotation implements
115     ("@" Name ("." Name)* "(" AnnotationMemberValue)=>Annotation =
116         "@" Name (options{greedy=true;}: "." Name)*
117         "(" MemberValue:AnnotationMemberValue ")" ;
118
119
120     MarkerAnnotation implements Annotation =
121         "@" Name (options{greedy=true;}:  "." Name)*
122         ("(" ")" )?  ;
123
124
125     AnnotationMemberValuePair =
126         MemberName:Name "=" Value:AnnotationMemberValue  ;
127
128     AnnotationMemberValue returns Expression =
129         ret = Annotation
130         | ret = ConditionalExpression
131         | ret = AnnotationMemberArrayInitializer  ;
132
133
134     AnnotationMemberArrayInitializer implements
135     ("{" (AnnotationMemberValue ("," AnnotationMemberValue)*)?
136     (",")? "}")=>Expression =
137         "{"
138           (
139             Initializers:AnnotationMemberValue
140             (
141               options{warnWhenFollowAmbig=false;}:
142               "," Initializers:AnnotationMemberValue
143             )*
144           )?
145           (",")?
146         "}";
```

```
147     ClassDeclaration implements
148     ((Modifiers:Modifier)* "class")=>TypeDeclaration =
149         (Modifiers:Modifier)* "class" Name
150         TypeParameters
151         ("extends" ExtendedClass:Type)?
152         ("implements" ImplementedInterfaces:Type
153         ("," ImplementedInterfaces:Type)*)?
154         "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}"  ;
155
156
157     InterfaceDeclaration
158     implements (Modifier* "interface")=>TypeDeclaration =
159         (Modifiers:Modifier)* "interface" Name
160         TypeParameters
161         ("extends" ExtendedInterfaces:Type
162         ("," ExtendedInterfaces:Type)* )?
163         "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}"  ;
164
165
166     interface MemberDeclaration;
167
168
169     TypeInitializer implements
170     (("static")? BlockStatement)=>MemberDeclaration =
171         (IsStatic:["static"])?
172         Body:BlockStatement ;
173
174
175     interface MethodOrConstructorDeclaration;
176
177     ast MethodOrConstructorDeclaration astextends MemberDeclaration =
178         Name
179         Throwables:QualifiedName*
180         Modifiers:Modifier*
181         TypeParameters
182         Parameters:ParameterDeclaration*
183         Block:BlockStatement   ;
184
185
186     ConstructorDeclaration implements
187     ((Modifier)* TypeParameters Name "(")=>MemberDeclaration =
188         (Modifiers:Modifier)*
189         TypeParameters
190         Name
191         "("
192           (Parameters:ParameterDeclaration
193           ("," Parameters:ParameterDeclaration)*)?
194         ")"
195         (
196           "throws" Throwables:QualifiedName
197           ("," Throwables:QualifiedName)*
198         )?
199         Block:BlockStatement ;
```

```
200     MethodDeclaration implements
201     ((Modifier)* TypeParameters ReturnType Name "("=>MemberDeclaration =
202         (Modifiers:Modifier)*
203         TypeParameters
204         ReturnType Name
205         "("
206           (Parameters:ParameterDeclaration
207           ("," Parameters:ParameterDeclaration)*)?
208         ")"
209         (
210           options{greedy=true;}: "[" "]"
211           {a.setAdditionReturnTypeDimensions(
212               a.getAdditionReturnTypeDimensions()+1);}
213         )*
214         (
215           "throws" Throwables:QualifiedName
216           ("," Throwables:QualifiedName)*
217         )?
218         ( ";" | Block:BlockStatement ) ;
219
220
221     Methods =
222         MethodDeclaration+;
223
224     FieldDeclaration
225      implements ((Modifier)* Type VariableDeclarator)=>MemberDeclaration,
226      ((Modifier)* Type VariableDeclarator)=>AnnotationMemberDeclaration =
227         Declaration:VariableDeclaration ";" ;
228
229
230     VariableDeclarator =
231         Name
232         (
233           options{greedy=true;}: "[" "]"
234           {a.setAdditionalArrayDimensions
235             (a.getAdditionalArrayDimensions()+1);}
236         )*
237         ("=" Initializer:Initializer)?  ;
238
239
240     ArrayInitializer implements Expression =
241         "{"
242         (
243           Initializers:Initializer
244           (options{warnWhenFollowAmbig=false;}:
245           "," Initializers:Initializer)*
246         )?
247         (",")?
248         "}"  ;
249
250
251     Initializer returns Expression =
252         ret=Expression ;
```

```
253     ParameterDeclaration =
254         (Modifiers:Modifier)* Type:Type
255         (Ellipsis:["..."])?
256         Name
257         (
258           options{greedy=true;}: "[" "]"
259           {a.setAdditionalArrayDimensions
260           (a.getAdditionalArrayDimensions()+1);}
261         )* ;
262
263     BlockStatement implements ("{")=>Statement =
264         "{" (Statements:Statement)* "}" ;
265
266     interface Statement;
267
268     ExpressionStatement implements (Expression ";")=>Statement =
269         Expression:Expression ";" ;
270
271     TypeDeclarationStatement implements Statement =
272         TypeDeclaration:TypeDeclaration ;
273
274     EmptyStatement implements Statement = ";" ;
275
276     ContinueStatement implements Statement =
277         "continue" (Label:Name)? ";"  ;
278
279     SwitchDefaultStatement astimplements Statement =
280         "default"  ;
281
282     BreakStatement implements Statement =
283         "break" (Label:Name)? ";"  ;
284
285     ReturnStatement implements Statement =
286         "return" (Expression:Expression)? ";"  ;
287
288     SwitchStatement implements Statement  =
289         "switch" "(" Expression:Expression ")"
290         "{"
291         (
292           (
293             options{greedy=true;}:
294             (Statements:CaseStatement |
295             Statements:SwitchDefaultStatement) ":"
296           )+
297           (Statements:Statement)*
298         )*
299         "}"  ;
300
301     CaseStatement astimplements Statement =
302         "case" Expression:Expression  ;
303
304
305
```

```
306     ThrowStatement implements Statement =
307         "throw" Expression:Expression ";"  ;
308
309
310     AssertStatement implements Statement =
311         "assert" Assertion:Expression (":" Message:Expression)? ";"  ;
312
313
314     WhileStatement implements Statement =
315         "while" "(" Condition:Expression ")" Statement:Statement ;
316
317
318     DoWhileStatement implements Statement =
319         "do" Statement:Statement "while"
320         "(" Condition:Expression ")" ";"  ;
321
322
323     ForEachStatement implements
324     ("for" "(" ParameterDeclaration ":")=>Statement =
325         "for" "("
326           VariableDeclaration:ParameterDeclaration ":"
327           Iterable:Expression
328         ")" Statement:Statement  ;
329
330
331     ForStatement implements Statement =
332         "for" "("
333           (
334             (VariableDeclaration)=>
335                Initializations:VariableDeclarationExpression
336             |
337             Initializations:Expression ("," Initializations:Expression)*
338           )?
339           ";" (Condition:Expression)?
340           ";" (Updates:Expression ("," Updates:Expression)*)?
341         ")" Statement:Statement  ;
342
343
344
345     IfStatement implements Statement =
346         "if" "(" Condition:Expression ")" SuccessStatement:Statement
347         (options{greedy=true;}: "else" OptionalElseStatement:Statement)?;
348
349
350
351     LabeledStatement implements Statement =
352         Label:Name ":" Statement:Statement  ;
353
354
355     SynchonizedStatement implements Statement =
356         "synchronized" "(" Expression:Expression ")" Block:BlockStatement;
357
358
```

```
359     TryStatement implements Statement =
360         "try" Block:BlockStatement
361         (CatchClause:CatchClause)*
362         (FinallyClause:FinallyClause)?  ;
363
364
365     FinallyClause =
366         "finally" Block:BlockStatement  ;
367
368
369     CatchClause =
370         "catch" "(" ExceptionVariable:ParameterDeclaration ")"
371         Block:BlockStatement ;
372
373     interface Expression;
374
375      AssignmentExpression implements (ConditionalExpression
376     (("="|"+="|"-="|"*="|"/="|"%="|">>="|">>>="|"<<="|"&="|"^="|"|=")
377      AssignmentExpression)?)=>Expression returns Expression =
378         ret=ConditionalExpression
379         (
380           astscript{!(LeftHand=ret;);}
381           AssignmentOperator:["=" | "+=" | "-=" | "*=" | "/="  | "%="
382               | ">>=" | ">>>=" | "<<=" | "&=" | "^=" | "|="]
383           RightHand:AssignmentExpression
384         )? ;
385
386     ConditionalExpression implements (LogicalOrExpression)=>Expression
387       returns Expression =
388         ret=LogicalOrExpression
389         (
390           astscript{!(Condition=ret;);}
391           "?" IfTrue:AssignmentExpression
392           ":" IfFalse:ConditionalExpression
393         )?  ;
394
395
396     LogicalOrExpression:InfixExpression implements
397     (LogicalAndExpression)=>Expression returns Expression =
398         ret=LogicalAndExpression
399         (
400           astscript{!(LeftOperand=ret;);}
401           Operator:["||"] RightOperand:LogicalAndExpression
402         )* ;
403
404
405     LogicalAndExpression:InfixExpression implements
406     (BitWiseOrExpression)=>Expression returns Expression =
407         ret=BitWiseOrExpression
408         (
409           astscript{!(LeftOperand=ret;);}
410           Operator:["&&" ] RightOperand:BitWiseOrExpression
411         )* ;
```

```
412     BitWiseOrExpression:InfixExpression
413       implements (XorExpression)=>Expression returns Expression =
414         ret=XorExpression
415         (
416           astscript{!(LeftOperand=ret;);}
417           Operator:["|" ] RightOperand:XorExpression
418         )* ;
419
420
421     XorExpression:InfixExpression
422       implements (BitWiseAndExpression)=>Expression returns Expression =
423         ret=BitWiseAndExpression
424         (
425           astscript{!(LeftOperand=ret;);}
426           Operator:["^"] RightOperand:BitWiseAndExpression
427         )* ;
428
429
430     BitWiseAndExpression:InfixExpression
431       implements (EqualityExpression)=>Expression returns Expression =
432         ret=EqualityExpression
433         (
434           astscript{!(LeftOperand=ret;);}
435           Operator:["&" ] RightOperand:EqualityExpression
436         )* ;
437
438
439     EqualityExpression:InfixExpression
440       implements (RelationalExpression)=>Expression returns Expression =
441         ret=RelationalExpression
442         (
443           astscript{!(LeftOperand=ret;);}
444           Operator:["!=" | "=="] RightOperand:RelationalExpression
445         )* ;
446
447
448     RelationalExpression:InfixExpression
449       implements (ShiftExpression)=>Expression returns Expression =
450         ret=ShiftExpression
451         (
452           (
453             astscript{!(LeftOperand=ret;);}
454             Operator:["<" | ">" | "<=" | ">="]
455             RightOperand:ShiftExpression
456           )*
457           |
458           ret=InstanceOfExpression->[ret]
459         ) ;
460
461
462     InstanceOfExpression implements
463     ("instanceof" Type)=>Expression [Expression:Expression] =
464         "instanceof" Type:Type ;
```

```
465     ShiftExpression:InfixExpression
466       implements (AdditiveExpression)=>Expression returns Expression =
467         ret=AdditiveExpression
468         (
469           astscript{!(LeftOperand=ret;);}
470           Operator:["<<" | ">>" | ">>>"] RightOperand:AdditiveExpression
471         )* ;
472
473
474     AdditiveExpression:InfixExpression implements
475     (MultiplicativeExpression)=>Expression returns Expression =
476         ret=MultiplicativeExpression
477         (
478           astscript{!(LeftOperand=ret;);}
479           Operator:["+" | "-"] RightOperand:MultiplicativeExpression
480         )*  ;
481
482
483     MultiplicativeExpression:InfixExpression
484       implements (PrefixExpression)=>Expression returns Expression =
485         ret = PrefixExpression
486         (
487           astscript { !(LeftOperand=ret;);}
488           Operator:["*"|"/" | "%"] RightOperand:PrefixExpression
489         )* ;
490
491
492     PrefixExpression implements
493     (("++" | "--" | "-" | "+" | "~" | "!") | CastExpression)=>Expression
494       returns Expression =
495         (
496           ( astscript{!();}
497             Operator: ["++"  | "--" | "-" | "+" |  "~" | "!"]
498             Expression:PrefixExpression
499           )
500           |
501           ret=CastExpression
502         );
503
504
505     CastExpression implements (
506     ("(" PrimitiveArrayType ")")=> ("(" PrimitiveArrayType ")")
507     |
508     ("(" ComplexArrayType ")" CastExpression)=>
509     ("(" ComplexArrayType ")" CastExpression)
510     |
511     PostfixExpression)=>Expression returns Expression =
512         ("(" PrimitiveArrayType ")" )=>
513         ( astscript{!();}
514           "(" TargetType:PrimitiveArrayType ")"
515           Expression:PrefixExpression
516         )
517         |
```

```
518         ( "(" ComplexArrayType ")" CastExpression)=>
519         ( astscript{!();}
520           "(" TargetType:ComplexArrayType ")" Expression:CastExpression
521         )
522         | ret=PostfixExpression
523     ;
524
525
526     PostfixExpression
527       implements (Primary ("++"|"--")?)=>Expression returns Expression =
528         ret=Primary
529         (astscript { !(Expression=ret;); } Operator:["++"|"--"] )? ;
530
531     ConstructorInvocation implements
532       (TypeArguments? (ThisReference->[null] | SuperReference->[null])
533       "(" (Expression ("," Expression)*)? ")" )=>Expression =
534         (TypeArguments:TypeArguments )?
535         (
536           Reference:ThisReference->[null]
537           | Reference:SuperReference->[null]
538         )
539         "("
540           (Arguments: Expression ("," Arguments: Expression)*)?
541         ")" ;
542
543
544     QualifiedConstructorInvocation: ConstructorInvocation
545     implements (ConstructorInvocation "." )=>
546     Expression [qualification=Expression] =
547         (TypeArguments:TypeArguments )?
548         Reference:SuperReference->[qualification]
549         "("
550           (Arguments: Expression ("," Arguments: Expression)*)?
551         ")" ;
552
553     Primary returns Expression =
554         ret=PrimarySuffix
555         (
556           "." ret=ThisReference->[ret]
557           |
558           "."
559           (
560             ((TypeArguments)? MethodInvocationWithSingleName->
561                  [null,null])=>
562             (typeArguments=TypeArguments)?
563             ret=MethodInvocationWithSingleName->[ret, typeArguments]
564             |
565             (FieldAccess->[null])=>
566             ret=FieldAccess->[ret]
567             |
568             (QualifiedConstructorInvocation->[null])=>
569             ret=QualifiedConstructorInvocation->[ret]
570             |
```

```
571             superReference=SuperReference->[ret]  "."
572             (
573               ((TypeArguments)? Name "("=>
574               (typeArguments=TypeArguments)?
575               ret=MethodInvocationWithSingleName->
576                   [superReference, typeArguments]
577               |
578               ret=FieldAccess->[superReference]
579             )
580             |
581             ret=NewExpression->[ret]
582           )
583           |
584           ret=ArrayAccessExpression->[ret]
585         )*;
586
587
588     ThisReference implements ("this")=>
589     Expression [Qualification:Expression] =
590         "this" ;
591
592
593     FieldAccess implements (Name)=>
594     Expression [Qualification:Expression]=
595         Name;
596
597
598     MethodInvocationWithSingleName:MethodInvocation
599     implements (Name "("=>Expression
600       [Qualification:Expression, TypeArguments:TypeArguments] =
601         Name
602         "("
603           (Arguments: Expression ("," Arguments: Expression)*)?
604         ")" ;
605
606
607     MethodInvocationWithQualifiedName:MethodInvocation
608     implements (Name (options{greedy=true;}: "." Name)* "("=>Expression
609       [ Qualification:Expression, TypeArguments:TypeArguments] =
610         Name
611         (options{greedy=true;}: "." Name)*
612         "("
613           (Arguments: Expression ("," Arguments: Expression)*)?
614         ")" ;
615
616
617     ArrayAccessExpression
618     implements ("[" Expression "]")=>Expression [Receiver:Expression] =
619         "[" Component:Expression "]" ;
620
621
622
623
```

```
624     PrimarySuffix returns Expression =
625         ((Type|VoidType) "." "class")=> ret=ClassAccess
626         | (MethodInvocationWithQualifiedName->[null, null])=>
627           (ret=MethodInvocationWithQualifiedName->[null, null])
628         | ret=QualifiedName
629         | ret=Literal
630         | ret=NewExpression->[null]
631         | (ConstructorInvocation)=>
632           ret=ConstructorInvocation
633         | ret=ThisReference->[null]
634         | ret=SuperReference->[null]
635         | ret=ParenthesizedExpression;
636
637
638     ParenthesizedExpression implements
639     ("(" AssignmentExpression ")")=>Expression =
640         "(" Expression:AssignmentExpression ")" ;
641
642
643     ClassAccess implements ((Type|VoidType) "." "class")=>Expression =
644         (Type|VoidType) "." "class" ;
645
646
647     SuperReference implements
648     ("super")=>Expression [Qualification:Expression] =
649         "super";
650
651     NewExpression [qualification=Expression] returns Expression =
652         "new" (typeArgs=TypeArguments)?
653         (type=ComplexType | type=PrimitiveType)
654         (
655           ret=ClassInstantiation->[qualification, typeArgs, type]
656           | ret=ArrayInstantiation->[qualification, typeArgs, type]
657         ) ;
658
659     interface Instantiation;
660
661     ClassInstantiation
662     [Qualification:Expression, TypeArguments:TypeArguments, Type:Type] =
663         "("
664           (ConstructorArguments: Expression
665           ("," ConstructorArguments: Expression)*)?
666         ")"
667         (
668           options{greedy=true;}:
669           "{"
670             (MemberDeclarations:MemberDeclaration | ";")*
671           "}"
672           {a.getMemberDeclarations().set_Existent(true);}
673         )?
674     ;
675
676
```

```
677     ArrayInstantiation
678     [Qualification:Expression, TypeArguments:TypeArguments, Type:Type] =
679         (
680           (options{greedy=true;}: "["(Sizes:Expression) "]")+
681           (options{greedy=true;}: "[" "]" {a.setDims(a.getDims()+1);})*
682           |
683           ("[" "]" {a.setDims(a.getDims()+1);})+
684           Initializer:ArrayInitializer
685         ) ;
686
687
688     QualifiedName implements Expression =
689         parts:Name (options{greedy=true;}: "." parts:Name)*;
690
691
692     interface Literal extends Expression;
693
694     NullLiteral implements Literal;
695
696     BooleanLiteral implements Literal;
697
698     CharLiteral implements Literal;
699
700     StringLiteral implements Literal;
701
702     interface NumericLiteral extends Literal;
703
704     IntLiteral implements NumericLiteral;
705
706     LongLiteral implements NumericLiteral;
707
708     FloatLiteral implements NumericLiteral;
709
710     DoubleLiteral implements NumericLiteral;
711
712
713
714     concept attributes {
715         syn JavaTypeEntry: /mc.javadsl.ets.entries.JavaTypeEntry;
716         global SymbolTable: /interfaces2.helper.SymbolTableInterface;
717     }
718
719
720     concept antlr {
721         lexer java {
722           private boolean assertEnabled = true;
723           private boolean enumEnabled = true;
724           public void enableAssert() { assertEnabled = true; }
725           public void disableAssert() { assertEnabled = false; }
726           public boolean isAssertEnabled() { return assertEnabled; }
727           public void enableEnum() { enumEnabled = true; }
728           public void disableEnum() { enumEnabled = false; }
729           public boolean isEnumEnabled() { return enumEnabled; }
```

```
730             @Override
731             public int testLiteralsTable(int ttype) {
732               int ret;
733               ret = super.testLiteralsTable(ttype);
734               if (!assertEnabled
735                   && "assert".equals(
736                           new String(text.getBuffer(), 0, text.length()))) {
737                 ret = Name;
738               }
739               if (!enumEnabled
740                   && "enum".equals(
741                           new String(text.getBuffer(), 0, text.length()))) {
742                 ret = Name;
743               }
744               return ret;
745             }
746           }
747       }
748
749 }
750
```

Anhang H.

Lebenslauf

Name	Völkel
Vorname	Steven
Geburtsdatum	09.12.1978
Geburtsort	Dessau
Staatsangehörigkeit	deutsch
seit 2009	Wissenschaftlicher Mitarbeiter am Lehrstuhl Informatik 3 (Software Engineering) der RWTH Aachen
2006 - 2009	Wissenschaftlicher Mitarbeiter am Institut für Software Systems Engineering der TU Braunschweig
2006	Abschluss als Diplom-Informatiker
2001-2006	Studium der Informatik an der TU Braunschweig
1997 - 2001	Zeitsoldat bei der Bundeswehr
1997	Abitur
1985 - 1997	Polytechnische Oberschule und Gymnasium, Dessau

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.