RWTH AACHEN

SE SOFTWARE ENGINEERING

Holger Krahn

MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering

Berichte der Aachener Informatik,
Software Engineering Band 1
Hrsg.: Prof. Dr. rer. nat. Bernhard Rumpe

MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering

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

vorgelegt von

Dipl.-Inform. Holger Krahn
aus Braunschweig

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

Tag der mündlichen Prüfung: 18.12.2009

Die Druckfassung dieser Dissertation ist unter der ISBN 978-3-8322-8948-5 erschienen.

[Kra10] H. Krahn
MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering.
Shaker Verlag, ISBN 978-3-8322-8948-5.
Aachener Informatik-Berichte, Software Engineering Band 1. 2010.
www.se-rwth.de/publications

Kurzfassung

Domänenspezifische Sprachen (engl. domain specific language - DSL) sind Sprachen der Informatik, mit denen kompakte Problemlösungen aus eng umrissenen fachlichen oder technischen Anwendungsgebieten formuliert werden können. Durch die Nutzung einer fachspezifischen Notation gelingt die Integration von Experten einfacher als bei einer herkömmlichen Softwareentwicklung, weil die Modelle von ihnen besser verstanden werden. Die automatische Erzeugung von Produktivcode aus domänenspezifischen Modellen ist eine effektive Form der modellgetriebenen Entwicklung.

Die derzeitige DSL-Entwicklung erschwert aufgrund der fehlenden zentralen Sprachreferenz, die die abstrakte und konkrete Syntax umfasst, und der unzureichenden Modularisierung eine agile und effiziente Vorgehensweise. Es mangelt an Methoden und Referenzarchitekturen, um komplexe modellbasierte Werkzeuge strukturiert entwerfen und in der Softwareentwicklung einsetzen zu können.

In dieser Arbeit wird daher die DSL-Entwicklung mit dem MontiCore-Framework beschrieben, das die modulare Entwicklung von textuellen DSLs und darauf basierten Werkzeugen erlaubt. Die wichtigsten wissenschaftlichen Beiträge lassen sich wie folgt zusammenfassen:

- Die Definition von textuellen domänenspezifischen Sprachen wird durch ein kompaktes grammatikbasiertes Format ermöglicht, das sowohl die abstrakte als auch die konkrete Syntax einer Sprache definiert und so als zentrale Dokumentation für die Entwickler und Nutzer einer DSL fungiert. Die entstehende abstrakte Syntax entspricht etablierten Metamodellierungs-Technologien und erweitert übliche grammatikbasierte Ansätze.

- Die Wiederverwendung von Teilsprachen innerhalb der modellgetriebenen Entwicklung wird durch zwei Modularitätsmechanismen vereinfacht, da so existierende Artefakte auf eine strukturierte Art und Weise in neuen Sprachdefinitionen eingesetzt werden können. Grammatikvererbung erlaubt die Spezialisierung und Erweiterung von Sprachen. Einbettung ermöglicht die flexible Kombination mehrerer Sprachen, die sich auch in ihrer lexikalischen Struktur grundlegend unterscheiden können.

- Das MontiCore-Grammatikformat ist modular erweiterbar, so dass zusätzliche Informationen als so genannte Konzepte spezifiziert werden können und darauf aufbauend weitere Infrastruktur aus der Sprachdefinition generiert werden kann. Die Erweiterungsfähigkeit wird durch Konzepte zur deklarativen Spezifikation von Links in der abstrakten Syntax und durch ein Attributgrammatiksystem demonstriert.

- Die Entwicklung von Codegeneratoren und Analysewerkzeugen für DSLs wird durch die Bereitstellung einer Referenzarchitektur entscheidend vereinfacht, so dass bewährte Lösungen ohne zusätzlichen Entwicklungsaufwand genutzt werden können. Die Qualität der entstehenden Werkzeuge wird damit im Vergleich zu existierenden Ansätzen gehoben.

- Es wird demonstriert, wie compilierbare Templates ein integriertes Refactoring der Templates und einer Laufzeitumgebung ermöglichen. Darauf aufbauend wird eine Methodik definiert, die aus exemplarischem Quellcode schrittweise einen Generator entwickelt, dessen Datenmodell automatisiert abgeleitet werden kann.

Die beschriebenen Sprachen und Methoden sind innerhalb des Frameworks MontiCore implementiert. Ihre Anwendbarkeit wird durch die Entwicklung des MontiCore-Frameworks im Bootstrapping-Verfahren und durch zwei weitere Fallstudien demonstriert.

Abstract

Domain specific languages (DSLs) are languages in computer science which permit specifying compact solutions in clear-cut functional or technical application areas. Using a domain specific notation simplifies the integration of experts in comparison to conventional software development because the models are easier understood by them. The automatic creation of production code from domain specific models is an effective form of model-driven development.

An agile and efficient development process is hard to establish using existing DSL development methods because of the missing central language reference which includes abstract and concrete syntax and inadequate modularization techniques. Methods and reference architectures are lacking for designing and using complex and model-based tools in structured way for software development.

Thus, in this thesis the modular development of textual DSLs and tools with the MontiCore-framework is described. The most important contributions to research can be summarized as follows:

- Textual domain specific languages can be defined by a compact grammar-based format that defines abstract syntax as well as concrete syntax of a language and can therefore be used as a central documentation for developers and users of a DSL. The emerging abstract syntax is equivalent to well-established metamodeling-techniques and extends common grammar-based approaches.

- The reuse of language fragments within model-driven development is supported by two modularity mechanisms that combine existing artifacts in a structured way to form new languages: Grammar inheritance supports the specialization and extension of languages. Embedding permits the flexible combination of multiple languages, which can also differ fundamentally in their lexical structure.

- The used grammar format is extensible in a modular way such that additional information can be specified as so-called concepts. Based on them, further infrastructure can be generated from the language definition. The extensibility by concepts is demonstrated by a declarative way to specify links in the abstract syntax and by an attribute grammar system.

- The development of code generators and tools for the analysis of DSLs is simplified considerably by making a reference architecture available. Approved solutions can be used without further development effort. Thus, the quality of the emerging tools is increased in comparison to existing approaches.

- It is demonstrated how compilable templates can be used for an integrated refactoring of templates and a runtime environment. Based on that, a method is defined to develop a generator in a stepwise manner from existing source code. The data model of the generator can be derived automatically.

The abovementioned languages and methods are developed within the framework MontiCore. Their applicability is demonstrated by the development of the framework itself in a bootstrapping-process and by two further case studies.

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr. Bernhard Rumpe für die Betreuung dieser Dissertation bedanken. Durch zahlreiche konstruktive Diskussionen und wertvolle Ratschläge hat er entscheidend zum Gelingen der Promotion beigetragen. Insbesondere möchte ich ihm für die Möglichkeit, neben der akademischen Arbeit auch immer wieder Einblicke in die Praxis erlangen zu können, und für ein abwechselungsreiches und spannendes Arbeitsumfeld danken.

Weiterer Dank gebührt Prof. Dr. Andy Schürr für die Bereitschaft, die Dissertation ebenfalls zu betreuen, durch konstruktive Anmerkungen mich zum Nachdenken anzuregen und neue Perspektiven aufzuzeigen.

Herrn Prof. Dr.-Ing. Stefan Kowalewski möchte ich für die Leitung der Promotionskommision danken und Herrn Prof. Dr. Wolfgang Thomas für die Bereitschaft, mich zu prüfen.

Die Entwicklung des MontiCore-Frameworks ist keine Einzelleistung, sondern in Zusammenarbeit mit Kollegen und Studierenden am Lehrstuhl entstanden. Für die konstante Diskussion und die Rückmeldungen, für den nötigen sportlichen Ausgleich, für die hoffentlich weiterhin erfolgreiche und spannende Arbeit am Quellcode und an Veröffentlichungen und das gemeinsame Feiern möchte ich mich besonders bei den folgenden Personen bedanken: Christian Berger, Marita Breuer, Michael Dukaczewski, Christoph Ficek, Diego Gaviola, Tim Gülke, Arne Haber, Christoph Herrmann, Conrad Hoffmann, Christian Jordan, Björn Mahler, Ingo Maier, Fabian May, Miguel Paulos Nunes, Claas Pinkernell, Dirk Reiß, Jan-Oliver Ringert, Henning Sahlbach, Martin Schindler, Mark Stein, Jens Stephani, Jens Theeß, Christoph Torens, Galina Volkova, Ingo Weisemöller und Gerald Winter.

Besonders hervorzuheben ist Steven Völkel für das Führen von fachlichen Diskussionen und die Zusammenarbeit am Quellcode und an Papieren über MontiCore, die stetige Motivation und seine pragmatische Herangehensweise, die mir oft eine Hilfe bei der Lösung von Problemen war, sowie seine kritische Auseinandersetzung mit dieser Ausarbeitung.

Ebenfalls besonders hilfreich war die Unterstützung von Hans Grönniger. Vielen Dank für die Zusammenarbeit an den gemeinsamen Projekten und Papieren, die intensive Beschäftigung mit der vorliegenden Arbeit und die interessanten Einblicke in die formalen Seiten des Software Engineerings.

Zusätzlich möchte ich Holger Rendel für Kommentare zu Teilen dieser Arbeit danken und für seinen Enthusiasmus bei der Entwicklung von MontiCore.

Danken möchte ich auch Arne, Cecile, Daniela, Edgar, Gerrit, Jens, Jörg, Karolina, Marcus, Melanie, Jens, Ralf, Sabine und Sabrina für die spannenden und entspannenden Momente neben der Arbeit und die Abwechslung, die einem immer wieder neue Perspektiven aufzeigt.

Meinen Eltern möchte ich für die Unterstützung auf meinem Lebensweg und die Hilfe in schwierigen Phasen danken. Meinem Bruder Oliver gebührt Dank für die aufmunternden Worte und Taten sowie sein stetes Vorbild. Ihm wünsche ich alles Gute, damit bald die nächste Doktorfeier folgen kann.

Der größte Dank geht an meine Frau Angelika und meine Tochter Johanna. Angelika möchte ich für ihren aufopfernden Einsatz für unsere Familie danken, der mir den notwendigen Freiraum für diese Arbeit verschafft hat. Sie war stets eine konstruktive und kritische Leserin der Arbeit und fleißige Lektorin, abwechselnd eine fordernde Antreiberin und eine verständnisvolle Zuhörerin. Sie hat so einen bedeutenden Teil zur Fertigstellung dieser Dissertation beigetragen. Johanna danke ich für ihre freundliche Art, die mich mit einem einzigen Lächeln auch aus dem tiefsten Tal befreien kann.

Inhaltsverzeichnis

Teil I

Grundlagen

Kapitel 1

Einführung

Domänenspezifische Sprachen (engl. domain specific language - DSL) sind Sprachen der Informatik, mit denen kompakte Problemlösungen aus eng umrissenen fachlichen oder technischen Anwendungsgebieten formuliert werden können. Durch die Anlehnung an ein Fachgebiet gelingt die Integration von Fachexperten besser als bei einer herkömmlichen Softwareentwicklung. Die automatische Erzeugung von Produktivcode aus DSL-Modellen stellt eine Form der modellgetriebenen Entwicklung dar, die eine direkte Erstellung eines Softwaresystems aus Modellen ermöglicht. In dieser Arbeit werden textuelle DSLs mittels einer grammatikbasierten modularen Sprachdefinition beschrieben, die durch das MontiCore-Framework in ausführbare Komponenten überführt werden können. Dabei können Methoden eingesetzt werden, die die Definition einer DSL ausgehend von anderen Formaten ermöglichen. Die Ausgestaltung eines Generators wird durch die Verwendung eines Entwicklerframeworks erleichtert, das eine Referenzarchitektur für modellbasierte Codegeneratoren darstellt.

Dieses erste Kapitel dient als eine grobe Übersicht über diese Arbeit. Zunächst wird die Verwendung von DSLs innerhalb der Softwareentwicklung motiviert. Daraufhin werden verbreitete Definitionen des DSL-Begriffs zusammengestellt und zu einer eigenen Definition entwickelt. Darauf aufbauend werden die Erfolgsfaktoren und Risiken des Einsatzes von DSLs dargestellt. Es wird beschrieben, wie solche Sprachen und darauf basierende Werkzeuge derzeit entwickelt werden und welche Probleme dabei identifiziert wurden. Daraus wird ein Vorgehen zur Spezifikation von domänenspezifischen Sprachen mit einem kompakten grammatikbasierten Format entwickelt, deren Verarbeitung durch ein Entwickler-Framework unterstützt wird. Schließlich werden die wesentlichen Ziele des Forschungsvorhabens und der Beitrag zur Forschung aufgelistet sowie ein Überblick über diese Arbeit gegeben.

1.1 Motivation

Die Entwicklung von Softwaresystemen hat sich in den letzten 40 Jahren zu einer eigenständigen Ingenieursdisziplin herausgebildet. Namensgebend hierfür war die Nato-Konferenz im Jahre 1968 [NR69], auf der der Begriff „Software Engineering“ erstmals verwendet wurde, auch wenn die entscheidenden Werkzeuge, Techniken und Methoden erst in den folgenden Jahren entwickelt wurden.

Ein wichtiger Aspekt der Softwareentwicklung sind Programmiersprachen, in denen die Entwickler die Programmlogik formulieren und so direkt die Eigenschaften des Soft-

waresystems bestimmen. Anfänglich orientierten sich die Programmiersprachen sehr an den technischen Instruktionen der verwendeten Prozessoren. Dieses hatte zur Folge, dass die Programme für einen neuen Prozessortyp jeweils neu entwickelt werden mussten. Mit der zunehmenden industriellen Relevanz der programmierten Applikationen ging die Notwendigkeit einher, Programme plattformunabhängig zu schreiben und sie so über einen längeren Zeitraum verwenden zu können. Zunächst wurde dieses durch Verwendung von Programmiersprachen wie Pascal und C erreicht, wobei die Programme jeweils durch spezifische Compiler auf unterschiedlichen Plattformen übersetzt werden mussten. Mittlerweile hat sich vor allem die Idee einer virtuellen Maschine durchgesetzt, die einen virtuellen Befehlssatz zur Verfügung stellt, der von einem erzeugten Zwischencode genutzt wird. Die aktuell zur Entwicklung von Geschäftssoftware am häufigsten genutzten Programmiersprachen, Java und die Sprachen der .net-Plattform, verwenden diese Technik.

Ein Erfolgsfaktor moderner Programmiersprachen ist das von Parnas entwickelte Konzept des „information hiding“ [Par72]. Zentral ist dabei die Einführung von so genannten Schnittstellen, die eine Modularisierung der Software erlauben und im Gegensatz zur konkreten Implementierung nur die Informationen enthalten, die für eine Nutzung notwenig sind. Dieses Prinzip erlaubt Entwicklern Teile einer Software zu verwenden, ohne die konkrete Ausprägung verstehen zu müssen, was letztlich für eine kollaborative Arbeit unerlässlich ist. Die Einführung von Schnittstellen in Programmiersprachen ermöglicht die Entwicklung komplexer APIs und Frameworks, die wiederkehrende Programmierlogik kapseln und so den Entwickleraufwand für ein individuelles Projekt reduzieren. Die Entwickler können so bereits qualitätsgesicherte Programmteile immer wieder verwenden.

Über die Nutzung der Programmiersprache hinaus hat sich die Disziplin der Software-Modellierung gebildet. Dabei wird versucht, geeignete Modelle einzusetzen, um die wesentlichen Aspekte der Software frühzeitig zu erkennen und somit die Planung zu erleichtern. Eine solche Planung ist aufgrund der wachsenden Komplexität und wirtschaftlichen Bedeutung der Systeme notwendig. Dieses Vorgehen ist bei anderen Ingenieurstätigkeiten üblicherweise akzeptiert: Zum Beispiel werden bei der Planung eines Gebäudes auf der Grundlage von Modellen die Baukosten geschätzt und grundlegende Eigenschaften wie die Statik geprüft. Somit wird vergleichbar zur Software-Modellierung in die Planung investiert, weil so die Gesamtinvestitionskosten, die deutlich höher liegen als die Kosten für die Planung, gesichert werden sollen. Durch diese Analogie hat sich auch der Begriff Software-Architektur herausgebildet.

Industriell eingesetzte Werkzeuge zur Software-Modellierung verwenden teilweise die UML [OMG07b, OMG07a], die eine standardisierte Zusammenfassung verschiedener Modellierungstechniken ist. Ihr Erfolg ist jedoch umstritten, weil sich gerade frühe Versionen durch solch eklatante Fehler und wenig Komfort auszeichneten, dass eine Benutzung durch viele Entwickler abgelehnt wird. Solche Werkzeuge können wie in der Architektur als rein planerische Werkzeuge eingesetzt werden. Sie dienen dann als Vorlage der Implementierung und zu deren Überwachung. Diese Verwendung lässt jedoch die Möglichkeiten der Informationstechnik ungenutzt, die es nämlich prinzipiell erlaubt, Modelle direkt in laufende Software umzusetzen. Daher können Werkzeuge wie Gentleware Apollo [Gen] und IBM Rational Software Architect [RSA] auch teilweise zur Implementierung eines Softwaresystems eingesetzt werden, indem eine automatische Codeerzeugung aus den Modellen genutzt und so die UML als eine abstrakte Programmiersprache verwendet wird.

Die Object Management Group hat mit der Model Driven Architecture [OMG03] eine Initiative ins Leben gerufen, eine auf Modellen basierende Entwicklung zu etablieren. Dabei werden Modelle auf verschiedenen Abstraktionsebenen eingesetzt, die untereinander entwe-

der manuell oder automatisiert mit Hilfe von Modelltransformationen abgebildet werden. Die verschiedenen Abstraktionsebenen erschweren jedoch den Umgang mit sich wandelnden Anforderungen, weil so oftmals Änderungen nachgepflegt werden müssen. Daher greift die so genannte modellgetriebene Entwicklung den Wunsch nach einer direkten Codeerzeugung aus Modellen auf. Die Integration des generierten Quellcodes erfolgt teilweise auf Codeebene, was jedoch eher eine pragmatische Lösung als ein anzustrebendes Ideal ist. Der entscheidende Nachteil ist, dass die Entwickler die Struktur der Codegenerierung verstehen müssen und sich nicht auf das Verständnis der Modelle beschränken können.

Frameworks und APIs kapseln sich wiederholende Aspekte einer Implementierungsarbeit und stellen so eine Alternative zur modellgetriebenen Entwicklung dar. Gerade bei der Entwicklung von Geschäftssoftware sind Frameworks weit verbreitet, die auf der JavaEE-Spezifikation basieren. Das Hauptproblem bei der Verwendung von Frameworks und APIs ist jedoch, dass deren Freiheitsgrade durch die Modularität der gewählten Programmiersprache begrenzt sind, so dass wiederkehrende Funktionalität zum Beispiel nur in Form von Klassen und Methoden bereitgestellt werden kann. Mehr Freiheit erlauben da generative Ansätze, die auch sich ähnelnden Code aus Modellen erzeugen können, der nicht gekapselt werden kann. Dabei müssen die notwendigen Informationen jedoch für den Codegenerator formuliert werden.

Mit zunehmender Abstraktion einer Programmiersprache von technischen Gegebenheiten muss oftmals zwischen zwei Extremen abgewogen werden: Einerseits erleichtern abstrakte und daher kompakte Programme die Implementierung eines Softwaresystems, weil weniger Quellcode geschrieben werden muss. Dieses reduziert allein durch die Übersichtlichkeit die Anzahl der Entwicklungsfehler und verringert so die Kosten und die time-to-market. Andererseits birgt die Abstraktion das Risiko, dass die gewählte Lösung ineffizient ist oder das genau gewünschte Verhalten nicht erreichbar ist, weil für das gegebene Problem falsch abstrahiert wurde. Diese Entscheidung kann nicht für jede Domäne generalisiert werden: Für Geschäftssoftware ist teilweise die time-to-market die entscheidende Komponente, da bei schnell wechselnden Märkten lange Entwicklungszeiten hemmend für das Geschäft sind. Bei der Entwicklung automotiver Systeme sind die Stückkosten weiterhin entscheidend, weil sich die Entwicklungskosten, die zu einer kompakteren, preisgünstigeren Hardware führen, schnell amortisieren.

Die Anzahl der Anwendungsgebiete der Informatik, die auch Domänen genannt werden, steigt mit der zunehmenden Verbreitung der Informationstechnik in vielen Bereichen des alltäglichen Lebens. Die ständig steigende Leistungsfähigkeit und Kompaktheit der Hardware erlaubt dabei eine Vielzahl an neuen Anwendungen. Die in einigen Domänen bereits existierenden Notationen, um Systeme zu beschreiben, und Sprachen, die für Domänenexperten zugänglicher sind als Programmiersprachen, werden als domänenspezifische Sprachen bezeichnet. DSLs haben im Gegensatz zu allgemeinen abstrakteren Programmiersprachen weniger das Problem, dass durch die Wahl der Abstraktion eine ineffiziente Implementierung erzeugt wird, weil die Abstraktion spezifisch für die Domäne gewählt wurde und keine Allgemeingültigkeit besitzen muss. Normalerweise ist daher auch kein neues Programmiersprachenparadigma in einer DSL enthalten. Die Idee der domänenspezifischen Sprache ist in der Informatik keineswegs neu. Bereits Ende der sechziger Jahre wurde die Idee formuliert und gipfelt in [Lan66] sogar darin, dass eine Vielzahl an neuen Programmiersprachen postuliert wurde. Erfolgreich eingesetzte Sprachen wie Cobol wurden spezifisch für einen Anwendungsbereich wie die betriebswirtschaftlichen Anwendungen konzipiert und auch vor allem dort eingesetzt.

Die Vorteile der Verwendung von DSLs liegen vor allem in der engen Einbindung von Domänenexperten, die durch ihre Expertise das Softwaresystem selbst besser verstehen,

optimieren oder sogar selbständig entwickeln können [DKV00]. Dadurch wird die Produktivität der Entwickler gesteigert, die Verlässlichkeit der Systeme verbessert und die Wartung vereinfacht [DK98]. Das so formalisierte Domänenwissen kann unabhängig von einer Implementierung verwendet werden und für eine Vielzahl an Aufgaben wie Migration von technischen Plattformen, Optimierung und Wissensmanagement verwendet werden. Die Risiken bei der Verwendung von DSLs liegen vor allem bei den Kosten für den Entwurf, die Implementierung und Wartung einer DSL und den Kosten für die Schulung der Entwickler [DKV00]. Neben den Kosten ist es schwierig, den richtigen Aufgabenbereich für die DSL zu finden, um dabei einen Mehrwert für die Entwickler zu erreichen, ohne eine neue allgemeingültige Programmiersprache zu entwickeln.

Aktuellere Beispiele für DSLs kommen aus der Entwicklung von eingebetteten Systemen für Regelungssysteme. Innerhalb der Regelungstechnik haben sich Modellierungstechniken herausgebildet, die die wesentlichen Eigenschaften eines Systems bestimmen und sich als Grundlage für eine Implementierung eignen. Die Realisierung einer Regelung für diese Systeme erfordert eine fachliche Umsetzung der Modelle, wobei meistens eine Zeit- und Wert-Diskretisierung erreicht werden muss. Die DSL Simulink [Sim] der Firma MathWorks in Verbindung mit verschiedenen Codegeneratoren ermöglicht, die üblichen regelungstechnischen Diagramme direkt in Programmcode umzusetzen, wobei Entscheidungen, die bisher durch einen Programmierer getroffen wurden, nun automatisiert durch den Codegenerator erledigt werden. Dieser Einsatz von DSLs erlaubt die direkte Einbindung von Domänenexperten, hier den Regelungstechniker, in den Softwareentwicklungsprozess. Er kann somit direkt seine Lösung erstellen, validieren und optimieren. Die Codegeneratoren erzeugen nicht direkt ausführbaren Maschinencode, sondern vielmehr Quellcode einer allgemeinen Programmiersprache (engl. General Purpose Language - GPL), der dann automatisiert in Maschinencode durch übliche Compiler umgesetzt wird. Dieses Vorgehen verkürzt die Entwicklung von Codegeneratoren, weil die etablierten Compiler und deren eingebaute Optimierungen genutzt werden können, und ermöglicht gleichzeitig den für einige Anwendungsbereiche notwendigen Einsatz von zertifizierten Werkzeugen.

Mit der zunehmenden Komplexität der Software-Produkte und der damit größeren Anzahl an beteiligten Personen hat sich neben den immer ausgereifteren Programmiersprachen und Compilern eine Disziplin herausgebildet, die die Planung von Software betrifft. Dabei sind verschiedene Softwareentwicklungsprozesse entstanden, die die wesentlichen Tätigkeiten bei Anforderungserhebung, Entwurf, Erstellung und Qualitätssicherung beschreiben. Es existieren verschiedene Gegenentwürfe, die versuchen, den mit den Prozessen verbundenen Dokumentationsaufwand zu reduzieren. Die bei der Entwicklung solcher agilen Methoden beteiligten Personen haben im agilen Manifest [BBB$^+$] Werte festgehalten, die für jede Softwareentwicklung unabhängig vom eingesetzten Prozess gelten sollten: „Individuals and interactions over processes and tools, Working software over comprehensive documentation, Customer collaboration over contract negotiation, Responding to change over following a plan“. Zentral ist dabei die Forderung, dass das Produkt mit seinen Eigenschaften im Zentrum stehen soll und dass der Erfolg eines Projekts nicht durch die Einhaltung des Prozesses beurteilt werden kann. Wichtig ist ebenso das Erkennen der Notwendigkeit, auf sich ändernde Anforderungen flexibel zu reagieren.

Die agile modellgetriebene Entwicklung wird unter anderem in [Rum04a, Rum04b] beschrieben. Die Abbildung 1.1 zeigt, wie verschiedene UML-basierte Modelle und weitere Sprachen zur direkten Erzeugung und Validierung eines Softwaresystems eingesetzt werden. Dabei werden die Modelle nur auf einem Abstraktionsgrad verwendet, d.h. es gibt nicht explizite Planungsmodelle, die erstellt und dann später verworfen werden, sondern die

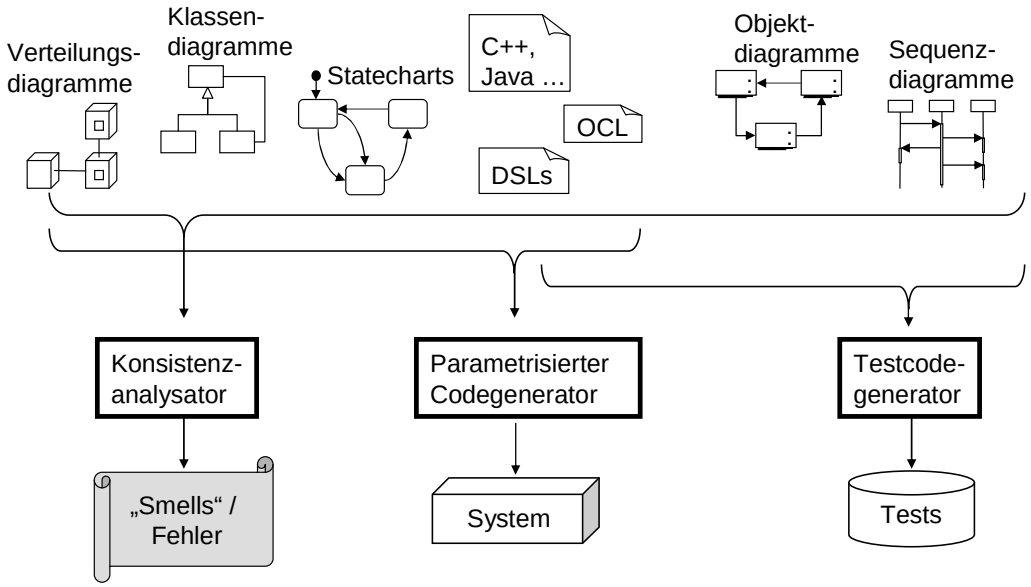


Abbildung 1.1: Agile modellgetriebene Entwicklung (adaptiert aus [Rum04a])

Modelle werden im Laufe der Entwicklung fortlaufend verfeinert, konkretisiert und an neue Anforderungen angepasst.

Diese Form der Entwicklung korrespondiert gut mit den agilen Methoden auf Quellcodeebene, weil weniger explizite Planungsdokumente existieren, sondern viele der erstellten Artefakte direkt Teil der Lösung sind. Im Sinne einer *ausführbaren Spezifikation* beeinflussen die Modelle direkt die Software. Die Modelle sind eine Form der Spezifikation, die jedoch automatisch in Code umgesetzt werden kann. Daher wird oftmals auch von „Correct-by-design“ gesprochen, weil die entstehende Software direkt aus der Spezifikation umgesetzt wird und ihr somit automatisch entspricht, sofern der Generator korrekt implementiert ist. Dadurch wird eine Verifikation der Software unnötig, aber die Validierung gewinnt an Bedeutung, insbesondere weil der Entwickler, der vorher die Modelle manuell umgesetzt hat, als eine kritisch kontrollierende Instanz entfällt. Daher kann es, wie in [Rum04a] beschrieben wird, sinnvoll sein, explizite Testmodelle zu verwenden, die den modellierten Sachverhalt aus einem anderen Blickwinkel betrachten. Durch diese neu eingeführte Redundanz lassen sich Fehler während der Entwicklungsphase entdecken.

1.2 Domänenspezifische Sprachen

Innerhalb der Literatur gibt es keine einheitlich akzeptierte Definition, was unter einer domänenspezifischen Sprache zu verstehen ist. Einige häufig zitierte Beispiele sind die folgenden Definitionen:

- „DSLs show an increased correspondence of language constructs to domain concepts when contrasted with general purpose languages [...] [and are] often computationally incomplete.“ [KPKP06]

- „A DSL is a language designed to be useful for a limited set of tasks, in contrast to general-purpose languages that are supposed to be useful for much more generic tasks, crossing multiple application domains.“ [JB06]

- „A domain-specific language is a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.“ [DKV00]

- „By focusing on a problem domain's idioms and jargon, DSLs avoid the notational *noise* required when using overly general constructs of a general-purpose language to express the same thing. Moreover, DSLs are not necessarily programming languages: they are languages tailored to express something about the solution to a problem.“ [Wil01]

Eine Extraktion der wesentlichen Aspekte aus den aufgeführten Definitionen hat die folgenden charakteristischen Eigenschaften einer DSL ergeben:

- Die Sprache orientiert sich an einer Anwendungsdomäne, für die sie hauptsächlich verwendet wird, und weniger an der technischen Realisierung.

- Falls die DSL ausführbar ist, ist die Ausdrucksmächtigkeit der Sprache soweit eingeschränkt, wie es für die Domäne ausreichend ist, was meistens weniger als die universelle Berechenbarkeit ist.

- Eine DSL erlaubt, die Problemstellungen einer Domäne kompakt zu lösen.

Die genannten Kriterien lassen sich nicht nutzen, um scharf zwischen GPLs und DSLs zu unterscheiden, zumal sich aus DSLs im Laufe der Zeit auch GPLs entwickeln können [JB06]. In [MHS03] wird dieses Argument untermauert, weil bestimmte DSLs in Bezug auf eine spezialisierte Domäne domänenspezifischer als andere sind und somit die Unterscheidung zwischen GPL und DSL nicht unbedingt eindeutig ist, sondern ein Frage des Grades der Spezifität. In Bezug auf das Anwendungsgebiet ist auch nicht eindeutig, was genau unter einer Domäne verstanden wird. Neben den fachlichen Domänen wie betriebswirtschaftliche Anwendungen oder automotive Software gibt es auch technische Domänen wie Persistenz oder Nutzerschnittstellen [KT08]. Die Domänenspezifität bedeutet dann die Eignung für eine spezifische Aufgabe. Viel wichtiger als die scharfe Unterscheidung zwischen DSLs und GPLs ist die Idee, dass durch die Beschränkung der Ausdrucksfähigkeit einer Sprache gleichzeitig eine Effizienzsteigerung in der Softwareentwicklung erreicht werden kann. Beim Entwurf einer DSL ist die Fokussierung auf das Anwendungsgebiet wichtig, das (und nur das) erfasst werden soll [Hud98]. Nur durch das gezielte Auslassen von Konzepten lässt sich eine Effizienzsteigerung gegenüber GPLs erreichen, die stets gewisse Grundkonzepte enthalten müssen, um universale Berechnungsfähigkeiten zu besitzen. DSLs hingegen müssen nicht zwangsläufig ausführbar sein, sondern können auch zum Beispiel nur für Analysezwecke verwendet werden.

Auf die folgenden Sprachen treffen die genannten Kriterien zu, und sie sind Beispiele für erfolgreich etablierte DSLs, die die Entwicklung spezifischer Softwaresysteme vereinfachen.

ACT-R [TLA06] bezeichnet eine auf Lisp basierende DSL, um kognitions-psychologische Modelle zur Beschreibung menschlicher Wahrnehmungs- und Lernprozesse zu verwenden.

ARMS [GHK+07, GKPR08, GHK+08b] bezeichnet eine Sprache zur Funktionsnetzmodellierung mit dem Ziel der Strukturierung und Wiederverwendung von Lastenheften für automotive Software.

Autofilter [RVWL03] ist eine Sprache zur Konfiguration von Kalmanfiltern für die Satellitensteuerung. Neben der Erzeugung von Produktivcode kann auch ein Beweis generiert werden, dass die gewählte Lösung optimal ist.

Ereignisgesteuerte Prozessketten [Sch02] dienen der Darstellung von Geschäftprozessen. Diese können die Grundlage für die Implementierung von Informationssystemen bilden oder zur Optimierung eingesetzt werden.

Graphviz [GV] ist eine DSL zur Beschreibung und Visualisierung von Graphen.

HTML [HTML] ist eine Sprache zur Strukturierung von Text und Grafik, die zusätzlich eine Verlinkung erlaubt und vor allem im Internet verwendet wird.

Matlab Simulink [Sim] ist eine Sprache zur Beschreibung regelungstechnischer Algorithmen. Neben der Simulation erlauben verschiedene Codegeneratoren die Erzeugung von Produktionscode.

Feature-Bäume [CE00] erlauben die Modellierung der Variabilität innerhalb einer Softwareproduktlinie.

SQL [SQL96] ist eine Datenbanksprache, die sowohl die Definition von Datenbanken erlaubt als auch die Abfrage und Manipulation von Daten ermöglicht.

Verilog [Ver01] ist eine Hardwarebeschreibungssprache zur Modellierung elektronischer Systeme, die für die Implementierung, Verifikation und zum Testen eingesetzt werden kann.

Die Definition einer DSL umfasst wie die Definition einer GPL die folgenden vier Elemente [HR04]:

- Die *konkrete Syntax* einer Sprache definiert die Menge der zulässigen Sprachinstanzen, die die Entwickler verwenden können.
- Die *abstrakte Syntax* einer Sprache beschreibt die strukturelle Essenz einer Sprache [DKLM84, Wil97a]. Sie bildet damit die zentrale Datenstruktur, auf der weitere Verarbeitungsschritte wie Codegenerierungen, Analysen und Transformationen aufbauen, und wird im Zusammenhang mit DSLs auch Domänenmodell genannt.
- Die *Kontextbedingungen* sind prüfbare Bedingungen, die die Menge der zulässigen Sprachinstanzen begrenzen. Sie ergänzen die abstrakte und konkrete Syntax um Einschränkungen, die sich in diesen Formalismen nicht oder nur schwer formulieren lassen. Die Kontextbedingungen sollen ausschließen, dass Sprachinstanzen als wohldefiniert verstanden werden, für die die Semantik keine Bedeutung definiert.
- Die *Semantik* einer Sprache bezeichnet deren Bedeutung. Dabei können verschiedene Techniken wie denotationale Semantik [SS71] und operationale Semantik [Plo81] genutzt werden, die jeweils auf der abstrakten Syntax aufbauen, ohne direkt Bezug auf die konkrete Syntax zu nehmen.

DSLs können durch verschiedene Techniken definiert werden, wobei zwischen grammatikbasierten und modellbasierten Ansätzen unterschieden wird. Grammatikbasierte Ansätze verwenden als Sprachdefinition meistens verschiedene Arten von Grammatiken für konkrete und abstrakte Syntax. Daraus ergibt sich, dass die Instanzen eine Baumstruktur

aufweisen, sofern kontextfreie Grammatiken verwendet werden. Verschiedene Techniken wie die Attributgrammatiken wurden entwickelt, um aus der Ausgangsdatenstruktur weitere Zusammenhänge zu errechnen. Alternativ können Graphgrammatiken [Nag79] verwendet werden, die die Spezifikation von visuellen Sprachen erleichtern und deren Instanzen ebenfalls eine Graphstruktur haben. Die existierenden Techniken orientieren sich meistens an Programmiersprachen, können aber auch zur Definition von DSLs verwendet werden. Modellbasierte Ansätze verwenden zu Klassendiagrammen ähnliche Datenmodellierungstechniken, die die abstrakte Syntax festlegen. Diese werden vor allem zur Definition von grafischen Modellierungssprachen, wie zum Beispiel die UML, genutzt.

1.3 Erfolgsfaktoren und Risiken von DSLs

In der Literatur werden ausführlich die Vorteile sowie die Herausforderungen und Randbedingungen für den erfolgreichen Einsatz von DSLs diskutiert. Die Vorteile der Verwendung von DSLs liegen vor allem in der engen Einbindung von Domänenexperten, die durch ihre Expertise das Softwaresystem selbst besser verstehen oder sogar selbständig entwickeln können [DKV00]. Damit ist vor allem gemeint, dass die Modelle einen Abstraktionsgrad und eine Notation verwenden, die aus dem Anwendungsgebiet entspringen und sich nicht zwangsläufig an den in der Informatik üblichen Abstraktionen und Notationen orientieren. Mit der Entwicklung von DSLs stellt die Informatik somit Hilfsmittel zur Verfügung, die durch Experten zur Lösung ihrer Probleme verwendet werden können. Der Abstraktionsgrad einer DSL erlaubt es den Entwicklern, auf dem Abstraktionsgrad der Domäne Optimierungen durchzuführen [MP99]. Diese können mit automatischen Optimierungen bei der Codegenerierung kombiniert werden.

Durch den Einsatz von DSLs wird die Produktivität der Entwickler gesteigert, die Verlässlichkeit der Systeme verbessert und die Wartung vereinfacht [DK98]. Die gesteigerte Produktivität entsteht vor allem durch die kompakte Notation einer DSL gegenüber einer funktional gleichwertigen Implementierung in einer GPL. Ein direkter Vergleich ist bei DSL-Code und daraus generiertem GPL-Code möglich: Innerhalb der Entwicklung des MontiCore-Frameworks [GKR+06, KRV07b, KRV08] konnte bei der Parsererzeugung ein Faktor von ungefähr 16 und bei der AST-Klassengenerierung ein Faktor von ungefähr 47 beobachtet werden.[1] Andere Quellen erreichen bei der Generierung von Modelltraversierungen und -transformationen ein Verhältnis von 1 zu 15 [GK03]. Bei den Vergleichen ist zu beachten, dass das reine Verhältnis von Quellcodezeilen zueinander kein allein aussagekräftiges Kriterium ist, sondern nur eine Orientierung erlaubt. Dieses zeigt sich vor allem an der seit langem geführten Diskussion, ob die Anzahl der Quellcodezeilen ein gutes Maß für die Größe einer Software ist. In jedem Fall gilt als gesichert, dass die Verlässlichkeit und Wartbarkeit durch die Kompaktheit der Entwicklerquellen verbessert wird. Kompakte Modelle lassen sich besser verstehen und Fehler werden somit schneller offensichtlich.

Insbesondere die Trennung der fachlichen Problemlösung von der technischen Realisierung stellt einen Vorteil gegenüber der normalen Softwareentwicklung dar. Das in einer DSL formalisierte Domänenwissen kann unabhängig von einer Implementierung für eine Vielzahl an Aufgaben wie Migration von technischen Plattformen, Optimierung und Wissensmanagement verwendet werden. Die erstellten Generatoren können in weiteren Projekten mit anderen Modellen eingesetzt werden. Die Trennung in DSL-Modelle und Generatoren er-

[1] Die Zahlen beziehen sich auf die SLOC der Grammatiken im Vergleich zu den SLOC der daraus erzeugten Klassen und Schnittstellen für das MontiCore-Grammatikformat in der Version 1.1.5.

leichtert die Wartung von Software, weil bei geänderten Anforderungen häufig nur eines von beiden angepasst werden muss und sich so die Auswirkungen besser lokal begrenzen lassen. Somit begünstigt der Einsatz von DSLs die Wiederverwendung von Software oder Modellen in anderen Anwendungsbereichen [MHS03].

Die modellgetriebene Entwicklung verwendet DSL-Modelle als zentrale Artefakte bei der Softwareentwicklung. Die kompakten Modelle sind durch die Nähe zur Domäne nahezu selbst-dokumentierend. Dadurch können anders als bei der Programmierung, in der das Programm durch einen Compiler in eine ausführbare Form übersetzt wird, leichter auch weiterführende Ziele verfolgt werden: automatisierte Testcodeerzeugung, Dokumentation und statische sowie dynamische Analysen. Zusätzlich besteht für DSLs im Gegensatz zu GPLs nicht der Zwang, dass sie ausführbar sein müssen [MHS03], was ein anderes Herangehen an Problemstellungen erlaubt und auch den Einsatz von DSLs in frühen Phasen der Softwareentwicklung ermöglicht.

Die Risiken bei der Verwendung von DSLs liegen vor allem in den Kosten für den Entwurf, die Implementierung, Weiterentwicklung und Wartung einer DSL und den Kosten für die Schulung der Entwickler [DKV00]. Dieses Risiko ergibt sich für fast jede neue Technologie, weil Compiler für eine Vielzahl an Plattformen in hoher Qualität kostengünstig verfügbar sind. DSLs hingegen müssen relativ oft spezifisch für ein Projekt entwickelt oder zumindestens angepasst werden [DK98]. Dieser Punkt wird in dieser Arbeit insofern adressiert, als mit dem Grammatikformat eine möglichst einfache und kompakte Sprachdefinition erreicht werden kann. Ergänzend dazu kapselt das DSLTool-Framework viel Wissen, das nötig ist, um modellbasierte Codegeneratoren zu entwickeln, und verringert somit den Entwicklungsaufwand. Die Schulung der DSL-Nutzer ist notwenig, kann aber dadurch vereinfacht werden, dass die DSLs aus Elementen aufgebaut werden, die zum Standardrepertoire in der Informatik gehören, was den Lernaufwand reduziert.

Es ist außerdem schwierig, den richtigen Aufgabenbereich für die DSL zu definieren, um dabei einen Mehrwert für die Entwickler zu erreichen, ohne eine neue allgemeingültige Programmiersprache zu entwickeln [DKV00]. Der Entwurf einer DSL unterscheidet sich hier kaum vom Entwurf einer Programmiersprache. Geeignete Abstraktionen zu finden ist ein iterativer Prozess, in dem die Sprache immer wieder den Erfordernissen der Entwickler angepasst werden muss. GPLs haben hier den Vorteil, dass oftmals die Nutzeranzahl deutlich höher ist und sich somit der hohe Aufwand leichter amortisieren kann. Bei DSLs hingegen ist es hilfreich, wenn die DSL-Entwickler zunächst mit einer kleinen Anzahl von Produktentwicklern zusammenarbeiten und so schnelle Rückmeldungen zur Verbesserung der Sprache nutzen können.

Ein oft genanntes Gegenargument für den Einsatz von DSLs ist, dass der aus einer DSL erzeugte Code nicht so effizient wie eine handcodierte Anwendung ist [DKV00]. Dieses Argument gilt insbesondere, wenn die Abstraktion der DSL ungünstig gewählt ist, und erinnert an Argumente, die bei der Einführung von Hochsprachen und Compilern gültig waren. Eine Kompensation dieses Effektes ist durch Optimierungen bei der Codeerzeugung möglich.

Paul Hudak fasst die Abwägung der Vor- und Nachteile von DSLs in der Abbildung 1.2 zusammen, wobei ausschließlich die Kosten miteinander verglichen werden. Die beschriebenen Vorteile von DSLs in Bezug auf die allgemeinen Verbesserungen des Prozesses, die verbesserte Wartbarkeit und die Möglichkeiten der modellbasierten Entwicklung sind ebenfalls zu beachten. Paul Hudak gibt keine Maßzahl für den Schnittpunkt der beiden Kurven und die Höhe der notwendigen Anfangsinvestition an [Hud98]. Das publizierte Zahlenmaterial hierzu ist wenig aussagekräftig, weil es für DSLs im Wesentlichen die bereits zitierten Vergleiche der generierten Quellcodemenge mit der ursprünglichen Menge an DSL-Code

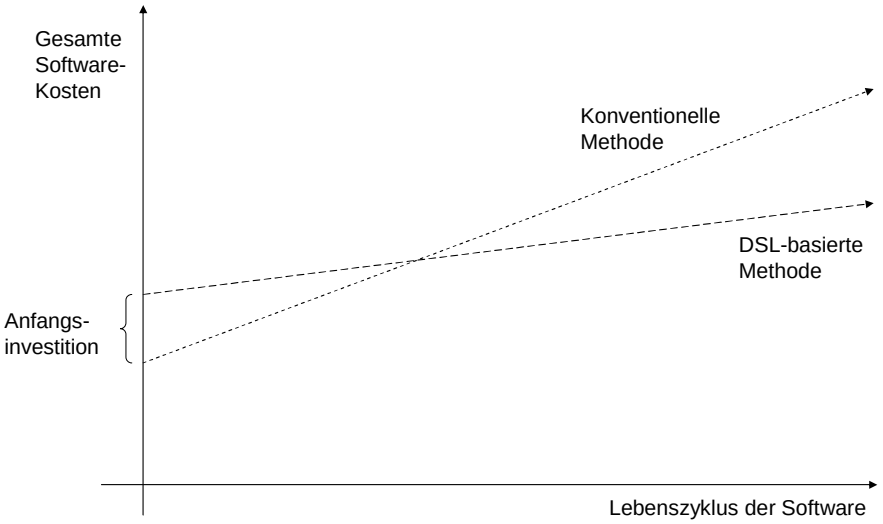


Abbildung 1.2: Amortisierung beim Einsatz von DSLs [Hud98]

gibt. Zusätzlich wird von Produktivitätssteigerungen von Faktor 3 [KMB+96] und Faktor 10 [MCN] in bestimmten Fallstudien berichtet.

Hilfreich ist ein Vergleich mit der verwandten Disziplin Software-Produktlinien. Diese setzen Feature-Bäume, eine spezielle DSL, ein, um die Variabilitäten einer Familie von Softwareprodukten zu beschreiben. Darauf aufbauend können dann Produkte durch Angabe einer Parametrisierung aus einem Pool an Softwarefragmenten abgeleitet werden, wobei auch spezifische DSLs verwendet werden können [CE00]. Die Anfangsinvestition dieser speziellen Form der DSL-basierten Entwicklung amortisiert sich, wie eine Auswertung verschiedener Fallstudien ergeben hat, durchschnittlich nach dem dritten Produkt [CN02]. Einzelne Studien berichten von dreimal kürzeren Entwicklungszeiten mit einem viermal kleineren Entwicklungsteam. Dabei könnte zusätzlich die Qualität deutlich gesteigert werden, insbesondere weil durchschnittlich nur 3% des Quellcodes noch spezifisch für ein Produkt neu entwickelt werden müssen [MO07].

1.4 Herausforderungen der DSL-Entwicklung

Der heutige Stand der agilen DSL-Entwicklung ist durch die folgenden Herausforderungen gekennzeichnet, die im folgenden Abschnitt detailliert adressiert werden:

- Die Sprachdefinition einer DSL ist oftmals auf mehrere Dokumente verteilt, so dass keine zentrale Referenz existiert, die sowohl den Anforderungen der Entwickler als auch der Nutzer einer DSL genügt.

- Die Definition einer DSL und darauf basierender Algorithmen kann in den aktuell verwendeten Ansätzen nicht ausreichend genug modularisiert werden. Dadurch wird die modulare Integration von häufig verwendeten Sprachfragmenten wie Anweisungen einer Programmiersprache in Modelle schwierig.

- Die konstruktive Entwicklung einer DSL und modellbasierter Werkzeuge aus existierenden Sprachreferenzen, Datenmodellen oder exemplarischem Code ist nicht systematisch beschrieben.

- Die Strukturierung von komplexen und erweiterbaren DSL-Werkzeugen und darin realisierten Codegenerierungen, die über die Verwendung von Modelltransformationen und einer Template-basierten Codeerzeugung hinausgehen, erfolgt individuell für jede DSL und es wird keine allgemeingültige Lösung verwendet.
- Die aufeinander abgestimmte Weiterentwicklung von DSL-Definitionen, DSL-Modellen, Laufzeitumgebungen und Generatoren ist in den existierenden Ansätzen nicht systematisch möglich.

1.5 Der MontiCore-Ansatz

In dieser Arbeit wird das Paradigma der generativen DSL-Entwicklung entwickelt, das heißt, für die DSLs werden jeweils Generatoren entwickelt, die zur agilen modellgetriebenen Softwareentwicklung beitragen. Dabei besteht die Eingabe der Generatoren aus Sätzen, die konform zur Grammatik der Sprache sind und im Folgenden einheitlich als Modelle bezeichnet werden.

Wird aus den DSLs Produktionscode erzeugt, dann muss er nicht zwangsläufig eine komplette in sich lauffähige Software sein. Vielmehr wird üblicherweise ein fixer Teil extrahiert, der den für alle Modelle gleichen und damit vom Modell unabhängigen Code enthält, und ein variabler Teil, der das Ziel der Generierung ist. Der fixe Teil der Software kann eine Sammlung von Klassen (Laufzeitumgebung) sein, auf die sich der variable Teil durch Unterklassenbildung oder Methodenaufruf bezieht. Alternativ kann der fixe Teil eine komplette Software sein, die dann über den variablen Anteil nur noch parametrisiert wird. Die Generatoren erzeugen aus diesen Modellen eine Menge an Artefakten wie zum Beispiel Produktivcode. Die Abbildung 1.3 zeigt mögliche Nutzungsformen von DSLs.

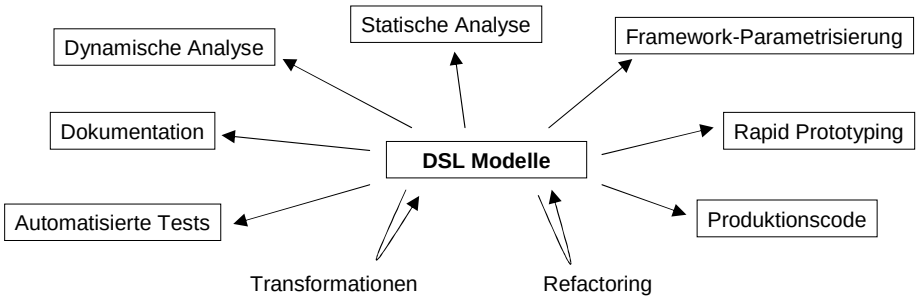


Abbildung 1.3: Nutzungsformen von DSLs bei agiler modellgetriebener Entwicklung

MontiCore ist ein Framework, das für die agile modellgetriebene Softwareentwicklung eingesetzt werden kann. Dafür werden die Teile einer Software identifiziert, für die sich der Einsatz einer DSL mit einer Codegenerierung lohnt. Dabei ist insbesondere eine kompakte Beschreibung eines Sachverhalts gegenüber einer umfangreichen Implementierung in einer Programmiersprache wünschenswert. Es wird nicht die vollständige Erzeugung eines Softwaresystems aus Modellen als Ziel definiert, sondern die Ergänzung des manuell implementierten Quellcodes. Dieses gründet sich auf der Tatsache, dass es bestimmte Teile einer Software gibt, die sich in einer Programmiersprache knapp und eindeutig implementieren lassen, und somit keine DSL existiert, die hier eine Effektivitätssteigerung ermöglicht. Dieses deckt sich mit der Empfehlung in [Wil03], eine 80%-Lösung in Zusammenhang mit DSLs anzustreben.

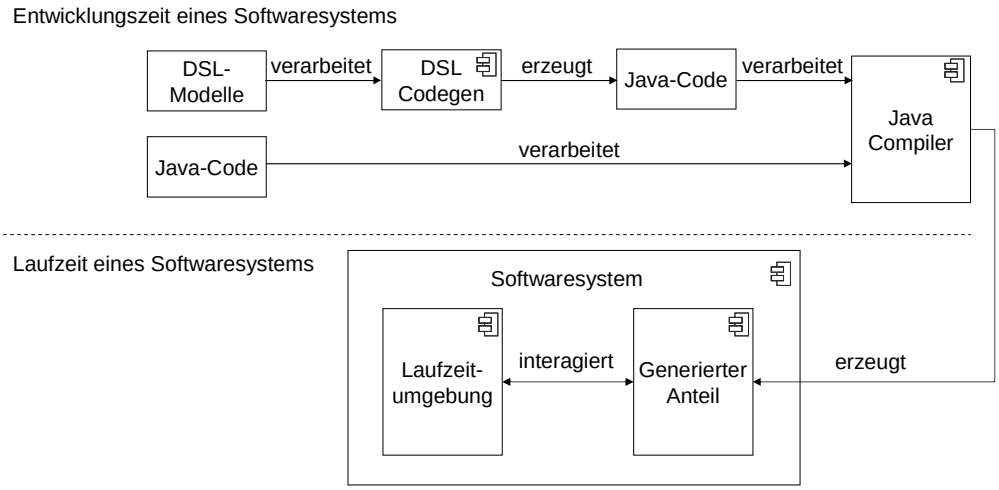


Abbildung 1.4: Verwendung von domänenspezifischen Codegeneratoren

MontiCore wird innerhalb der agilen modellgetriebenen Softwareentwicklung dazu verwendet, DSLs und Generatoren zu entwickeln. Diese Generatoren werden dann ihrerseits bei der Erstellung einer Software genutzt, die mit Hilfe von DSL-Modellen und ergänzendem Quellcode entwickelt wird. Die Abbildung 1.4 zeigt die Verwendung von Generatoren bei der Erstellung von Software.

Die Verwendung von visuellen oder textuellen Sprachen hat unterschiedliche Vor- und Nachteile. Die Vorteile visueller Sprachen liegen vor allem in der einfachen Erfassbarkeit der Inhalte, wohingegen textuelle Repräsentationen schneller erstellt werden können und eine automatische Formatierung technisch einfacher zu realisieren ist. Eine detaillierte Diskussion findet sich in [GKR+07]. In dieser Arbeit wird sich auf die Entwicklung textueller Sprachen konzentriert, weil so eine nahtlose Integration von Quellcode und Modellen erreicht werden kann, die im Sinne der agilen modellgetriebenen Entwicklung [Rum04a, Rum04b] auf einer Abstraktionsebene verwendet werden und sich vom Erscheinungbild voneinander nicht unterscheiden.

Bei anderen Ansätzen zur Definition von DSLs wird die Sprachdefinition in verschiedene partielle sich ergänzende Definitionen aufgeteilt. Die getrennte Spezifikation von konkreter und abstrakter Syntax erlaubt generell, beide Artefakte unabhängig voneinander wieder zu verwenden. Dadurch kann beispielsweise die abstrakte Syntax mit verschiedenen konkreten Syntaxen genutzt werden, um den Bedürfnissen verschiedener Nutzergruppen gerecht zu werden und somit eine gemeinsame Codegenerierung für verschiedene Repräsentationen zu verwenden. Diese Trennung hat sich bei der Realisierung von Programmiersprachen als vorteilhaft erwiesen, wird jedoch in dieser Arbeit aus zwei Gründen für DSLs als ungeeignet angesehen:

- Eine DSL hat oft eine einzige domänenspezifische Notation, die sich auch in der beschriebenen Domäne entwickelt hat. Unterschiedliche Notationen für denselben Modelltyp, die von verschiedenen Nutzerkreisen verwendet werden, würden die Kosten für die Erstellung der Werkzeuge und deren Dokumentation erhöhen und leicht zu Inkonsistenzen führen.

- Durch die Etablierung von Modelltransformationssprachen stehen deklarative, wartbare und flexible Möglichkeiten zur Überführung einer abstrakten Syntax in eine andere zur Verfügung, so dass auf diesem Wege Codegenerierungen und Analysen für unterschiedliche Notationen gemeinsam genutzt werden können.

Dagegen hat eine integrierte Sprachdefinition für abstrakte und konkrete Syntax die folgenden Vorteile:

- Die abstrakte und die konkrete Syntax sind automatisch konsistent.
- Semantisch ähnliche Elemente einer Sprache sollten sich auch für den Nutzer ähnlich darstellen, damit von der Eingabe leichter auf die Bedeutung eines Programms zu schließen ist (vgl. auch [Hoa73, Wir74]). Eine gemeinsame Spezifikation erzwingt dieses per Konstruktion.
- Die Sprachdefinition kann effektiv als eine zentrale Dokumentation innerhalb einer agilen evolutionären Entwicklung dienen, so dass externe Dokumentation zweitrangig ist.
- Es existiert automatisch eine konkrete Syntax, was die Verwendung der Sprache durch Nutzer und das Testen darauf basierender Werkzeuge durch Entwickler erleichtert, da diese nicht, wie bei Metamodellierung oft üblich, eine generische Repräsentation verwenden müssen.

Die modellgetriebenen Ansätze zur Definition von Modellierungssprachen wie [BSM+03, OMG06a] konzentrieren sich vor allem auf die abstrakte Syntax. Die konkrete Syntax wird, wenn sie überhaupt definiert wird, mittels einer zweiten Modellart definiert (vgl. z.B. [JBK06, MFF+06]). Dieses ignoriert die Tatsache, dass Modelle auch direkt von den Entwicklern der DSL zum Testen der Generatoren erstellt werden müssen und daher eine komfortable konkrete Syntax die Qualitätssicherung erleichtert.

Innerhalb einer DSL-basierten Entwicklung kann es vorkommen, dass dieselben Entwickler in unterschiedlichen Rollen einerseits Modelle erstellen und andererseits auch die Codegeneratoren auf Basis der abstrakten Syntax entwickeln. Eine starke Kopplung der abstrakten und der konkreten Syntax erleichtert hierbei das Verständnis der Sprache, weil die Eingaben direkt mit den Elementen der abstrakten Syntax korrespondieren und keine komplexe Abbildung zwischen beiden Artefakten möglich ist. Zusätzlich wird ein Wechsel zwischen den Rollen vereinfacht, weil mit einem Rollenwechsel trotzdem die für die Rolle relevante Struktur der Sprache ähnlich bleibt.

Die existierenden Ansätze betrachten die Definition einer DSL meistens als eine atomare Einheit. Für eine ingenieurmäßige Sprachentwicklung [KLV05], die sich als Ziel setzt, systematisch neue DSLs zu entwickeln, sind jedoch Modularitätsmechanismen wichtig. Nur so kann die systematische Wiederverwendung von Sprachen erreicht werden. In dieser Arbeit werden Konzepte zur Modularisierung der Sprachdefinition erarbeitet. Weitergehend wird beschrieben, wie darauf aufbauende Artefakte wie Kontextbedingungen, Traversierung von Datenstrukturen und Codegenerierungen modular entwickelt werden können. Nur so lassen sich systematisch Bibliotheken etablieren, die Sprachfragmente und Generierungen enthalten, die es langfristig erlauben, durch Wiederverwendung den Aufwand für die Entwicklung generativer Werkzeuge zu reduzieren. Ein Beispiel hierfür sind GPL-Fragmente, die häufig als eine Aktionssprache in Modellen zur Formulierung ausführbarer Anteile genutzt werden. Um eine Konsistenzprüfung für solche heterogenen Modelle zu realisieren, ist zumindest eine Repräsentation der GPL als Modell notwendig. Zur Amortisierung des Entwicklungsaufwands werden daher im Folgenden Möglichkeiten dargestellt, Sprachfragmente in mehreren Modellarten nutzen zu können.

Der MontiCore-Generator wird bei der Erstellung von Generatoren selbst eingesetzt, um aus der DSL-Definition Komponenten zur Sprachverarbeitung automatisch zu erzeugen. Dabei ist es insbesondere möglich, auf Bibliotheken zurückzugreifen, die bereits entwickelte

Sprachdefinitionen enthalten. Diese können auf eine strukturierte Art und Weise wieder verwendet werden. Die Erzeugung von zusätzlichen Komponenten aus der Sprachdefinition ist durch die Erweiterung des Generators an den vorgesehenen Erweiterungspunkten möglich. Die Sprachdefinition wird in Kapitel 3 erklärt und die modulare Wiederverwendung bereits entwickelter Anteile in Kapitel 4 erläutert. Beides wird in Kapitel 5 formalisiert. Die Eigenschaften, die Nutzung und die Erweiterung des MontiCore-Generators werden in Kapitel 6 erklärt und in Kapitel 7 zu anderen Ansätzen abgegrenzt.

Eine Form zur Verwendung modellgetriebener Techniken ist die Definition einer DSL und anschließend das Schreiben einer Codegenerierung. Dieses Vorgehen ist aber nicht zwangsläufig der einfachste und effizienteste Zugang für ungeschulte Entwickler. Daher wird die Einbindung von bereits vorhandenen Sprachbeschreibungen und die Entwicklung aus exemplarischem Quellcode in Kapitel 8 als pragmatische Vorgehensweise beschrieben.

Die derzeitige Forschung in der Entwicklung von DSLs fokussiert sich stark auf die Definition der Sprache. Codegenerierungen werden teilweise als Modelltransformationen formalisiert, indem ein Metamodell für die Zielsprache verwendet wird. Zur Quellcodeerzeugung werden oftmals auch Template-basierte Ansätze verwendet. Komplexe Codegenerierungen lassen sich jedoch so meistens nur schwer und umständlich erstellen. Praktische Unterstützung für Entwickler von Codegenerierungen existiert kaum, so dass wiederkehrende Aufgaben von den Entwicklern auch wiederkehrend gelöst werden müssen. Compiler-Frameworks sind nur begrent einsetzbar, weil nicht eine homogene Eingabesprache mit einer einzelnen Codegenerierung verwendet wird, sondern aufgrund der verschiedenen Zielsetzungen heterogene Eingabesprachen mit jeweils mehreren Codegenerierungen zu einem Werkzeug kombiniert werden sollen. Zusätzlich wurde eine einheitliche Zwischencodeerstellung wie zum Beispiel die Single-Static-Assignment-Form für GPLs bei Modellen bisher nicht identifiziert. Dieses führt zu einer Diversifikation der verwendeten Lösungen in der modellbasierten Entwicklung. Als Mittel für eine strukturierte Entwicklung domänenspezifischer Codegeneratoren wird in dieser Arbeit ein Entwicklerframework, das DSLTool-Framework, vorgestellt, das wiederkehrende Aufgaben auf eine allgemeingültige Art und Weise löst, so dass der Entwickler sich auf die Kernpunkte der Generatorenprogrammierung konzentrieren kann.

Der MontiCore-Ansatz bei der Erstellung der Generatoren zeichnet sich vor allem durch die Verwendung des DSLTool-Frameworks aus. Details zu diesen Lösungen finden sich in Kapitel 9. Dabei werden die Modelle im Framework durch Algorithmen, so genannte Workflows, verarbeitet. Diese können programmiert oder durch spezielle Templates definiert werden. Zusätzlich ist auch hier eine Bibliotheksbildung möglich, so dass passend zur Sprachdefinition auch verarbeitende Algorithmen wieder verwendet werden können. Abbildung 1.5 gibt einen Überblick über die Entwicklung eines domänenspezifischen Codegenerators.

Die so entstandenen Codegeneratoren können bei der Softwareerstellung genutzt werden, indem aus DSL-Modellen Code erzeugt wird, der nach der Generierung nicht verändert wird. Dieses ermöglicht bei Änderungen der Modelle jederzeit eine erneute, automatische Generierung des Quellcodes, ohne dass manuelle Änderungen am generierten Code beachtet werden müssen. Ergänzend dazu kann Java-Quellcode erstellt werden, der zusätzliche Funktionalität realisiert, die sich durch Modelle nicht oder nur schlecht darstellen lässt.

Die Entwicklung eines Softwaresystem wird bei der Verwendung von Generatoren im Folgenden modellgetrieben genannt, weil die DSL-Modelle bei ausreichender Identifikation geeigneter DSLs einen Großteil des Zielsystems direkt beeinflussen. Die DSL-Modelle sind dann primäre Artefakte der Softwareentwicklung, weil sie sich nicht nur mittelbar in Form von Anforderungen oder einer Spezifikation auf die Software auswirken, sondern diese direkt beeinflussen.

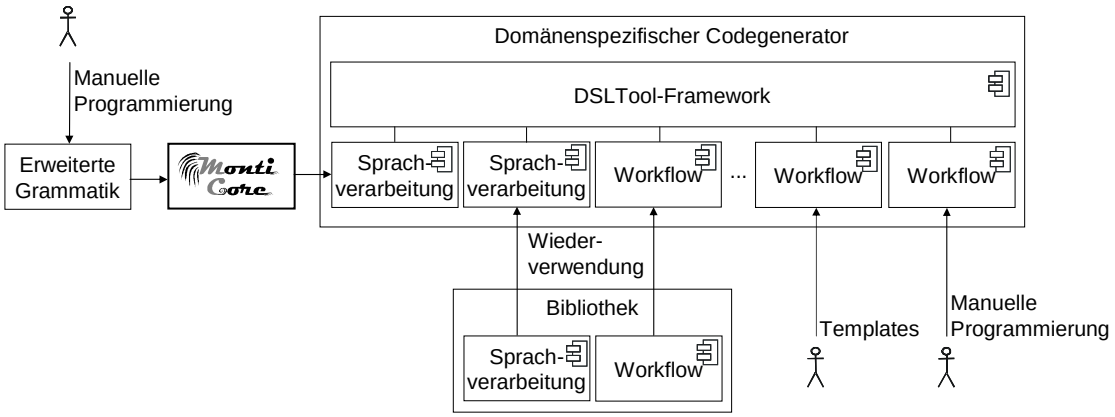


Abbildung 1.5: Übersicht über die Erstellung von domänenspezifischen Codegeneratoren mit MontiCore

Eine agile Form der Softwareentwicklung kann bedeuten, dass der Generator während der Erstellung des Softwaresystems verändert wird. Die Programmierung des Generators wird somit Teil der Produktentwicklung, weil der Generator während der Entwicklung fortlaufend ergänzt und angepasst wird. Somit können Fortschritte bei der Entwicklung des Zielsystems durch die Implementierung von DSL-Modellen, Java-Quellcode und die Weiterentwicklung des Generators gemacht werden. Die einzelnen Aktivitäten sind dabei ineinander verschränkt und orientieren sich an den Bedürfnissen des Projekts. Die Nachteile eines solchen Vorgehens liegen darin, dass mit der DSL-Definition auch die DSL-Modelle angepasst werden müssen, was bei einer großen Anzahl an Modellen sehr aufwändig sein kann. Es existieren erste akademische Ansätze [PJ07a, PJ07b], die dieses Problem adressieren und Koevolutionsstrategien für Modelle und DSLs definieren. In dieser Arbeit wird in Abschnitt 9.3.9 die gleichzeitige Weiterentwicklung der Laufzeitumgebung und der Codegenerierung besprochen.

Die Eigenschaften des Ansatzes und der Einsatz des DSLTool-Frameworks werden durch Fallstudien in Kapitel 10 illustriert. Dabei wird exemplarisch für die generative Entwicklung die Konzeption einer Java-Spracherweiterung zur Modelltransformation beschrieben. Als ein Beispiel für den analytischen Einsatz wird die Funktionsnetzmodellierung automotiver Systeme verwendet.

1.6 Wichtigste Ziele und Ergebnisse

Die wichtigsten Ziele dieser Arbeit sind:

- Die Konzeption und Erstellung eines Frameworks zur Definition von domänenspezifischen Sprachen und deren Einsatz innerhalb eines agilen Entwicklungsprozesses.
- Die Identifikation von Möglichkeiten zur Wiederverwendung bereits qualitätsgesicherter Artefakte innerhalb der DSL-Entwicklung zur Reduktion von Entwicklungsaufwand oder Steigerung der Qualität.
- Die Erarbeitung von praxistauglichen Methoden, die nicht von einer direkten DSL-Definition ausgehen, sondern andere strukturelle Formalismen oder exemplarischen Quellcode als Beginn der Entwicklung verwenden.
- Verwendung agil erstellter DSL-Werkzeuge in software-intensiven Projekten.

Die wichtigsten wissenschaftlichen Beiträge dieser Arbeit können wie folgt zusammengefasst werden:

- Die vorliegende Arbeit erlaubt die Spezifikation abstrakter und konkreter Syntax in einem integrierten kompakten Format. Diese Sprachdefinition bildet die Grundlage für die Einbindung von DSLs in einen agilen Entwicklungsprozess und dient als zentrale Dokumentation.
- Es werden zwei Modularitätsmechanismen für Grammatiken erklärt, die die modulare DSL-Definition ermöglichen. Erstens wird die in [MŽLA99] für die konkrete Syntax beschriebene Grammatikvererbung auf die abstrakte Syntax erweitert und so die Spezialisierung und Erweiterung von Sprachen ermöglicht. Zweitens wird ein zu [Bos97] erweiterter Mechanismus zur Verwendung von Sprachkomponenten bereitgestellt, der eine flexible Kombination mehrerer Sprachen erlaubt, die sich in ihrer lexikalischen Struktur grundlegend unterscheiden können. Beide Mechanismen können beliebig miteinander kombiniert werden.
- Das verwendete Grammatikformat ist im Gegensatz zu anderen Formalismen modular erweiterbar, so dass zusätzliche Informationen spezifiziert werden können und darauf aufbauend weitere Infrastruktur automatisch generiert werden kann.
- Die Entwicklung von Codegeneratoren und Analysewerkzeugen für DSLs wird durch die Bereitstellung eines Entwicklerframeworks vereinfacht, so dass bewährte Lösungen ohne zusätzlichen Entwicklungsaufwand genutzt werden können und damit die Qualität der entstehenden Werkzeuge im Vergleich zu existierenden Ansätzen gehoben wird.
- Die Entwicklung einer Template-Engine, die compilierbaren Quellcode als Templates verwendet, ermöglicht ein integriertes Refactoring der Templates und einer Laufzeitumgebung. Darauf aufbauend wurde eine Methodik definiert, die aus exemplarischem Quellcode schrittweise einen Generator entwickelt, dessen Datenmodell automatisiert abgeleitet werden kann.

1.7 Aufbau der Arbeit

Teil I, Grundlagen, gibt einen Überblick über die Verwendung von DSLs in der Softwareentwicklung.

Kapitel 1, Einführung, führt in die Thematik ein, diskutiert die Definition einer DSL und erläutert die Vor- und Nachteile der Verwendung von DSLs innerhalb der Softwareentwicklung.

Kapitel 2, Entwurf und Einsatz von domänenspezifischen Sprachen in der Softwareentwicklung, erläutert die Verwendung von DSLs in der Softwareentwicklung und beschreibt eine Methodik zur Entwicklung von DSLs.

Teil II, MontiCore, beschreibt die Entwicklung von DSLs mit MontiCore.

Kapitel 3, Grammatikbasierte Erstellung von domänenspezifischen Sprachen, beschreibt die Definition einer Sprache mittels eines kompakten Grammatikformats. Dabei werden die konkrete und die abstrakte Syntax einer DSL aus diesem Format abgeleitet.

Kapitel 4, Modulare Entwicklung von domänenspezifischen Sprachen, beschreibt die Modularitätsmechanismen des MontiCore-Grammatikformats.

Kapitel 5, Formalisierung, formalisiert die Ableitung der konkreten und abstrakten Syntax einer DSL aus dem Grammatikformat.

Kapitel 6, MontiCore, erklärt die Software-Architektur des MontiCore-Frameworks sowie dessen Erweiterungspunkte und existierende Erweiterungen.

Kapitel 7, Verwandte Arbeiten zu MontiCore, erklärt die Unterschiede und Gemeinsamkeiten des MontiCore-Ansatzes mit anderen Arbeiten.

Teil III, Definition von Werkzeugen mit dem DSLTool-Framework, fasst die in dieser Arbeit betrachteten Aspekte der DSL-Entwicklung mit dem DSLTool-Framework zusammen und illustriert sie durch Fallstudien.

Kapitel 8, Methodiken zur Entwicklung von DSLs, zeigt drei Methodiken zum Entwurf von DSLs, die über die direkte Sprachdefinition hinausgehen.

Kapitel 9, Verarbeitung von Sprachen mit dem DSLTool-Framework, erklärt die Erstellung analytischer und generativer Werkzeuge und die dabei identifizierte Querschnittsfunktionalität.

Kapitel 10, Fallstudien, beinhaltet drei Fallstudien mit dem MontiCore-Framework, insbesondere die Erstellung einer Sprache zur Funktionsnetzmodellierung automotiver Systeme.

Teil IV, Epilog, schließt die Arbeit mit einer Zusammenfassung und einem Ausblick ab.

Kapitel 11, Zusammenfassung und Ausblick, fasst die Arbeit zusammen. Generelle Begrenzungen werden diskutiert und zukünftige Arbeiten werden vorgestellt, die den präsentierten Ansatz ergänzen können.

Teil V, Anhänge, ergänzt die anderen Abschnitte.

Anhang A, Glossar, erklärt die wichtigsten in dieser Arbeit verwendeten Fachbegriffe.

Anhang B, Zeichenerklärungen, erläutert die verwendeten Abbildungen.

Anhang C, Abkürzungen, listet die Abkürzungen auf.

Anhang D, Grammatiken, beschreibt die Unterschiede der in Kapitel 5 formalisierten Essenz zum kompletten Grammatikformat.

Anhang E, Index der Formalisierung, listet die wichtigsten Elemente des Kapitels 5 auf.

Kapitel 2

Entwurf und Einsatz von domänenspezifischen Sprachen in der Softwareentwicklung

Durch die zunehmende Verbreitung von informationsverarbeitenden Systemen in vielen Bereichen des alltäglichen Lebens und früher informatikfernen Anwendungsbereichen wie der Automobilindustrie gibt es eine erhöhte Notwendigkeit zur Zusammenarbeit von Informatikern mit den Experten dieser Anwendungsgebiete. In der üblichen Softwareentwicklung definieren diese Experten, auch Domänenexperten genannt, die notwendigen Anforderungen an das System und formulieren dabei Lösungen auf Fachprobleme in ihrer eigenen Terminologie. Dieser mittelbare Einfluss von Experten ist insbesondere bei heterogenen Systemen wie einem Automobil, die neben der reinen Informationstechnik zum Großteil aus mechanischen Elementen bestehen, nur bedingt hilfreich, weil die Anforderungserhebung dadurch sehr aufwändig werden kann. Zusätzlich erfordert dieses Vorgehen viel Lernaufwand für die Informatiker, da sie speziell angestrebte Lösungen in verschiedenen fachlichen Anwendungsbereichen verstehen müssen und sich so ebenfalls zu Domänenexperten entwickeln. Eine zusätzliche Herausforderung bei der Entwicklung dieser heterogenen Systeme ist, dass eine komplette Anforderungserhebung am Anfang eines Projekts selten möglich ist, da die mechanischen Anteile sich oft ebenfalls noch in der Entwicklung befinden und daher die Entwicklungsprozesse miteinander verschränkt sind. Der Einsatz von DSLs ermöglicht die direkte Einbindung von Domänenexperten in den Entwicklungsprozess der Software und reduziert ihren Einfluss nicht auf das Schreiben von Spezifikationen, sondern lässt sie Teil des Entwicklerteams werden. Durch die Generierung der Software aus den DSL-Modellen lässt sich der fachliche vom technischen Teil der Realisierung trennen, so dass auf fachliche Änderungen aufgrund sich wandelnder Anforderungen besser und kontrollierter reagiert werden kann.

Die Entwicklung einer DSL und ihr erfolgreicher Einsatz innerhalb solcher Softwareprojekte erfordert eine gute Einbettung in den Entwicklungsprozess. Der Einsatz und die Entwicklung von DSLs können sich am Generative Programming [CE00] und an Software-Produktlinien [PBL05] orientieren, die beide anstatt der Entwicklung handerstellter individueller Software die Erstellung von Softwarefamilien behandeln. Die beiden Ansätze teilen den Softwareentwicklungsprozess in zwei Teile, das Domain Engineering, das als Ziel hat, eine Plattform für die Produktfamilie zu etablieren, und das Product Engineering, das die Plattform nutzt, um konkrete Produkte abzuleiten. In der DSL-unterstützten Softwareentwicklung wird diese Zweiteilung ebenfalls verwendet, wobei neben einem Produkt

auch eine oder mehrere DSLs verwendet werden. Die Entwicklung einer DSL ist immer mit der Erstellung dazugehöriger Werkzeuge verbunden, die die Modelle zur Parametrisierung der in Abschnitt 1.5 beschriebenen Codegenerierungen verwenden. Wird im Folgenden der Begriff DSL-Entwicklung verwendet, ist damit nicht nur die Sprachdefinition, sondern immer auch die Entwicklung der Werkzeuge gemeint. Der Fokus einer DSL-unterstützten Softwareentwicklung liegt jedoch weniger auf der Etablierung einer Plattform wie bei den Software-Produktlinien, da die Produkte nicht zwangsläufig eine gemeinsame Referenzarchitektur verwenden müssen. Vielmehr können die Produkte auch nur insofern ähnlich sein, als die Struktur von Subsystemen oder die Realisierung bestimmter Querschnittsfunktionalitäten gleich sind. Dies ist insbesondere dann der Fall, wenn DSLs aus technischen Domänen bei der Realisierung des Produkts verwendet werden.

Dieses Kapitel beschreibt zunächst in Abschnitt 2.1, welche Aktivitäten in den meisten Softwareentwicklungsprozessen auftauchen und daher im Folgenden betrachtet werden. In Abschnitt 2.2 wird dargestellt, für welche dieser Aktivitäten DSLs verwendet werden können und wie eine aktivitätenübergreifende Verwendung aussieht. In Abschnitt 2.3 wird dargestellt, was die typischen Arbeitspakete innerhalb der Aktivitäten einer DSL-Entwicklung sind. Daraufhin wird in Abschnitt 2.4 erklärt, wie sich die DSL-Entwicklung und die Produkterstellung miteinander zu einem agilen Prozess eng koppeln lassen. In Abschnitt 2.5 werden die verschiedenen Möglichkeiten, eine DSL zu realisieren, übersichtsartig dargestellt. In Abschnitt 2.6 wird der eigenständige DSL-Entwurf detaillierter erklärt.

2.1 Aktivitäten in der Softwareentwicklung

Damit DSLs erfolgreich in der Softwareentwicklung verwendet werden können, müssen sie in den Softwareentwicklungsprozess eingebunden werden. In der derzeitigen Entwicklung von Softwaresystemen lassen sich grob die folgenden vier Aktivitäten identifizieren: Analyse, Entwurf, Implementierung, Verifikation/Validierung. Zusätzlich kann noch die Softwareauslieferung und -wartung betrachtet werden, die die Aktivitäten nach der Fertigstellung der Software bezeichnen. Die Analyse hat als Ziel, alle Anforderungen der vom zu erstellenden System mittelbar oder unmittelbar betroffenen Personen oder Institutionen zu erfassen. Der Entwurf nimmt diese Anforderungen als Grundlage und plant die Struktur der Software. Die Ergebnisse werden in der Implementierung verfeinert und in eine lauffähige Software umgesetzt. Die Verifikation/Validierung überprüft, ob die Ergebnisse der Analyse, des Entwurfs und der Implementierung mit den formulierten und den impliziten Anforderungen der vom zu erstellenden System mittelbar oder unmittelbar betroffenen Personen oder Institutionen übereinstimmen. Die Softwareauslieferung beschreibt den Prozess zur Installierung der Software auf den Zielsystemen. Die Softwarewartung fasst die Prozesse zusammen, die auftreten, wenn sich nach der Auslieferung der Software Änderungen an Anforderungen oder der technischen Infrastruktur ergeben.

Die beschriebenen Aktivitäten finden sich in allen gängigen Softwareentwicklungsprozessen wieder, wobei jeweils eine etwas andere Terminologie verwendet wird und der genaue Inhalt der einzelnen Aktivitäten variieren kann. Zusätzlich unterscheiden sich die Prozesse auch durch die Priorisierung der Aktivitäten und die Reihenfolge ihrer empfohlenen Ausführung. Zum Beispiel wird die Softwarewartung im Wasserfallmodell [Roy70] als Betrieb bezeichnet. Iterative Prozesse wie das Spiralmodell [Boe88] verwenden grundsätzlich dieselben Aktivitäten, führen diese aber innerhalb der Entwicklung der Software wiederholt aus. Das V-Modell [VMo97] dagegen teilt die Verifikation/Validierung in mehrere Aktivitäten auf, wobei jeweils die Sicherung der Qualität auf einer bestimmten Granularität betrachtet wird. Die Weiterentwicklung des V-Modells [RB08] erweitert den Prozess um

Aktivitäten während der Auftragsphase und die iterative Entwicklung. Der Unified Prozess [Kru95, Kru03] führt zusätzlich zu den Aktivitäten so genannte Phasen ein, in denen die unterschiedlichen Aktivitäten verschieden stark betont werden. Agile Methoden wie zum Beispiel [DS90, BA04] hingegen verschränken in Mikroiterationen die einzelnen Aktivitäten und nutzen dabei Hilfsmittel zur Evolution der Software, um mit sich ändernden Anforderungen umzugehen.

Im Folgenden wird das in Abbildung 2.1 gezeigte Vorgehensmodell genutzt, in dem die einzelnen Phasen iterativ ausgeführt werden. Es passt zu den meisten Vorgehensmodellen, insbesondere kann auch auf die dargestellte Iteration verzichtet werden. Das Ergebnis des Entwicklungsprozesses ist eine in die Wartung überführte Software. Diese kann dann wiederum bei größeren Änderungen der Anforderungen die Grundlage für eine weitere Softwareentwicklung bilden.

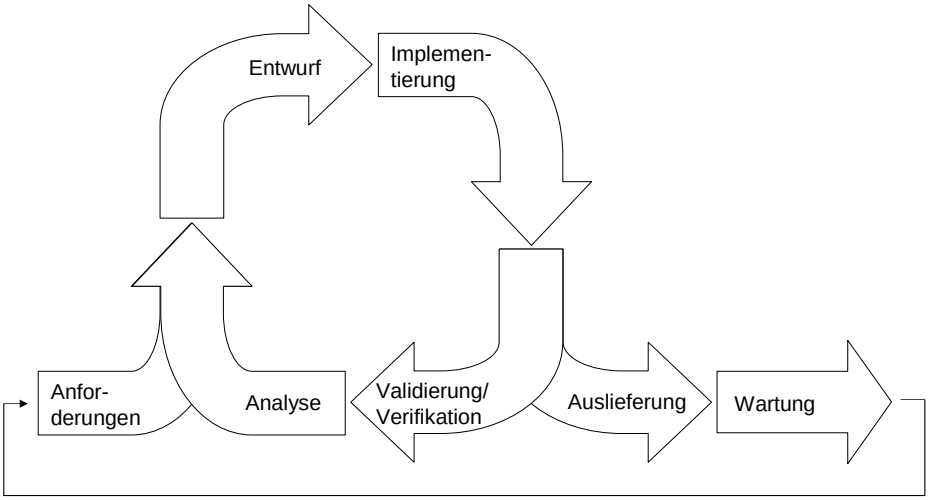


Abbildung 2.1: Allgemeines iteratives Vorgehensmodell

2.2 Einsatz von DSLs in der Softwareentwicklung

Im Folgenden wird beschrieben, wie domänenspezifische Sprachen innerhalb der einzelnen Aktivitäten verwendet werden können, um sie so erfolgreich und gewinnbringend in der Softwareentwicklung einzusetzen. Dabei werden die grundsätzlichen Anforderungen an solche DSLs formuliert und Beispiele vorgestellt. Zusätzlich wird erklärt, wie DSLs übergreifend eingesetzt werden können, so dass sie in mehreren Aktivitäten eine Rolle spielen.

Die Abbildung 2.2 stellt das allgemeine Vorgehensmodell bei der Verwendung von DSLs dar. In der Abbildung ist die Verwendung einer DSL als schwarzer Pfeil dargestellt. Eine geeignete DSL kann die übliche Vorgehensweise innerhalb einer Aktivität unterstützen, ergänzen oder ersetzen. Die übergreifende Verwendung von DSLs also zum Beispiel eine direkte Verbindung der beiden Aktivitäten Entwurf und Implementierung ist nicht dargestellt.

2.2.1 DSLs in der Analyse

In heute üblichen Softwareentwicklungsprozessen formuliert der Domänenexperte die Eigenschaften eines Softwaresystems meistens auf eine informelle Weise. In Zusammenarbeit

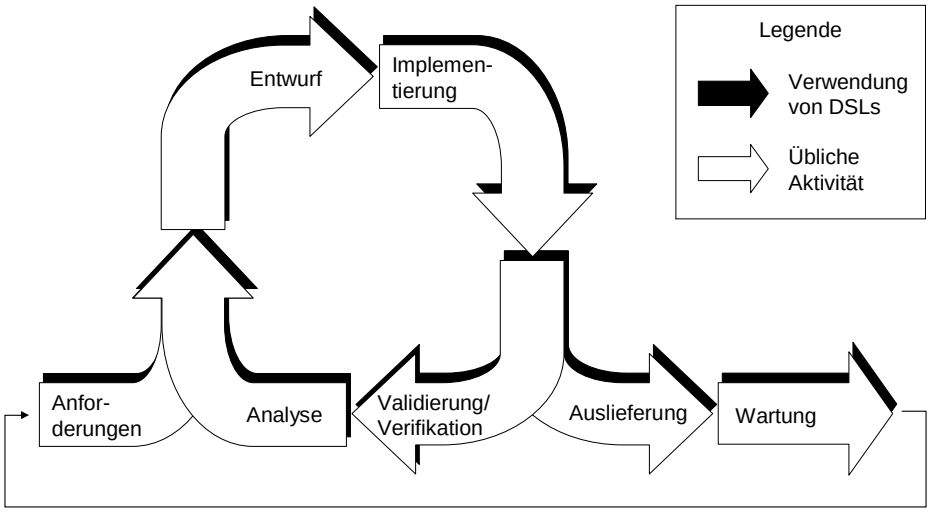


Abbildung 2.2: Allgemeines Vorgehensmodell mit DSLs

mit den Entwicklern werden diese dann präzisiert und ein entsprechendes System realisiert. Dieser mittelbare Einfluss von Experten ist, wie bereits beschrieben, hinderlich für eine Softwareentwicklung, die mit sich ändernden Anforderungen umgehen muss. DSLs ermöglichen die direkte Einbindung von Domänenexperten in den Entwicklungsprozess. Wird eine DSL zur Analyse genutzt, sollte sie die Möglichkeit bieten, die wesentlichen Eigenschaften eines Softwaresystems spezifizieren zu können.

Die in dieser Dissertation verwendete Fallstudie (Kapitel 10) und insbesondere die Ausführungen in [GHK+08a] zeigen, dass automotive Funktionsnetze zur Dokumentation baureihenübergreifenden funktionalen Wissens geeignet sind. Die Funktionsnetze bilden die Grundlage für die Ausgestaltung der Software-Architektur und eignen sich zur formalen Erfassung von Anforderungen.

2.2.2 DSLs im Entwurf

Der Entwurf dient zur Strukturierung der Software und zur frühzeitigen Analyse der entscheidenden Eigenschaften. Die dabei häufig verwendete universelle Modellierungssprache UML [OMG07b] beinhaltet verschiedene Teilsprachen zur Modellierung bestimmter Aspekte eines Softwaresystems. Die UML erlaubt die Individualisierung ihrer Notation, indem für spezifische Anwendungsgebiete so genannte Profile definiert werden. Ein Beispiel hierfür ist das MARTE-Profil [OMG08] zur Spezifikation echtzeitfähiger Anwendungen.

Alternativ kann eine DSL verwendet werden, die als Architekturbeschreibungssprache bezeichnet wird, wie zum Beispiel ACME [GMW97] oder Rapide [LKA+95]. Diese bieten jeweils Formalismen zur Beschreibung der Architektur eines Systems, wobei die genau zu spezifizierenden Informationen durch anschließende Analysen motiviert sind, die eine frühzeitige Abschätzung der zentralen Eigenschaften des Systems erlauben und somit helfen, den Entwurf zu verbessern. In der automotiven Domäne etablieren sich gerade Beschreibungssprachen, um automotive Softwaresysteme besser entwerfen und analysieren zu können. Wichtige Ansätze sind dabei die Autosar Softwarearchitektursprache [AUT] und die East-ADL [EAS], die innerhalb des ITEA EAST-EEA-Projekts entstanden ist. Die durch die Society of Automobile Engineers (SAE) standardisierte Architecture Analysis and Design Language (AADL) [SAE06] beschreibt die modellbasierte Entwicklung eingebetteter Systeme im Automobil und in der Avionik.

2.2.3 DSLs in der Implementierung

Aus einer mehr technischen Sicht heraus gibt es innerhalb eines Softwareprojekts oft Anteile, die durch sich wiederholende standardisierte Programmierung entstehen. Dieser Quellcode folgt grundlegenden Prinzipien, das heißt die eigentlich notwendigen Informationen zur Erstellung der Software sind deutlich geringer, als der entstehende Quellcode vermuten lässt. Das Problem liegt oft darin, dass die verwendete Programmiersprache keine kompaktere Formulierung erlaubt und eine Auslagerung in APIs teilweise nicht möglich ist.

Hier verhindert der Einsatz von DSLs sich wiederholende und daher fehlerträchtige Implementierungsarbeit, indem der Quellcode aus kompakten DSL-Modellen generiert wird. Durch die Übersichtlichkeit werden Fehler vermieden, die Entwicklungszeit verringert und zusätzlich die Wartbarkeit sowie die Evolutionsfähigkeit von Software verbessert [DK98, DKV00]. Ein weiterer Vorteil ist, dass ein Teil der Entwicklungszeit in die Generatoren fließt. Ändern sich zu einem späteren Zeitpunkt die Anforderungen an diesen Systemteil, müssen meistens nur die DSL-Modelle oder der Generator angepasst werden, selten beides. Dadurch kann in einer späten Phase eines Projekts, wo sich Änderungen der Anforderungen sonst meistens fatal auswirken, unter Umständen doch eine strukturierte Evolution erreicht werden.

Die Implementierung ist die Aktivität innerhalb der Softwareentwicklung, für die es die größte Anzahl an DSLs gibt. Die folgende Auflistung zeigt exemplarisch einige Anwendungsgebiete, in denen häufig DSLs eingesetzt werden oder eingesetzt werden könnten:

Anwendungsoberflächen werden durch grundlegende Informationen wie Position und Art der Eingabefelder und Beschriftungen definiert. Zur Implementierung einer solchen Oberfläche ist trotz existierender APIs noch eine Vielzahl an Quellcodezeilen notwendig, so dass sich hier (grafische) Werkzeuge mit einer Codegenerierung etabliert haben.

Reguläre Ausdrücke stellen eine kompakte Form der Verarbeitung von Zeichenketten dar. In vielen Programmiersprachen stehen APIs zur Formulierung von regulären Ausdrücken zur Verfügung, jedoch müssen diese oft als String formuliert werden, was gewisse Einschränkungen mit sich bringt. Diese Einschränkungen sind eine beliebte Fehlerquelle, die durch den Einsatz einer DSL vermieden werden kann.

Datenbankanfragen werden häufig in der Implementierung betriebswirtschaftlicher Informationssysteme eingesetzt. Aktuelle Entwicklungen in Programmiersprachen wie [MBB06] integrieren daher eine DSL (einen Dialekt von SQL) fest in eine GPL.

Datenmodelle wie Klassendiagramme oder Entity-Relationship-Modelle lassen kanonisch in Quellcode umsetzten. Dadurch lässt sich die Struktur eines Softwaresystem auf eine übersichtliche Art und Weise planen.

2.2.4 DSLs zur Validierung/Verifikation

Die Validierung und Verifikation von Softwaresystemen dient zur Qualitätssicherung innerhalb des Softwareentwicklungsprozesses. Eine erfolgreich eingesetzte Methode ist das Software-Testen [Bin00], wobei gezielt exemplarische Situationen für die Software oder Teile davon hergestellt werden und dann die Reaktion der Software auf Stimuli mit definierten Sollwerten verglichen wird. Diese Technik kann sowohl zur Verifikation formulierter Anforderungen genutzt werden als auch zur Validierung, wenn Anwender die Tests selbst schreiben. DSLs eignen sich zur Testfallmodellierung, da dem Entwickler spezifische

Konstrukte zur Verfügung gestellt werden können, die die Formulierung von kompakten Testfällen ermöglichen.

Ein Beispiel für eine solche DSL ist die Testing and Test Control Notation Version 3 (TTCN-3) [GHR+03]. TTCN-3 ist eine standardisierte Sprache zur Erstellung von Testfällen, wobei spezifische Programmkonstrukte wie parametrisierbare Vergleichsoperatoren, Timer und nebenläufige Ausführung nativ zur Verfügung gestellt werden. Als eine mögliche Alternative wird in [Rum04a] eine Form der Testfallmodellierung mit UML-Objekt- und Sequenzdiagrammen beschrieben.

Eine weitere Möglichkeit zur Validierung der Software stellt die Spezifikation von Invarianten dar. Diese können mit Hilfe von DSLs wie Alloy [Jac02] oder OCL [OMG06b] spezifiziert werden. Model checking [CGP00] ist eine Technik, die bestimmte (temporallogische) Invarianten einer Spezifikation gegenüber einem Modell überprüft oder gezielt deren Invalidierung zeigen kann. Die verschiedenen Systeme verwenden jeweils eine bestimmte, meistens automatenbasierte DSL zur Beschreibung des Modells und eine weitere, oft logikbasierte DSL für die Spezifikation.

2.2.5 DSLs zur Softwareauslieferung

Die Auslieferung verteilter Anwendungen umfasst mehr als nur ein reines Installationsprogramm. Dabei sind häufig der Einsatz von Deploymentdeskriptoren und die Verwaltung der Serverzugänge notwendig.

In der aktuellen Entwicklungsumgebung von Microsoft (Visual Studio 2008) ist eine an UML-Verteilungdiagramme angelehnte DSL integriert, die es erlaubt, Softwarekomponenten auf physikalische Ressourcen zu verteilen. Bei einer Modifikation der Software kann so der Auslieferungsprozess automatisiert ausgeführt werden.

2.2.6 DSLs zur Softwarewartung

Mit zunehmender Komplexität und wirtschaftlicher Bedeutung existierender Anwendungen werden komplette Neuentwicklungen von Software seltener und die Evolution bestehender Anwendungen auf Basis veränderter Anforderungen häufiger. Zusätzlich ist die reine Wartung, also die Konservierung der Funktionalität bei Austausch der verwendeten Technologie oftmals aufgrund veralteter Hardware oder verwendeter Software-Plattformen, eine wichtige Disziplin innerhalb der Softwaretechnik.

In [KR05, KR06a] wird die Evolution von Software-Architekturen auf der Basis einer expliziten Software-Architekturbeschreibung in Form einer kompakten DSL formuliert. Der Einsatz von DSL-Technologien in der Implementierung vereinfacht oftmals auch die Evolution, weil somit Domänenwissen und technische Realisierung in DSL-Modellen und Generatoren getrennt voneinander entwickelt werden. In diesem Fall kann das explizit verfügbare Domänenwissen in Form der DSL-Modelle wieder verwendet und der Generator an die neuen Gegebenheiten angepasst werden. Ein aufwändiges Reverse-Engineering der impliziten Geschäftslogik in Softwaresystemen kann dann entfallen.

2.2.7 Übergreifender Einsatz

Die einzelnen bisher dargestellten Aktivitäten treten bei jedem Softwareentwicklungsprozess auf. Das heißt jedoch nicht zwangsläufig, dass die Aktivitäten nicht eng miteinander verzahnt sein können oder sogar zu einem Entwicklungsschritt zusammengefasst werden. Die modellbasierte Entwicklung auf der Basis von DSLs ermöglicht dieses Vorgehen im Sinne einer ausführbaren Spezifikation. Dabei wird eine DSL verwendet, die aus der Domäne

entnommen wird und dort bereits genutzt wurde, um ein Softwaresystem zu modellieren. Diese Modelle werden nun verwendet, um automatisiert ein lauffähiges System zu erzeugen. Dadurch wird ebenfalls die Verifikation vereinfacht, weil die automatische Codegenerierung getrennt qualitätsgesichert oder zumindest automatisiert getestet werden kann [Stü06]. Die Validierung muss jedoch weiterhin durchgeführt werden.

Ein Beispiel für ein solches Vorgehen ist die Verwendung der DSL Matlab Simulink [Sim] zur Implementierung automotiver Kontrollsysteme. Die dabei verwendete Notation ist den Domänenexperten vertraut, weil sie sich an der üblichen Art zur Modellierung von Kontrollsystemen orientiert. Dabei wird gleichzeitig die Struktur der Software entworfen und automatisiert die Implementierung erzeugt. Die automatische Codegenerierung übernimmt eine Zeit- und Wertdiskretisierung und erzeugt Code für verschiedene Plattformen.

2.3 Entwicklung einer DSL

In dieser Arbeit umfasst die Entwicklung einer DSL grundsätzlich die Definition der Sprache und die Implementierung dazu passender Werkzeuge. Diese Entwicklung unterscheidet sich nicht grundsätzlich von der Erstellung anderer Software, so dass die üblichen Softwareentwicklungsprozesse in angepasster Form verwendet werden können. Dabei umfassen die einzelnen Aktivitäten mehrere Aufgaben, die spezifisch für die DSL-Entwicklung sind und daher im Folgenden dargestellt werden.

2.3.1 Analyse

Vor der Festlegung der genauen Ausgestaltung einer DSL und ihrer technischen Realisierung ist es wichtig, zunächst zu klären, wie sich die DSL in den Entwicklungsprozess der übergeordneten Software einfügen soll. Dabei ist speziell festzulegen, für welche Aktivität die DSL eingesetzt werden soll. Bei der Übernahme von Notationen aus der Domäne ist zu klären, welche zusätzlichen Möglichkeiten sich durch eine automatische Verarbeitung der Modelle ergeben. Grundsätzlich ergibt sich jedoch immer durch die Formalisierung der Vorteil, dass die innere Konsistenz der Modelle automatisch gesichert werden kann.

Die Anforderungen an eine DSL umfassen die gewünschten Eigenschaften der Sprache und ihrer unterstützenden Software. Das wesentliche Ziel der Anforderungsanalyse ist die Identifikation eines geeigneten Einsatzszenarios für die DSL. Beim generativen Einsatz von DSLs müssen Softwareteile, wie Subsysteme oder existierende Querschnittsfunktionalitäten identifiziert werden, die durch sie beschrieben werden sollen. Indikatoren für geeignete Stellen sind Anteile, die möglicherweise in mehreren Projekten stets wiederkehren und deren Implementierung reguläre Strukturen aufweist. Beim analytischen Einsatz müssen genau die Eigenschaften festgelegt werden, die durch die DSL analysiert werden sollen.

Im Sinne einer agilen Entwicklung müssen die Anforderungen nicht explizit formuliert werden, sondern können zum Beispiel auch aus exemplarischem Quellcode bestehen, der aus einem ebenfalls gegebenen Modell erzeugt werden soll. In Kapitel 8 wird eine Methodik vorgestellt, die schrittweise aus exemplarischem Code einen Generator und eine dazu passende DSL entwickelt.

Die Domänenanalyse ist die Aktivität, Objekte und Operationen einer Klasse von ähnlichen Systemen in einer bestimmten Domäne zu identifizieren [Nei80]. Dazu können verschiedene Analyseverfahren wie die Domain Analysis and Reuse Environment (DARE) [FPF95], die Family-Oriented Abstractions, Specifications, and Translation (FAST) [WL99] und die Feature-Oriented Domain Analysis (FODA) [KCH$^+$90] verwendet werden. Dabei wird ein Domänenmodell erstellt, das aus den folgenden Elementen besteht [MHS05]:

- Eine Definition der Domäne, die den Anwendungsbereich festlegt.
- Eine Domänenterminologie, die die wichtigsten Begriffe der Domäne enthält.
- Eine Beschreibung der Konzepte der Domäne.
- Feature-Modelle, die die Gemeinsamkeiten und Unterschiede der Domänenkonzepte und ihre Abhängigkeiten untereinander beschreiben.

Die Ergebnisse einer Domänenanalyse sind hilfreich, um eine DSL zu entwickeln oder existierende DSLs zu identifizieren. Sie bieten jedoch keine Unterstützung zur konkreten Ausgestaltung einer DSL. Die Verfahren sind stark auf die Ausgestaltung einer Software-Produktlinie ausgerichtet und fokussieren sich daher nur auf einen kleinen Teil der möglichen DSLs, die relativ grob-granular die Variabilität einer Familie von Produkten beschreiben. Insbesondere ist die Verwendung von Feature-Diagrammen als Grundlage für eine DSL wenig hilfreich. Sie stellen zwar prinzipiell den variablen Teil der Produkte dar und strukturieren somit die Informationen, die die DSL-Modelle beinhalten müssen, sind aber wenig geeignet, um komplexe Zusammenhänge zwischen Konzepten zu beschreiben. Durch fehlende Elemente wie zum Beispiel Kardinalitäten können sie nur schwer konstruktiv in eine DSL entwickelt werden. Eher geeignet sind Klassendiagramme, die die konzeptuellen Zusammenhänge der Domänenkonzepte beschreiben und im Entwurf in die abstrakte Syntax einer DSL weiterentwickelt werden können.

Eine effiziente Form der Entwicklung von DSLs ist die Wiederverwendung existierender DSL-Definitionen aus einer Sprachbibliothek. Die dort abgelegten Eigenschaften der DSLs können mit den Anforderungen verglichen werden, um so zu ermitteln, ob eine passende DSL für die Problemlösung bereits existiert. Wird dabei eine ähnliche DSL identifiziert, sollten die Unterschiede zwischen den Anforderungen und den Eigenschaften analysiert und dokumentiert werden. Dieses erleichtert im Entwurf die Wiederverwendung und Weiterentwicklung existierender Sprachen.

2.3.2 Entwurf

Der Entwurf einer DSL legt die genaue Struktur der Sprache fest. Dabei werden die Elemente der Domänenanalyse in eine nutzbare DSL überführt, die von einem Entwickler verwendet werden kann. Wurden in der Analyse Ähnlichkeiten zu einer existierenden DSL gefunden, können die in [MHS05] identifizierten Entwurfsmuster Spezialisierung und Erweiterung zur Modifikation einer Sprache oder Huckepack zur Verwendung von Sprachteilen benutzt werden. Somit wird der Entwurfsaufwand reduziert und es etablieren sich langfristig qualitativ hochwertige DSLs.

Grundsätzlich ist der Sprachentwurf eine komplexe Angelegenheit, die eine enge Zusammenarbeit mit den Domänenexperten erfordert. Hilfreich hierfür sind allgemeine Prinzipien zum Sprachentwurf von Programmiersprachen wie zum Beispiel [Hoa73] und [Wir74], Hinweise für Modellierungssprachen [POB99] sowie die dokumentierten Erfahrungen von David Wile beim DSL-Entwurf [Wil04].

Die Ausgestaltung der automatischen Verarbeitung von DSL-Modellen umfasst die Grundentscheidung, ob ein generatives Vorgehen, eine Interpretation oder eine Analyse der DSL gewünscht wird. Bei der Generierung muss zusätzlich entschieden werden, welche Teile durch die Generierung entstehen sollen und welche Teile in einer Laufzeitumgebung allgemeingültig programmiert werden können. Soll die DSL dabei so eingesetzt werden, dass aus ihr Software erzeugt wird, die direkt in das Produkt integriert wird, so sollten

vorab Schnittstellen entworfen werden, die die generierten Klassen implementieren müssen. Dadurch kann die Integration in die handcodierte Software bereits ohne die generierten Artfakte geplant werden. Ist die Generierung anders organisiert, weil sich beispielsweise die Struktur der Schnittstellen aus dem Inhalt der Modelle ergibt und nicht universell ist, muss dieser Ableitungsprozess bereits im Entwurf geplant und dokumentiert werden.

Die Ausgestaltung der DSL bringt es mit sich, dass neben der kontextfreien Syntax der Sprache noch weitere Einschränkungen notwendig sind, um nur konsistente Modelle zuzulassen und den Entwickler auf konzeptuelle Fehler hinweisen zu können. Die Kontextbedingungen ergeben sich aus der Kenntnis der Domäne oder aus einer formalen Semantikdefinition der DSL. Hier sollten für die Modelle, die keine Bedeutung haben, entsprechende Kontextbedingungen formuliert werden, die syntaktisch überprüft werden können und so diese Modelle von der weiteren Verarbeitung ausschließen.

Als weiterer Schritt ist beim Entwurf einer DSL die gewünschte Editorunterstützung und eventuelle horizontale Modelltransformationen wie Refactorings für die DSL-Modelle zu planen. Diese Unterstützung der Entwickler ist ein wesentlicher Erfolgsfaktor für die Etablierung der DSL und sollte daher nicht unterschätzt werden. Einige Elemente wie die Editorunterstützung können im Sinne einer übergreifenden Entwicklung weitestgehend direkt aus der Sprachdefinition abgeleitet werden. Dazu existieren Codegeneratoren, die durch spezielle DSLs innerhalb der Entwicklung parametrisiert werden. Dieses zeigt insbesondere, dass die DSL-Entwicklung einer normalen Softwareentwicklung gleicht und diese daher vom Einsatz von DSLs profitieren kann.

2.3.3 Implementierung

Für die technische Realisierung eines DSL-Werkzeugs gibt es verschiedene Möglichkeiten, die in Abschnitt 2.5 ausführlich dargestellt werden. Dabei muss je nach Kontext entschieden werden, welche Variante für eine konkrete DSL geeignet ist. Die gewählte Methode bestimmt dann die genauen Arbeitsschritte innerhalb der Implementierung. In dieser Dissertation wird sich vor allem auf den textbasierten eigenständigen DSL-Entwurf konzentriert. Wurden beim Entwurf der Sprache eine andere Sprache als Grundlage identifiziert oder Teile von existierenden DSLs verwendet, können die im MontiCore-Framework vorhandenen Modularitätsmechanismen (vgl. Kapitel 4) genutzt werden, um eine DSL aus einfachen Bausteinen zusammenzusetzen oder aber Varianten von existierenden DSLs zu bilden.

Die Implementierung einer DSL umfasst neben der eigentlichen Sprachdefinition weitere Artefakte. Bei generativen Ansätzen ist die Codierung der Codegenerierung und der Laufzeitumgebung notwendig. Bei einem interpretativen oder analytischen Vorgehen beschränkt sich die Implementierung auf den Interpreter. Zusätzlich müssen weitere Elemente wie benötigte Modelltransformationen und die Editorunterstützung codiert werden, sofern sie nicht im Sinne einer übergreifenden Entwicklung entstanden sind.

Die Implementierung umfasst auch die Codierung der Kontextbedingungen. Es kann bei der Umsetzung der Codegenerierung auch zur Identifikation weiterer Kontextbedingungen kommen, die meistens technischer Natur sind und weniger in der Semantik der Sprache begründet sind. Beispiele hierfür sind reservierte Wörter in der als Zielsprache verwendeten GPL, die dann auch nicht in der DSL verwendet werden dürfen, wenn die Bezeichner der DSL direkt in Bezeichner der GPL umgesetzt werden.

2.3.4 Validierung/Verifikation

Die Validierung und Verifikation von DSL-Werkzeugen dient zur Qualitätssicherung innerhalb der Softwareentwicklung. Dadurch kann gesichert werden, dass die vom Entwickler erstellten DSL-Modelle, sofern sie zu den fachlichen Anforderungen passen, auch korrekt in ein Softwaresystem übersetzt werden.

Für die Validierung der DSL sollten positive und negative Testfälle verwendet werden. Bei generativen Ansätzen ist eine erfolgreiche Kompilierung der erzeugten Ausgabedateien ein erster positiver Test. Dieser sollte, sofern dieses möglich ist, durch die Ausführung des generierten Quellcodes und die Überprüfung bestimmter dynamischer Eigenschaften, die sich aus der Domäne ergeben, ergänzt werden. Diese Art der Überprüfung ist oft robust gegenüber Änderungen der Codeerzeugung, da nur die funktionalen Eigenschaften des entstehenden Quellcodes überprüft werden, und nicht die genaue Form der Implementierung. In negativen Testfällen sollte insbesondere geprüft werden, ob die Kontextbedingungen für diese Instanzen fehlschlagen und so der Entwickler eine entsprechende Fehlermeldung bekommt. Ein ausschließliches Fehlschlagen der Codegenerierung oder gar ein Erzeugen von nicht übersetzbarem Quellcode sind zu vermeiden.

Zur Beurteilung der Testgüte können Abdeckungskriterien [Pur72, Läm01b] auf der Sprachdefinition verwendet werden. Diese können auch konstruktiv eingesetzt werden, indem sie als Instanzgenerator verwendet werden. Da die generierten Instanzen oftmals jedoch Kontextbedingungen verletzen und die Erzeugung von Instanzen, die von menschlichen Testern als sinnvoll eingestuft werden, eine schwierige Aufgabe ist, sollte die automatische Erzeugung nur als Ergänzung manuell erstellter Instanzen verstanden werden.

Als eine Alternative für die bisher dargestellten funktionalen Tests, bei denen die Abdeckung bezüglich der DSL das maßgebende Gütekriterium ist, können natürlich auch wie bei jeder Softwareentwicklung Unittests verwendet werden. Deren Güte wird dann durch die Abdeckung der Implementierung bestimmt. Diese eignen sich insbesondere für die Überprüfung der Laufzeitumgebung und weniger für die eigentliche Codegenerierung, weil hier oft aufwändige Zeichenkettenvergleiche notwendig wären und diese wenig robust gegen unwesentliche Änderungen in der Codeerzeugung sind.

Ausführbare DSLs ermöglichen das Testen des Codegenerators, sofern die DSL unabhängig von der Generierung interpretiert werden kann. In [Stü06] wird dieses Vorgehen für Matlab Simulink verwendet, um Codegeneratoren für verschiedene Plattformen automatisiert testen zu können. Dazu wird die Kosimulation des Interpreters und der durch die Codegenerierung entstandenen Software mit einer automatischen Instanzerzeugung kombiniert.

2.3.5 Auslieferung

Die Auslieferung eines Generators umfasst die Sprachdefinition, den Generator, die Laufzeitumgebung und eine geeignete Dokumentation. Dabei ist insbesondere eine Versionierung notwendig, wenn die Generatoren fortlaufend weiterentwickelt und in unterschiedlichen Projekten eingesetzt werden.

Die Integration in andere Projekte erfolgt dabei idealerweise über Build-Werkzeuge, die bereits ein Abhängigkeitsmanagement integriert haben, wie Maven [Mav] oder Ant [Ant] (mit der Erweiterung Ivy [Ivy]). So kann die komplexe Auslieferung der benötigten Generatorversion automatisiert werden, und die Entwickler verwenden stets die aktuelle Version der Software.

2.3.6 Wartung

Die Wartung einer DSL bei gleicher Funktionalität ist typischerweise eine relativ einfache Aufgabe, da Generatoren selten plattformspezifischen Code enthalten oder von komplexen Frameworks abhängen und idealerweise auf der Kommandozeile lauffähig sind. Dadurch entfallen Gründe, die technische Plattform wechseln zu müssen.

Komplizierter wird die Situation, wenn Generatoren im GenVoca-Prinzip [BLS98] miteinander kombiniert sind und sich abhängige Generatoren ändern. Eine Anpassung kann dann notwendig sein, um die neue Version zu unterstützen und so die eigene Funktionalität weiter bereitzustellen.

Ein spezielles Problem bei der Evolution von DSLs ist die oft gewünschte Koevolution von Modellen und Generatoren, so dass bei der Veränderung der Sprache auch die bisher existierenden Instanzen automatisch verändert werden. Dieses Vorgehen wurde bisher in ganz engen Grenzen demonstriert [PJ07a, PJ07b], wobei immer beachtet werden muss, dass eine automatische Evolution nur dann möglich ist, wenn sich die Ausdrucksmächtigkeit oder die Semantik der DSL nicht grundsätzlich verändert. Ist eine automatisierte Koevolution nicht möglich, ist es hilfreich, die Modelle zusammen mit den Generatoren zu versionieren. Innerhalb einer Produktentwicklung sollten unterschiedliche Generatorversionen koexistieren können und die Modelle schrittweise migriert werden.

2.4 Gekoppelte agile Entwicklung einer DSL

Der vorherige Abschnitt hat die Entwicklung einer DSL in einem allgemeinen iterativen Entwicklungsprozess dargestellt. Dabei kann nach der Auslieferung die DSL für die Softwareentwicklung eingesetzt werden. Dieser Abschnitt beschreibt eine enge Kopplung der Entwicklung und des Einsatzes einer DSL. Die Abbildung 2.3 zeigt die eng gekoppelte Entwicklung einer DSL mit der Entwicklung eines Softwaresystems. Dabei wird direkt bei der Entwicklung des Softwaresystems festgelegt, dass einige Teile der Software oder des Entwicklungsprozesses allgemein vom Einsatz einer DSL profitieren können. Dazu werden die Anforderungen an die DSL festgelegt und eine von der Entwicklung des Softwaresystems unabhängige Entwicklung der DSL begonnen.

Die beiden Entwicklungsprozesse können auf zwei Arten miteinander gekoppelt sein:

- Wurde innerhalb der DSL-Entwicklung in möglicherweise mehreren Iterationen ein gewisser Funktionsumfang realisiert, kann eine Version dieser DSL ausgeliefert und, wie in Abschnitt 2.2 gezeigt, in der Entwicklung eingesetzt werden. Bei dieser Form der Kopplung ist ein Buildwerkzeug mit integriertem Abhängigkeitsmanagement besonders geeignet, weil so die publizierten Versionen der DSL leicht in die Softwareentwicklung übernommen werden können. Die Modelle müssen beim Versionswechsel an die verwendete Version der DSL angepasst werden.

- Bei der kontinuierlichen Integration der DSL und des Softwareprojekts wird die DSL wie das Softwareprojekt ohne explizite Versionierung kontinuierlich weiterentwickelt. Innerhalb der Entwicklung von MontiCore haben sich spezielle in Ant [Ant] realisierte Buildskripte bewährt, die Änderungen an der DSL oder der Software auswerten und nur die notwendigen Übersetzungen und Generierungsschritte ausführen. Die Modelle müssen fortlaufend an die DSL angepasst werden.

Die agile Entwicklung einer DSL zeichnet sich vor allem durch sehr kurze Iterationszyklen aus. Dabei werden die verschiedenen beteiligten Artefakte wie die Sprachdefinition und

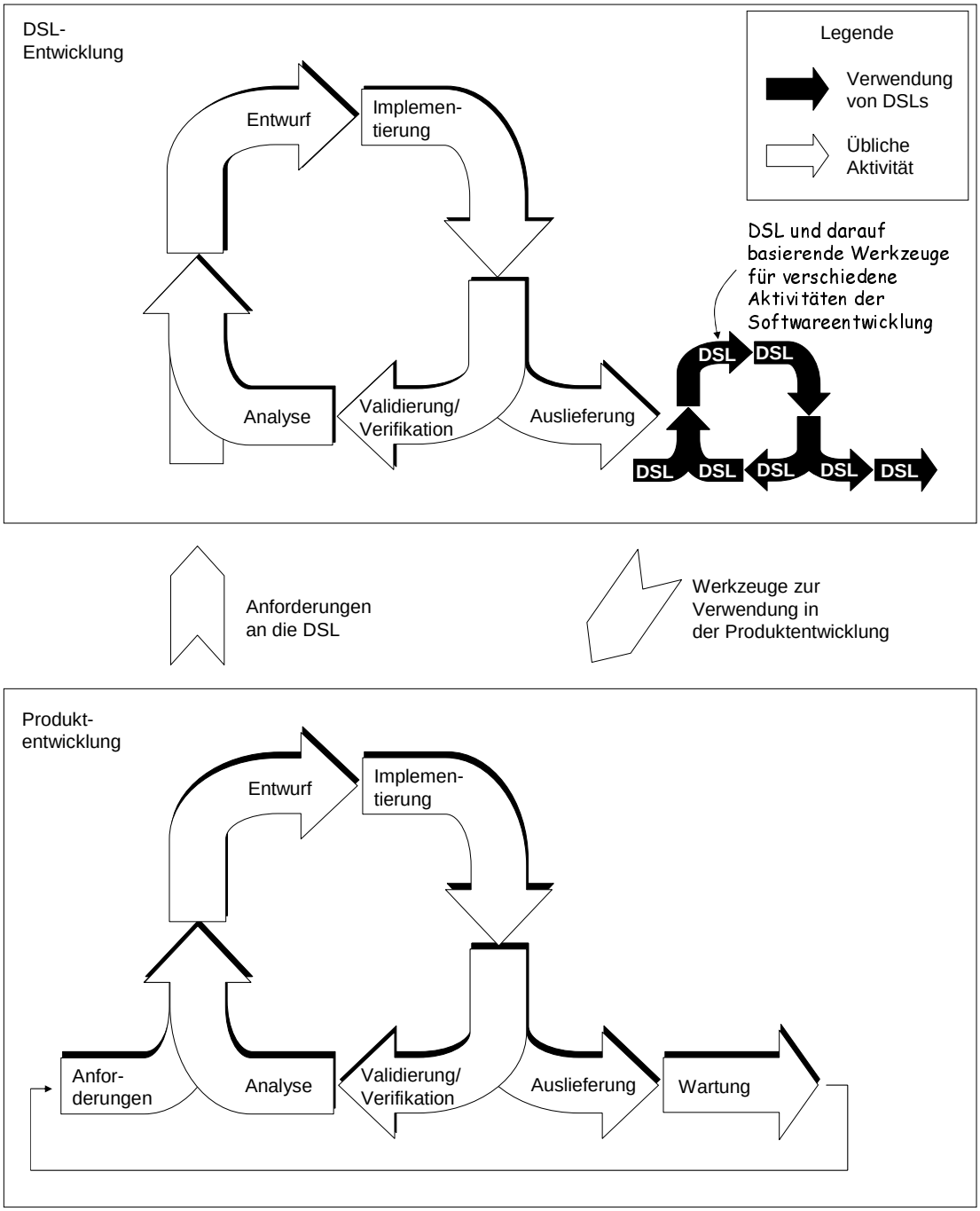


Abbildung 2.3: Agile Kopplung der DSL-Entwicklung mit der Softwareentwicklung

die Codegenerierung fortlaufend weiterentwickelt und getestet. Wie bereits in [KRV06] dargestellt, lassen sich dabei die folgenden Rollen innerhalb der Entwicklung einer DSL identifizieren. Die beschriebenen Rollen geben an, welche Personen mit welchen Aufgabengebieten an der Entwicklung und am Einsatz einer DSL zur Softwareerstellung mitwirken, wobei eine Rolle auch von mehreren Personen erfüllt werden oder eine Person mehrere Rollen ausfüllen kann:

- Ein *Domänenexperte* legt innerhalb der Analyse der DSL-Entwicklung die Ausgestaltung der DSL fest. Dabei leistet er wichtige Vorarbeit für die anderen Rollen, indem er vorhandene Notationen der Domäne zusammenträgt und existierende Frameworks und Bibliotheken der Domäne auf ihre Eignung prüft.

- Ein *Sprachentwickler* definiert oder erweitert eine DSL bezüglich der Bedürfnisse und Anforderungen des Produktentwicklers und den Festlegungen des Domänenexperten im Entwurf der DSL.

- Ein *Werkzeugentwickler* schreibt innerhalb der Implementierung der DSL einen Codegenerator, was sowohl die Erzeugung von Produktions- und Testcode als auch Analysen des Inhalts und der Qualität beinhaltet. Er testet seine Implementierung in der Validierung/Verfikation der DSL-Entwicklung. Zusätzlich integriert er neu entwickelte oder wieder verwendete Sprachverarbeitungskomponenten und Generatoren zu neuen Werkzeugen, die bei der Entwicklung der Software eingesetzt werden.

- Ein *Bibliotheksentwickler* entwirft Software-Komponenten oder Bibliotheken für das Produkt und vereinfacht daher den Codegenerator, weil konstante wieder verwendbare Softwareteile nicht generiert werden müssen, sondern nur durch die DSL verschaltet oder konfiguriert werden müssen. Daher ist diese Rolle eng verwandt mit dem Werkzeugentwickler, erfordert aber mehr Wissen über das Anwendungsgebiet. Ein Ziel solcher Bibliotheken ist die Kapselung detaillierten Domänenwissens und die Konzeption von Schnittstellen, die ausreichend für die Bedürfnisse einer Codegenerierung sind. Die so entstehende Laufzeitumgebung fasst die Teile der Software zusammen, die zur Laufzeit der Produkte ausgeführt werden. Sie muss jedoch zusammen mit der DSL entwicklet werden, weil die Komponenten unabhängig von einem konkreten Produkt konzipiert werden. Nichtsdestotrotz können die erste Version einer solcher Komponenten eine konkreten Produkt entstammen und entsprechende verallgemeinert Teil der Laufzeitumgebung werden.

Die Entwicklung des Produkts wird hier nicht stärker differenziert betrachtet:

- Ein *Produktentwickler* verwendet die bereitgestellten Werkzeuge in allen Aktivitäten innerhalb der Produktentwicklung. Die Hauptanwendung ist dabei die Spezifikation von Domänenwissen in DSL-Modellen. Zusätzlich erledigt er die übrigen Aufgaben bei der Entwicklung des Produkts mit konventionellen Techniken.

2.5 Technische Realisierung einer DSL

Unabhängig vom gewählten Entwicklungsprozess gibt es verschiedene technische Möglichkeiten zur Implementierung einer DSL. Die wichtigsten in der Literatur zu findenden Möglichkeiten sind [Wil01, MHS05]:

Makrosysteme realisieren einfache DSLs, die durch einen Preprozessor auf eine GPL umgesetzt werden.

Eingebettete DSLs bezeichnen Spracherweiterungen von GPLs (die in diesem Zusammenhang als Hostsprache bezeichnet werden), die zusätzliche kompakte Formulierungen bieten, jedoch die Grundstruktur der GPL erhalten. Die Elemente der DSL werden dabei auf die Konstrukte der GPL wie zum Beispiel API-Aufrufe abgebildet.

Erweiterbare Compiler erlauben die Erweiterung von GPLs ähnlich zu eingebetteten DSLs, wobei jedoch komplexe Codegenerierungen eingebunden werden können, da keine direkte Reduktion auf die GPL notwendig ist.

Erweiterbare Modellierungssprachen wie die UML mit ihrer Profilbildung erlauben in begrenztem Umfang, domänenspezifische Varianten vorhandener Elemente durch Spezialisierung zum Sprachumfang hinzuzufügen.

Eigenständige DSL-Definitionen bezeichnen die Realisierung von DSLs, ohne dass eine direkte vorhandene Programmiersprache als Umgebung genutzt wird.

Die Methoden besitzen jeweils verschiedene Vor- und Nachteile, die bei der Auswahl einer Methodik zur Realisierung einer DSL gegeneinander abgewogen werden müssen, was insbesondere den Aufwand für die Realisierung mit einschließt. In der vorliegenden Arbeit liegt der Fokus auf der eigenständigen DSL-Definition. Die wesentlichen Gründe hierfür sind:

- Die eigenständige DSL-Definition erlaubt völlige Freiheit in der Gestaltung der Sprache und so eine optimale Annäherung an die Konzepte der Domäne. Dies lässt sich mit Profilbildung, eingebetteten DSLs oder erweiterbaren Compilern oft nur schwer erreichen, weil meistens die Elemente der Basissprache noch zur Verfügung stehen und die Syntax sich nur bedingt anpassen lässt.

- Eingebettete DSLs werden direkt in eine ausführbare Form innerhalb einer GPL überführt. Dadurch ergeben sich verschiedene Einschränkungen, die einen DSL-Einsatz erschweren. Erstens ist nur eine einzige Verwendungsform möglich und die Modelle können nicht, wie in Kapitel 1 dargestellt, die Grundlage für mehrere Codegenerierungen sein. Zweitens ist eine Erzeugung von Dokumentation und Analysen auf diesem Weg kaum möglich, da sich diese nicht als ausführbarer Programmcode darstellen lassen. Drittens ist die Programmierung komplexer Codegenerierungen, deren Ergebnis sich aus mehreren Dateien der Hostsprache zusammensetzt, meistens nur sehr umständlich zu realisieren.

- Die Evolution eines Systems, das mit eingebetteten DSLs realisiert ist, beinhaltet keine klare Trennung von Domänenmodellen und technischer Realisierung innerhalb des Produkts im Sinne einer Trennung in plattformspezifische und plattformunabhängige Anteile (vergleiche auch [OMG03]). Dadurch lässt sich beispielsweise die Zielplattform oder Hostsprache nicht einfach austauschen, da die Modelle meistens eine Mischung aus Hostsprache und DSL sind.

- Die Verwendung von Makros eignet sich nicht für die Implementierung komplexer Codegenerierungen und bietet keine Analyse vor der Verarbeitung, um den Entwickler mit erklärenden Fehlermeldungen bei fehlerhaften Eingaben zu unterstützen.

- Die Realisierung zusammengesetzter Modellierungssprachen ist mit eingebetteten DSLs kaum möglich, weil keine geeignete Hostsprache verfügbar ist. Sollen zum Beispiel Klassendiagramme um Invarianten der OCL ergänzt werden, können diese nicht auf Elemente des Klassendiagramms zurückgeführt werden, weil die Invarianten genau die Defizite in der Ausdrucksmächtigkeit der Klassendiagramme kompensieren sollen. Die Rückführung ist aber ein essentielles Merkmal eingebetteter DSLs.

2.6 Realisierung einer eigenständigen DSL

Die abstrakte Syntax einer DSL beschreibt deren strukturelle Essenz und bildet somit das zentrale Element in der Realisierung einer eigenständigen DSL. Die abstrakte Syntax lässt sich auf verschiedene Arten beschreiben, wobei graphenbasierte und grammatikbasierte Formalismen am häufigsten verwendet werden. Graphenbasierte Sprachformalismen erlauben, dass die zulässigen Instanzen auch eine Graphenstruktur haben. Es gibt hier eine Menge an konkurrierenden Ansätzen, die sich nur in Details unterscheiden. Am verbreitetsten sind derzeit die Sprachen Ecore des Eclipse Modeling Frameworks [BSM+03] und die durch die OMG standardisierte Meta Object Facility (MOF) [OMG06a]. Es gibt jedoch auch proprietäre Sprachen wie GOPRR [KT08] und die Domain Model Definition Language [CJKW07], die zusammen mit dazugehörigen Werkzeugen verwendet werden können. Die universitäre Metasprache Kermeta [MFJ05] enthält auch Elemente von Programmiersprachen und kann so die operationale Semantik der Metamodelle spezifizieren. Grammatikbasierte Ansätze verwenden meistens eine Form der kontextfreien Grammatik, die die Modelle auf baumartige Konstrukte beschränken. Die abstrakte Syntax entspricht dabei hierarchischen Datentypen, die kanonisch aus der Grammatik abgeleitet werden. Einige Ansätze wie zum Beispiel [WW93] verwenden auch objektorientierte Vererbung innerhalb der abstrakten Syntax.

Eine Sprache kann mehr als eine konkrete Syntax besitzen und so sowohl eine textuelle als auch eine grafische Eingabe durch den Entwickler ermöglichen. Werkzeuge übersetzen die Eingabe in Instanzen der abstrakten Syntax, wobei Parser die übliche Bezeichnung bei textuellen Sprachen und Editor bei grafischen Ansätzen ist. Bei grafischen Ansätzen gibt es oftmals keine explizite Definition, sondern die konkrete Syntax ist in den Editor integriert. Einige modellbasierte Ansätze wie zum Beispiel [GMF] erlauben jedoch, die konkrete Syntax mittels so genannter Templates für die Elemente der abstrakten Syntax explizit zu modellieren. Für textuelle Ansätze haben sich Grammatiken als verwendeter Formalismus etabliert.

Diese Arbeit fokussiert sich auf die Entwicklung textueller konkreter Syntaxen. Die Gründe hierfür sind detailliert in [GKR+07] dargelegt. Für den Sprachentwickler liegen die Vorteile in der einfacher zu realisierenden Werkzeugunterstützung und in den Kompositionsmechanismen. Für die Entwickler, die die DSL einsetzen, bietet eine textuelle Version die Vorteile, dass sich die Modelle schneller und kompakter formulieren lassen sowie dass der Fokus automatisch auf dem fachlichen Inhalt der Modelle liegt. Die Formatierung der Modelle ist eher nebensächlich oder kann gut automatisiert werden. Des Weiteren können konventionelle Versionsverwaltungen genutzt werden und es ergibt sich kein konzeptueller Unterschied zwischen grafischen Modellen und dem ergänzenden textuellen GPL-Code.

Aufbauend auf der abstrakten Syntax kann ein Codegenerator oder ein Interpreter der DSL implementiert werden. Bei der Realisierung können verschiedene Techniken wie zum Beispiel die kaskadierte Anwendung der Generatoren im GenVoca-Prinzip [BLS98] oder Templates [GC03] verwendet werden. Die Codeerzeugung über eine Template-Sprache hat den Vorteil, dass die konkrete Syntax der Zielsprache zur Erstellung der Codegenerierung verwendet werden kann, was eine kompakte Formulierung und damit eine rasche Anpassung der Codegenerierung ermöglicht.

Die gewählte Zielsprache hat einen nicht unwesentlichen Einfluss auf die Realisierung der Codegenerierung. Mit XMF [CSW08] beispielsweise wurde eine Programmiersprache entwickelt, die spezielle Elemente zur Vereinfachung der Codegenerierung enthält: Aspekte, dynamisches Nachladen von Modulen und die Bereitstellung eines Metaprotokolls erlauben die technische Realisierung von mehrdimensionaler Modularität bei der Codegenerierung

aus Modellen. C# als eine verbreitete GPL bietet mit ihren partial classes die unabhängige Erzeugung zusammengehöriger Elemente im Zielsystem.

Die Nachteile einer eigenständigen DSL-Definition gegenüber eingebetteten DSLs liegen im erhöhten Aufwand der Realisierung, da eine DSL nicht die Infrastruktur zum Beispiel bezüglich eines Typsystems übernehmen kann. Wie in [Tra08] beschrieben ist dies nicht nur ein Problem des erhöhten Aufwands, sondern auch ein Problem der Qualität der entstehenden Elemente. Diese Nachteile werden im Folgenden versucht dadurch zu kompensieren, dass die Sprache modular entwickelt werden kann und somit auf Sprachdefinitionsebene eine Wiederverwendung stattfindet. Diese Wiederverwendung von bereits erstellten Elementen ist beim Entwurf einer DSL aus wenigstens zwei Gründen sinnvoll: Erstens verkürzt sich hierdurch die Entwicklungszeit, da wiederkehrende Sprachelemente nicht stets neu entwickelt werden müssen. Dabei kann die Sprachdefinition genauso wie darauf aufbauende Algorithmen wie eine Typanalyse wieder verwendet werden. Zweitens kann der Entwickler sicher sein, dass die wieder verwendeten Elemente bereits qualitätsgesichert sind, so dass sie ohne eine erneute Prüfung genutzt werden können.

Bei der Entwicklung von Modellierungssprachen ist auffällig, dass häufig Sprachfragmente wie Ausdrücke, Statements und Methoden zugrunde liegender Programmiersprachen benötigt werden, um die modellspezifischen Elemente zu ergänzen. Eingebettete DSLs können diese direkt aus der Basissprache übernehmen, was jedoch eine Übernahme der Hostsprache bedeutet. Für Modellierungssprachen ist hier jedoch teilweise die Verwendung von Sprachen wie die OCL [OMG06b] wünschenswert, die dann ihrerseits auf die GPL abgebildet werden müssen.

Teil II

MontiCore

Kapitel 3

Grammatikbasierte Erstellung von domänenspezifischen Sprachen

Dieses Kapitel beschreibt die einzelnen Elemente des MontiCore-Grammatikformats anhand eines fortlaufend ausgebauten Beispiels. Dabei wird zunächst in Abschnitt 3.1 die Verwendung einer zentralen Sprachdefinition gegenüber einer getrennten Spezifikation von abstrakter und konkreter Syntax motiviert. Zusätzlich werden die grundlegenden Vorteile und Probleme bei der Verwendung einer objektorientierten Datenstruktur zur Beschreibung der abstrakten Syntax einer Sprache dargestellt.

In Abschnitt 3.2 wird die Definition der lexikalischen Syntax einer DSL mit dem MontiCore-Grammatikformat erklärt. Darauf aufbauend werden weitere Elemente zur Spezifikation der kontextfreien Syntax in Abschnitt 3.3 erläutert. Insbesondere die Nichtterminalvererbung und die Verwendung von Schnittstellen und abstrakten Produktionen stellen Erweiterungen gegenüber grammatikbasierten Formaten dar. Ergänzt wird dieses durch die Spezifikation von Assoziationen in Abschnitt 3.4. Insgesamt kann eine abstrakte Syntax abgeleitet werden, die üblichen Metamodellierungsansätzen entspricht.

In Abschnitt 3.5 werden daraufhin weitere Elemente des Grammatikformats erklärt, die die Erstellung einer Sprachdefinition vereinfachen und für spezifische Problemstellungen benötigt werden. Der Abschnitt 3.6 fasst dieses Kapitel kurz zusammen.

Dieses Kapitel basiert teilweise auf dem Konferenzpapier [KRV07b] und dem technischen Bericht [GKR+06]. Die jeweiligen Ausführungen wurden überarbeitet und an den aktuellen Stand der MontiCore-Entwicklung angepasst.

3.1 Grundlagen

Ein zentraler Aspekt einer Sprachdefinition mit dem MontiCore-Framework ist die grammatikbasierte Entwicklung von konkreter und abstrakter Syntax in einem konsistenten Format. Dadurch unterscheidet sich das Grammatikformat von herkömmlichen Parsergeneratoren und vielen modellgetriebenen Ansätzen, die zumeist zwei getrennte Beschreibungsformen verwenden. Die getrennte Spezifikation von konkreter und abstrakter Syntax erlaubt generell beide Artefakte unabhängig voneinander wieder zu verwenden. Dadurch kann beispielsweise die abstrakte Syntax mit verschiedenen konkreten Syntaxen genutzt werden, um somit eine gemeinsame Codegenerierung für verschiedene Repräsentationen zu verwenden. Dieses Vorgehen ist aus zwei Gründen für DSLs als nachrangig anzusehen:

- Eine DSL ist eine domänenspezifische Notation, deren einzige Ausprägung sich auch in der beschriebenen Domäne entwickelt hat.

- Durch die Etablierung von Modelltransformationssprachen stehen deklarative, wartbare und flexible Möglichkeiten zur Überführung einer abstrakten Syntax in eine andere zur Verfügung, so dass auf diesem Wege Codegenerierungen und Analysen gemeinsam genutzt werden können.

Dagegen hat eine einzelne Sprachdefinition für abstrakte und konkrete Syntax die folgenden Vorteile:

- Die abstrakte und die konkrete Syntax sind automatisch konsistent.
- Semantisch ähnliche Elemente einer Sprache sollten sich auch für den Nutzer ähnlich darstellen, damit von der Eingabe leichter auf die Bedeutung eines Programms zu schließen ist (vgl. auch [Hoa73, Wir74]). Eine gemeinsame Spezifikation erzwingt dieses per Konstruktion.
- Die Sprachdefinition kann effektiv als eine zentrale Dokumentation innerhalb einer agilen evolutionären Entwicklung dienen, so dass externe Dokumentation zweitrangig ist.
- Es existiert automatisch eine konkrete Syntax, was die Verwendung der Sprache durch Nutzer und das Testen darauf basierender Werkzeuge durch Entwickler erleichtert, da diese nicht, wie bei Metamodellierung oft üblich, eine generische Repräsentation verwenden müssen.

Die in diesem Kapitel beschriebene Form der Ableitung der abstrakten Syntax aus der Grammatik assoziiert eine Produktion mit einer Klasse in der abstrakten Syntax. Diese Gleichsetzung ist durchaus üblich, wie zum Beispiel objektorientierte Attributgrammatiken [Hed89] zeigen. Es gibt jedoch prinzipielle strukturelle Unterschiede zwischen Grammatiken und Datentypen durch die Alternativen in einer Grammatik, so dass für eine Abbildung auf ein normales objektorientiertes Typsystem gewisse Kompromisse eingegangen werden müssen. Produktionen der Form `A=B|C` können durch Unterklassenbildung von `B` und `C` von einer Klasse `A` behandelt werden. Produktionen der Form `A = (B|C)* D` lassen sich jedoch nicht direkt repräsentieren. Durch die eingesetzte Abbildung wird eine relativ flache abstrakte Syntax erreicht, wobei `B`, `C` und `D` direkt als Attribute der Klasse A verwendet werden. Die beliebige Schachtelung durch Blöcke und Alternativen ist aber nicht wieder aus der abstrakten Syntax rekonstruierbar. Insbesondere wurde im vorliegenden Fall auch von der Reihenfolge der von `B` und `C` erkannten Elemente untereinander abstrahiert.

Die geschilderten Probleme könnten durch die Verwendung einer reinen BNF-Form vermieden werden. Die Spezifikation der konkreten und abstrakten Syntax mit EBNF-Elementen wie Blockbildung, Alternativen und Iterationen erlaubt jedoch im Gegensatz zur BNF-Notation die kompakte Erstellung von Grammatiken mit einer geringeren Anzahl an Produktionen. Allerdings muss eine zu starke Vereinfachung und der Verlust von notwendigen Informationen durch den Entwickler vermieden werden.

Die Unterschiede und Schwierigkeiten bei der Abbildung einer Sprachstruktur auf eine Objektstruktur wird intensiv in [LM07] diskutiert. Dabei wird die notwendige Erweiterung eines objektorientierten Typsystems aufgezeigt, um eine Unifikation mit grammatikbasierten Datentypen zu erreichen [MBB06]. Ein solcher Ansatz wurde hier nicht weiter verfolgt, weil sich so die abstrakte Syntax sehr an der konkreten Struktur der Eingabe orientiert und so der erreichte Abstraktionsgrad relativ gering ist. Die von MontiCore verwendeten flachen Strukturen sind akzeptierte Repräsentationen der abstrakten Syntax einer Sprache, wie die Metamodellierung und algebraische Datentypen zeigen. Des Weiteren kann die

Implementierung von Algorithmen, die einen Großteil der erforderlichen Arbeit bei einer DSL-Entwicklung ausmacht, mit den in objektorientierten Programmiersprache üblichen, stark getypten Datenstrukturen erfolgen. Somit unterscheiden sich die DSL-Modelle bei der Verarbeitung nicht von den sonst verwendeten Datentypen.

Das MontiCore-Grammatikformat erlaubt neben der Ableitung einer Baumstruktur, wie sie sich durch die Verwendung der Nichtterminale innerhalb der Grammatik ergibt, auch die explizite Definition nicht-kompositionaler Abhängigkeiten zwischen Nichtterminalen. Zusätzlich ist es möglich, Vererbungsbeziehungen in der abstrakten Syntax auszudrücken, um so insgesamt dieselben Konzepte wie in der Metamodellierung verwenden zu können. Daher wird eine weitgehende Konformität zu anderen modellgetriebenen Ansätzen wie [BSM+03, JB06, AKRS06, OMG06b] erreicht.

3.2 Lexikalische Syntax

Das MontiCore-Grammatikformat ist eine angereicherte kontextfreie Grammatik, die den Eingabesprachen von Parsergeneratoren, insbesondere der Eingabe von Antlr [PQ95, Par07] ähnelt, weil die Parsergenerierung dieses Werkzeug nutzt. Dabei wird zwischen der lexikalischen Syntax, die festlegt wie Wörter aus einem Zeichenstrom durch einen Lexer identifiziert werden, und der kontextfreien Syntax unterschieden, die darauf aufbauend angibt wie in einem Parser Sätze gebildet werden.

Die lexikalische Syntax wird in einer MontiCore-Grammatik durch lexikalische Produktionen festgelegt, die durch das Schlüsselwort `token` eingeleitet werden und reguläre Ausdrücke auf der rechten Seite zur Definition einer Tokenklasse verwenden. Zur Vereinfachung der Entwicklung von DSLs sind die Produkionen `IDENT` und `STRING` vordefiniert, um Namen und Zeichenketten standardisiert zu verarbeiten. Die von diesen Produktionen erstellten Werte werden üblicherweise in einem String gespeichert. Die Verwendung komplexerer Datentypen ist durch Angabe einer Funktion in der Programmiersprache Java möglich, die die Eingabezeichenkette auf einen beliebigen Datentyp abbildet. Es existieren Standardfunktionen für primitive Datentypen wie Fließkommazahlen (`float`) und Ganzzahlen (`int`).

Die Abbildung 3.1 zeigt illustrierende Beispiele für einige lexikalische Produktionen: In Zeile 2 wird die lexikalische Produktion `IDENT` definiert, die auf String abgebildet wird. `NUMBER` aus Zeile 5 wird auf Ganzzahlen abgebildet, wobei die Standard-Abbildung genutzt wird. In Zeile 8 wird `CARDINALITY` definiert, das auf ein `int` in der abstrakten Syntax abgebildet wird. Dazu wird der Java-Code aus den Zeilen 10 bis 13 genutzt, der Zahlen normal verarbeitet und die unbeschränkte Kardinalität durch -1 repräsentiert. Der Code ist dabei ein Methodenrumpf einer Methode und muss dabei ein Objekt vom Typ `Token` auf den entsprechenden Datentyp abbilden. Neben der dargestellten Methode `getText()` stehen zum Beispiel mit `getLine()` und `getColumn()` noch weitere Methoden zur Verfügung. Das Beispiel zeigt zusätzlich noch, wie innerhalb des Grammatikformats Kommentare verwendet werden können: `//` leitet einen Kommentar bis zum Zeilenende ein. Zusätzlich können auch Blockkommentare zwischen `/*` und `*/` verwendet werden.

3.3 Kontextfreie Syntax

Die kontextfreie Syntax einer Sprache wird in einer MontiCore-Grammatik durch so genannte Parserproduktionen beschrieben. Dabei gibt es drei verschiedene Arten von Parserproduktionen:

MontiCore-Grammatik

```
1 // Simple name
2 token IDENT = ('a'..'z'|'A'..'Z')+ ;
3
4 // Numbers (using default transformation)
5 token NUMBER = ('0'..'9')+ : int;
6
7 // Cardinality (STAR = -1)
8 token CARDINALITY = ('0'..'9')+ | '*' :
9   x -> int :
10  {
11    if (x.getText().equals("*")) return -1;
12    else return Integer.parseInt(x.getText());
13  };
```

Abbildung 3.1: Definition von lexikalischen Produktionen

- Klassenproduktionen ohne Schlüsselwort (vgl. Abschnitt 3.3.1),
- Schnittstellenproduktionen, gekennzeichnet durch das Schlüsselwort `interface`, und
- abstrakte Produktionen, eingeleitet durch `abstract`.

Die Klassen und Attribute der abstrakten Syntax werden aus den Rümpfen der Parserproduktionen abgeleitet. Dabei können Attribute und Kompositionen entstehen, deren Werte direkt von den Nutzereingaben abhängen (vgl. Abschnitt 3.3.2), oder boolsche Attributen, Konstanten und Enumerationen, die nur eine begrenzte Anzahl an Werten zulassen (vgl. Abschnitt 3.3.3). Der Ableitungsprozess garantiert die Typsicherheit beim Parsen von Modellen und ermittelt automatisch die Kardinalitäten der Kompositionen.

Die einzelnen Parserproduktionen können untereinander über Vererbung und Implementierungsbeziehungen verbunden sein, wobei sich dieses sowohl auf die Ableitung der konkreten als auch der abstrakten Syntax auswirkt (vgl. Abschnitt 3.3.4).

Die automatische Ableitung der Attribute kann durch so genannte AST-Regeln verändert werden. Dadurch können die automatisch ermittelten Kardinalitäten weiter eingeschränkt, die Typisierung der Attribute verändert oder manuell codierte Klassen eingebunden werden (vgl. Abschnitt 3.3.5).

3.3.1 Klassenproduktionen

Die Abbildung 3.2 enthält ein kurzes Beispiel, das die Kernelemente des Grammatikformats darstellt. Dabei wird eine Grammatik mit Namen `Shop` definiert, die aus den lexikalischen Produktionen und einer Menge von Klassenproduktionen besteht. Sie beschreibt eine fiktive Datenstruktur, die die Grundlage für ein einfaches Verkaufssystem sein könnte. Das Beispiel wird im gesamten Kapitel fortlaufend aus- und umgebaut, um die verschiedenen Merkmale des Grammatikformats zu erklären. Dabei steht weniger die Realitätsnähe des Beispiels, sondern vielmehr die Darstellung der verschiedenen Elemente des MontiCore-Grammatikformats im Vordergrund.

Das Shopsystem kennt dabei zwei verschiedene Arten von Kunden, normale und Premiumkunden, deren Adressen verwaltet werden. Den Premiumkunden wird ein gewisser Rabatt beim Einkauf eingeräumt. Die Kunden können die nicht modellierten Waren sowohl bar als auch mit einer Kreditkarte zahlen. Bei einer Buchung wird der Kundenname als eindeutige Zuordnung gespeichert.

MontiCore-Grammatik

```
1  grammar Shop {
2
3    token NUMBER = ('0'..'9')+ : int;
4
5    token CARDINALITY = ('0'..'9')+ | '*' : x -> int :
6                        { if (x.getText().equals("*")) return -1;
7                          else return Integer.parseInt(x.getText()); };
8
9    ShopSystem =
10     name:IDENT (Client | PremiumClient)* (OrderCash | OrderCreditcard)*;
11
12   Client =
13     "client" name:IDENT Address;
14
15   PremiumClient =
16     "premiumclient" name:IDENT discount:NUMBER Address;
17
18   Address =
19     street:STRING town:STRING;
20
21   OrderCash =
22     "cashorder" clientName:IDENT amount:NUMBER;
23
24   OrderCreditcard =
25     "creditorder" clientName:IDENT billingID:NUMBER;
26
27 }
```

Abbildung 3.2: Definition von Produktionen in MontiCore

Eine Parserproduktion hat dabei einen Namen und einen Produktionsrumpf (rechte Seite), die durch ein Gleichheitszeichen getrennt sind. Der Rumpf enthält dabei Nichtterminale (Verweise auf andere Produktionen) und Terminale (konstante Zeichenketten in Anführungszeichen). Der Rumpf kann dabei weiter durch Alternativen (getrennt durch `|`), Blöcke (in runden Klammern) und Wiederholungen in Form von Kardinalitäten strukturiert werden. Dabei können Blöcke, Nichtterminale und Terminale die Kardinalitäten `?` (optional), `*` (unbeschränkt) oder `+` (mindestens eins) haben. Nicht durch Kardinalitäten gekennzeichnete Elemente treten genau einmal auf.

Die interne Repräsentation oder abstrakte Syntax einer Sprache wird ebenfalls durch die Grammatik festgelegt. Die Abbildung 3.3 zeigt ein Klassendiagramm, das aus der Grammatik in Abbildung 3.2 abgeleitet wurde. Im Folgenden wird die Ableitung informell beschrieben, eine Formalisierung findet sich in Kapitel 5. Dieses Klassendiagramm kann dann auf verschiedene Weisen in Code umgesetzt werden. In Kapitel 6 wird eine kanonische Abbildung auf Java-Klassen dargestellt, die die Standard-Realisierung innerhalb des MontiCore-Frameworks darstellt. Eine alternative Abbildung auf EMF wird ebenfalls skizziert. Dieses zweischrittige Verfahren mag zunächst umständlich erscheinen, ermöglicht jedoch eine plattformunabhängige Definition des Grammatikformats und somit eine kompakte Semantikdefinition ohne technische Details.

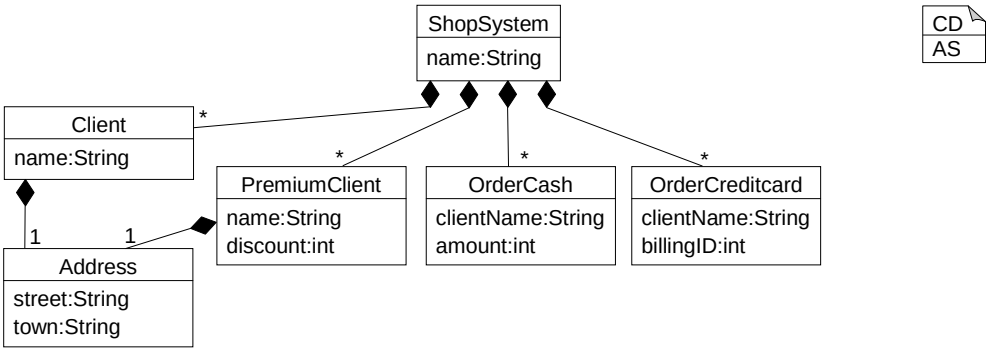


Abbildung 3.3: Aus der Grammatik in Abbildung 3.2 abgeleitete abstrakte Syntax

3.3.2 Ableitung der Attribute und Kompositionen

Jede Klassenproduktion der Grammatik wird zu einer Klasse in der abstrakten Syntax abgeleitet. Der Rumpf einer Produktion legt die Attribute und Kompositionen innerhalb der abstrakten Syntax fest. Dabei können die Elemente der rechten Seite explizit benannt werden, indem ein Name getrennt durch einen Doppelpunkt vor dem jeweiligen Element eingefügt wird. Falls sie unbenannt sind, wird der kleingeschriebene Name des Nichtterminals verwendet. Zusätzlich können Elemente explizit durch Anfügen eines Ausrufezeichens oder implizit durch die Nichtbenennung eines Terminals von der Abbildung in die abstrakte Syntax ausgeschlossen werden. Verweise auf jedes Nichtterminal, das eine Klassenproduktion repräsentiert, werden auf eine Komposition mit der jeweiligen Klasse abgebildet. Verweise auf lexikalische Produktionen und Terminale (sofern benannt) werden als Attribute in der abstrakten Syntax realisiert.

Die Struktur der Produktion, wie sie durch die Klammerung und die Iteration durch kleensche Sterne und optionale Elemente gegeben ist, wird analysiert und zur Ableitung der Kardinalitäten von Kompositionen verwendet. Bei Attributen werden ggf. Listen dieses Typs erzeugt, wenn eine mehrfache Zuweisung möglich ist.

Die Abbildung 3.3 zeigt, wie aus den Produktionen der Abbildung 3.2 jeweils gleichnamige Klassen entstehen. Die Klasse `ShopSystem` verfügt über ein Attribut `name` vom Typ `String`, weil es ein Nichtterminal `name:IDENT` beinhaltet. Der Typ `String` ist, wie bereits beschrieben, der Standard, weil in Abbildung 3.1 kein spezieller Typ definiert wurde. Die Klasse `PremiumClient` hingegen verfügt über ein Attribut `discount` vom Typ `int`, weil das Nichtterminal `discount:NUMBER` auf eine lexikalische Produktion verweist, die diesen verwendet. Die Klasse `ShopSystem` verfügt über vier verschiedene unbenannte Kompositionen, die aus den Nichtterminalen im Produktionsrumpf abgeleitet wurden. Die Kardinalität ist hierbei jeweils unbeschränkt (`*`), weil aufgrund der Produktionsstruktur von `ShopSystem` eine beliebige Anzahl der Nichtterminale auftreten kann. Anders hingegen ergibt bei `PremiumClient` dieselbe Analyse, dass die Kardinalität von `Address` stets eins ist.

3.3.3 Boolsche Attribute, Konstanten und Enumerationsproduktionen

Die Abbildung 3.4 zeigt eine alternative Definition für die Produktion `Client`, die die beiden Produktionen `Client` und `PremiumClient` aus Abbildung 3.2 ersetzt. Dabei wird die

konkrete Syntax und die Sprache, d. h. die Menge der validen Sätze, nicht verändert, sondern ausschließlich die abstrakte Syntax angepasst. Es gibt hier nur eine einzelne Klasse `Client`, die allerdings über ein boolsches Attribut `premium` verfügt, um so normale Kunden und Premiumkunden zu unterscheiden. Das boolsche Attribut ensteht durch die Angabe des Terminals `"premiumclient"` in eckigen Klammern. Wird dieses Terminal beim Parsen erkannt, wird das boolsche Attribut auf *wahr* gesetzt. Wird es, weil die zweite Alternative gewählt wurde, nicht auf wahr gesetzt, behält es seinen Standardwert *falsch*. Wie die anderen Elemente einer Produktion können diese benannt werden, was zum Namen `premium` führt. Für unbenannte Konstanten, die wie hier nur die zwei Werte *wahr* und *falsch* annehmen können, wird aus dem Konstantenwert ein Name abgeleitet.

Es ist auch möglich, unter dem einem Attributnamen mehrere Konstantenwerte zu verwenden. Dadurch entsteht eine Enumeration und das Attribut nimmt diesen Enumerationstyp an. Dabei sind die folgenden beiden Varianten sowohl für die abstrakte als auch die

MontiCore-Grammatik

```
1  grammar Shop2 {
2
3    ShopSystem =
4      name:IDENT Client* (OrderCash | OrderCreditcard)*;
5
6    Client =
7        premium:["premiumclient"] name:IDENT discount:NUMBER Address
8        |
9        "client" name:IDENT Address;
10
11   Address =
12     street:STRING town:STRING;
13
14   OrderCash =
15     "cashorder" clientName:IDENT amount:NUMBER;
16
17   OrderCreditcard =
18     "creditorder" clientName:IDENT billingID:NUMBER;
19
20 }
```

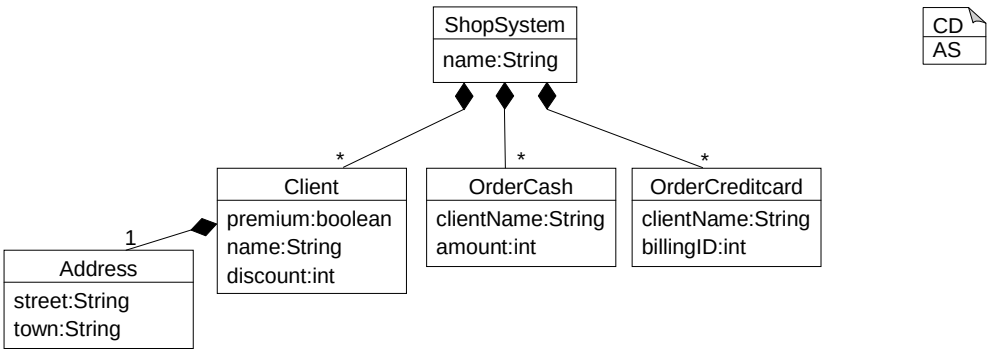


Abbildung 3.4: Definition des ShopSystems mit Konstanten und die daraus abgeleitete abstrakte Syntax

konkrete Syntax gleichbedeutend: `A = x:["a"] | x:["b"];` und `A = x:["a"|"b"];`. Für einige DSLs ist es nötig, in der konkreten Syntax verschiedene Konstantensymbole anzubieten, die jedoch dieselbe Bedeutung in der abstrakten Syntax haben. Daher kann die Ableitung des Konstantennamens aus der Zeichenkette durch den Entwickler überschrieben werden. Die Produktion `C = visibility:[PUBLIC:"+"] | visibility:[PUBLIC:"public"]` erzeugt ein boolsches Attribut, da beide Zeichenketten `"+"` und `"public"` für den Konstantenwert `PUBLIC` stehen. Für die zweite Alternative kann auch `visibility:["public"]` als Kurzform verwendet werden, da der Konstantenname der Standardableitung entspricht, die die Zeichenkette in Großbuchstaben umwandelt.

Zusätzlich zu den beschriebenen implizit definierten Konstanten kann auch eine explizite Variante genutzt werden, wobei in einer Enumerations-Produktion die möglichen Werte aufgeführt werden. Enumerations-Produktionen werden durch das Schlüsselwort `enum` eingeleitet und enthalten die Aufzählung der validen Werte getrennt durch `|`. Diese Enumeration kann dann in verschiedenen Produktionen verwendet werden und erlaubt so eine gemeinsame Nutzung von Konstanten in verschiedenen Klassen. In der abstrakten Syntax werden diese Produktionen auf Enumerationen abgebildet. Die Abbildung 3.5 zeigt ein Beispiel für eine Enumerations-Produktion und die dabei entstehende abstrakte Syntax.

MontiCore-Grammatik

```
1  Client =
2      Kind name:IDENT discount:NUMBER;
3
4  enum Kind =
5        "broke" | "basic" | "premium";
```

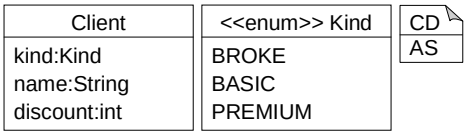


Abbildung 3.5: Verwendung von Enumerations-Produktionen

3.3.4 Schnittstellen, abstrakte Produktionen und Vererbung

Beide Versionen der Shopsystem-Sprache, wie sie bisher dargestellt wurden, haben Defizite: In Abbildung 3.2 haben die Produktionen `Client` und `PremiumClient` bzw. `OrderCash` and `OrderCreditcard` eine gemeinsame Teilstruktur, was sich aber nicht in der abstrakten Syntax widerspiegelt. In Abbildung 3.4, wurde `Client` und `PremiumClient` durch eine einzelne Produktion `Client` ersetzt. Die zusätzliche Invariante, dass nur ein Rabatt (Discount) definiert werden darf, wenn das boolsche Flag `premium` *wahr* ist, ist aus der abstrakten Syntax nicht ersichtlich.

Dieses Defizit ist der Grund für die Verwendung von objektorientierten Elementen in einer MontiCore-Grammatik wie sie in der Metamodellierung üblich sind. In Abbildung 3.6 werden diese Elemente verwendet, wobei neben der MontiCore-Grammatik auch die resultierende abstrakte Syntax als Klassendiagramm dargestellt ist. Der EBNF-Abschnitt zeigt die konkrete Syntax, um zu veranschaulichen, welche Eingabe die Parsergenerator-Komponente verwendet.

MontiCore-Grammatik

```
1  grammar Shop3 {
2
3    ShopSystem =
4      name:IDENT Client* Order*;
5
6    Client =
7      "client" name:IDENT Address;
8
9    PremiumClient extends Client =
10     "premiumclient" name:IDENT
11     discount:NUMBER Address;
12
13   Address =
14     street:STRING town:STRING;
15
16   interface Order;
17
18   ast Order =
19     clientName:IDENT;
20
21   OrderCash implements Order =
22     "cashorder"
23     clientName:IDENT amount:NUMBER;
24
25   OrderCreditcard implements Order =
26     "creditorder"
27     clientName:IDENT billingID:NUMBER;
28 }
```

EBNF

```
1
2
3  ShopSystem ::=
4    IDENT Client* Order*
5
6  Client ::=
7    PremiumClient | "client" IDENT Address
8
9  PremiumClient ::=
10   "premiumclient" IDENT
11   NUMBER Address
12
13 Address ::=
14   STRING STRING
15
16 Order ::=
17   OrderCash | OrderCredit
18
19
20
21 OrderCash ::=
22   "cashorder"
23   IDENT NUMBER
24
25 OrderCreditcard ::=
26   "creditorder"
27   IDENT NUMBER
28
```

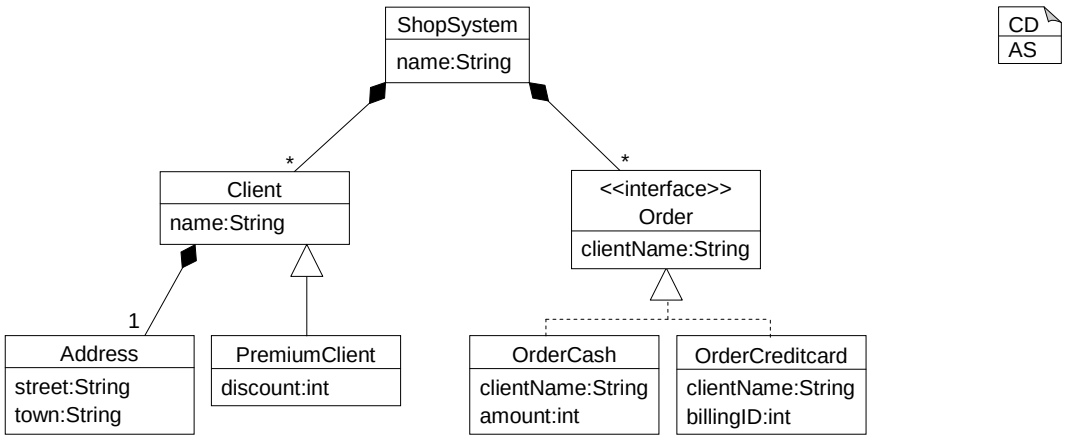


Abbildung 3.6: Vererbung und die Verwendung von Schnittstellen

Innerhalb einer MontiCore-Grammatik gibt es die folgenden drei Möglichkeiten, Vererbungs- und Implementierungsbeziehungen zu den Produktionen hinzufügen:

- Klassenproduktionen einer Grammatik können voneinander erben. Dabei wird das Schlüsselwort `extends` in der Unterproduktion verwendet, um auf die Oberproduktion zu verweisen. Im Beispiel in Abbildung 3.6 ist die Klasse `PremiumClient` eine Unterklasse von `Client`. In der konkreten Syntax wird eine neue Alternative zur Produktion `Client` hinzugefügt, die direkt auf `PremiumClient` ableitet.

- Die Einführung einer Schnittstellenproduktion `Order` beschreibt, dass eine Schnittstelle zur abstrakten Syntax hinzugefügt wird, wie aus Abbildung 3.6 ersichtlich ist. Dazu wird das Schlüsselwort `interface` gefolgt vom Namen der Produktion benutzt. Nichtterminale bzw. die daraus abgeleiteten Klassen können diese Schnittstelle implementieren, indem sie das Schlüsselwort `implements` verwenden. Bei diesem Vorgehen wird in der konkreten Syntax die Produktion `Order = OrderCash | OrderCreditCash` hinzugefügt. Dieses Vorgehen überträgt das Verständnis aus der abstrakten Syntax auf die konkrete Syntax, dass überall dort, wo eine Schnittstelle verwendet wird, Objekte aller Klassen auftreten können, die die Schnittstelle implementieren. Somit kann jedes Auftreten von `Order` direkt zu `OrderCash` oder `OrderCreditCash` abgeleitet werden.

- Abstrakte Produktionen beginnen mit dem Schlüsselwort `abstract` und werden zu einer abstrakten Klasse abgeleitet. Das Schlüsselwort `extends` kann eingesetzt werden, um eine Vererbungsbeziehung zwischen den beteiligten Klassen zu erreichen. Abstrakte Klassen bieten prinzipiell dieselbe Funktionalität wie Schnittstellen, erlauben jedoch zusätzlich die Definition von Methoden innerhalb der AST-Klassen.

Das MontiCore-Grammatikformat bietet in Bezug auf Vererbung und Schnittstellen weitergehende Unterstützung an, die eine Erstellung teilweise vereinfacht. Als eine Alternative zu den Schlüsselwörtern `extends` und `implements` können auch `astextends` und `astimplements` verwendet werden. Die Verfeinerung beschränkt sich dann auf die abstrakte Syntax und es werden keine zusätzlichen Alternativen in der konkreten Syntax erzeugt. Die Schlüsselwörter `extends` und `implements` bzw. `astextends` und `astimplements` können auch innerhalb der abstrakten Produktionen und Schnittstellenproduktionen genutzt werden, um hier Vererbungsbeziehungen zu spezifizieren.

Diese Notation ähnelt den meisten objektorientierten Programmiersprachen, bei denen die Implementierung einer Schnittstelle nicht in der Schnittstelle sondern der Klasse spezifiziert wird. Die Abbildung auf die abstrakte und konkrete Syntax entspricht ebenfalls der objekt-orientierten Form der Vererbung, bei der jedes Auftreten einer Superklasse oder Schnittstelle durch eine Unterklasse oder implementierende Klasse ersetzt werden kann. Insbesondere wird dadurch die Definition einer Schnittstelle nicht verändert, wenn eine neue implementierende Klasse hinzugefügt wird. Diese Form der Notation ist für Entwickler, die objektorientierte Programmiersprachen nutzen, intuitiv einzusetzen, widerspricht jedoch üblichen grammatikorientierten Notationen. Der objektorientierte Stil wird vor allem wegen der Erweiterungsfähigkeit von Grammatiken verwendet, wie sie in Kapitel 4 dargestellt wird. Es wird dadurch teilweise schwieriger, die Struktur der Grammatik nachzuvollziehen, weil die Ableitung für eine Schnittstellenproduktion über die gesamte Grammatik (und wie später dargestellt, auch Obergrammatiken) verteilt sein kann. Dieser Nachteil lässt sich jedoch durch die im Kapitel 6 dargestellte Werkzeugunterstützung ausgleichen, die dem Entwickler die notwendigen Informationen bei Bedarf anzeigt.

Alternativ ist es auch möglich, die für Grammatiken übliche Notation zu verwenden und so direkt in Schnittstellenproduktionen oder abstrakten Produktionen die jeweiligen Unterproduktionen getrennt durch `|` anzugeben. Wenn der Produktionsrumpf leer ist, wie in Beispiel (Abbildung 3.6, Zeile 16) gezeigt, werden alle implementierenden Produktionen getrennt durch `|` als Produktionsrumpf verwendet.

3.3.5 Zusätzliche Attribute der abstrakten Syntax

Schnittstellen und abstrakte Klassen enthalten standardmäßig keine Attribute. Eine automatische Strategie zur Ableitung der gemeinsamen Attribute für die bekannten Unterklassen wurde absichtlich nicht verwendet, weil Schnittstellen eine Möglichkeit für zukünftige Erweiterungen sind, welche vielleicht nur eine Teilmenge der verfügbaren Attribute bereitstellen. Es können jedoch grundsätzlich innerhalb des Grammatikformats für jede Klasse oder Schnittstelle, die aus einer Produktion abgeleitet wird, also auch für Schnittstellen- und abstrakte Produktionen, zusätzliche Attribute hinzugefügt werden. Dazu können so genannte AST-Regeln verwendet werden, die mit dem Schlüsselwort `ast` eingeleitet werden (vergleiche zum Beispiel Abbildung 3.6 (links), Zeile 18). Die Regeln verwenden dieselbe Syntax wie normale Produktionen, aber fügen nur Attribute zur abstrakten Syntax hinzu und verändern nicht die konkrete Syntax. Die Nichtterminale können dabei auf andere Produktionen oder direkt auf Java-Datentypen (eingeleitet durch `/`) verweisen. Ein Stern hinter einem Nichtterminal sorgt dafür, dass eine Liste bzw. eine Komposition mit mehrfacher Wertbelegung entsteht. Die Attribute von Schnittstellen werden schließlich als get- und set-Methoden in der Implementierung realisiert, daher sind Attribute in Schnittstellen erlaubt.

Zusätzlich zu den Möglichkeiten, wie sie bereits für normale Produktionen möglich sind, können die Schlüsselwörter `min` und `max` verwendet werden, um die Kardinalität von Attributen und Kompositionen festzulegen. Ebenfalls möglich ist die Definition von Methoden, die zu Klassen hinzugefügt werden sollen. Diese werden durch das Schlüsselwort `method` eingeleitet. Für Schnittstellen werden dabei die Methodenrümpfe ignoriert. Als eine redundante Möglichkeit zur Definition in den Produktionen können Schnittstellen und Klassen als Oberklassen auch hier zur abstrakten Syntax hinzugefügt werden.

Eine Alternative zur Verwendung von AST-Produktionen ist die spezielle Kennzeichnung eines Nichtterminals als handcodiert durch einen Schrägstrich vor der Produktion. Die Abbildung 3.7 zeigt, wie dadurch anstatt der Klasse `Client` eine Klasse `AbstractClient`

MontiCore-Grammatik

```
1 grammar Shop4 {
2
3
4   / Client =
5     "client" name:IDENT Address;
6
7   // ...
8
9 }
```

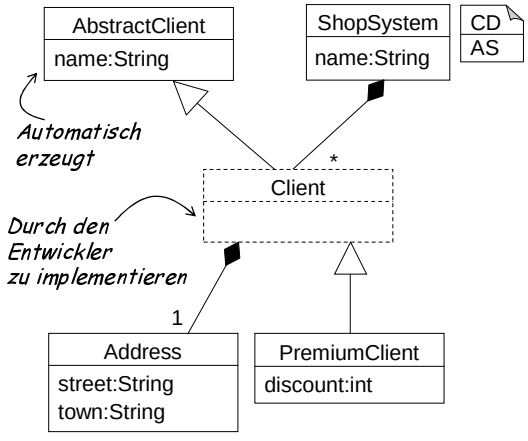


Abbildung 3.7: Verwendung von handcodierten Klassen (Erweiterung der Grammatik aus Abbildung 3.6).

erzeugt wird, die dieselben Attribute und Kompositionen wie die ursprüngliche Klasse `Client` enthält. Die Kompositionen und Attribute aller anderen Klassen verweisen jedoch weiterhin auf die Klasse Client, die nun vom Entwickler selbst implementiert werden muss. Dabei kann die Klasse `AbstractClient` als Oberklasse verwendet werden, um die gesamte von MontiCore generierte Infrastruktur zu erben und nicht separat implementieren zu müssen. Dann kann der Entwickler unterstützt durch seine IDE zusätzliche Funktionalitäten in der Klasse implementieren.

3.4 Assoziationen

Die beiden Attribute `name` der Klasse `Client` und `clientName` der Klasse `Order` in Abbildung 3.6 hängen offensichtlich zusammen, was sich jedoch nicht innerhalb der kontextfreien Syntax beschreiben lässt. In diesem Beispiel wird eine Invariante benötigt, so dass eine `Order` nur Kundennamen nutzen darf, die auch existieren, in dem Sinne, dass es ein Objekt vom Typ `Client` mit diesem Namen gibt. Zusätzlich soll eine effiziente Navigation von der Verwendung eines Namens zu seiner Definition möglich sein, so dass dort zusätzliche Informationen wie zum Beispiel der mögliche Rabatt ausgelesen werden können. Diese Navigation vereinfacht die Implementierung von Codegenerierungen und Analysen. Bei der Metamodellierung werden solche Verbindungen üblicherweise dadurch ausgedrückt, dass eine Assoziation erstellt wird, wobei eine `Order` einen `Client` als die ordernde Person referenziert. Diese Assoziationen erweitern die Baumstruktur zu einer Graphstruktur. Innerhalb von MontiCore wird die kontextfreie Grammatik so erweitert, dass Assoziationen definiert werden können. Dabei wird das Schlüsselwort `association` verwendet, um nichtkompositionale Assoziationen zwischen Klassen in der abstrakten Syntax zu beschreiben.

Die Abbildung 3.8 zeigt ein Beispiel für eine Assoziation. Die Assoziation `ClientOrder` verbindet jede `Order` mit einem einzelnen `Client` (wie durch `1` festgelegt). Umgekehrt ist ein `Client` mit einer unbeschränkter Anzahl an `Order` verbunden (durch `*` festgelegt). Dabei ist das eine Assoziationende mit `orderingClient` benannt. Die Namen der Assoziationsenden können ausgelassen werden, wodurch das Assoziationsende implizit durch die gegenüberliegende Klasse, beginnend mit einem Kleinbuchstaben, benannt ist. Daher wird im Beispiel der Name `order` ausgelassen. Die Kardinalitäten können neben den häufig verwendeten Symbolen `1`, `1..*` und `*` auch aus Bereichen wie `3..4` bestehen. Die Navigation einer Assoziation kann mit `<-` oder `->` auch auf eine Richtung beschränkt werden und der Assoziationsname kann ausgelassen werden, wobei er dann aus den Namen der beiden beteiligten Klassen gebildet wird. In diesem Beispiel könnte also die Bezeichnung `ClientOrder` ausgelassen werden, weil die automatische Ableitung dasselbe Ergebnis liefern würde.

Die Hauptaufgabe bei der Integration von Assoziationen in ein einheitliches Format zur Definition von abstrakter und konkreter Syntax ist jedoch nicht die Spezifikation von Assoziationen, sondern die automatische Erzeugung der Links zwischen den assoziierten Objekten nach der Syntaxanalyse. Grammatikbasierte Systeme verwenden als Eingabe lineare Zeichenketten und repräsentieren diese in einer Baumstruktur, die durch Grammatik induziert wird. Alle zusätzlichen Verbindungen werden durch die Definition und Verwendung von Namen (für Klassen, Methoden, Attribute, Zustände, Objekte etc.) sowie deren Kontext (wie Methodenbindung aufgrund von formalen Parametern) festgelegt. Dabei spielt insbesondere die Symboltabelle als eine abgeleitete Datenstruktur aus dem AST eine Rolle.

Für DSLs sind aufgrund ihrer Kompaktheit oftmals einfache Lösungen wie dateiweit eindeutige Identifikatoren die passende Wahl der Realisierung. Komplexere Sprachen wie die UML-Teilsprachen benötigen hingegen komplexere Namenssysteme. Um eine Unter-

MontiCore-Grammatik

```
1 grammar Shop5 {
2
3   // ...
4
5   association ClientOrder
6      Client 1 <-> * Order.OrderingClient;
7
8 }
```

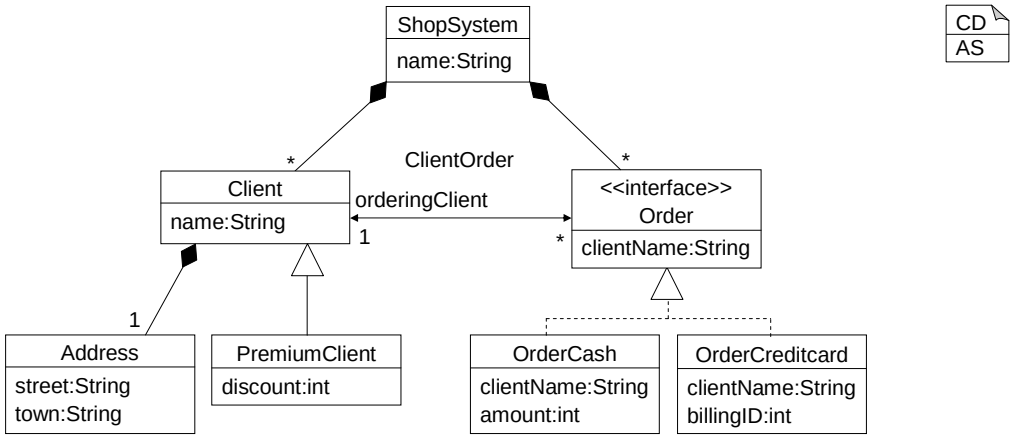


Abbildung 3.8: Definition einer Assoziation (Erweiterung der Grammatik aus Abbildung 3.6)

stützung für komplexere Sprachen in MontiCore zu integrieren, müssen das Scoping und die Namensauflösungsstrategien so formalisiert werden, dass ein Sprachentwickler diese in der Sprachdefinition konfigurieren kann.

Da jede Sprache dieselbe Art von Assoziationen enthalten kann, die Etablierung der Links jedoch individuell ist, wird in MontiCore eine zweigleisige Strategie verwendet: Erstens können Assoziationen auf eine einheitliche Weise innerhalb der Grammatik definiert werden. Daraus werden einheitliche Schnittstellen erzeugt, deren Methoden eine Navigation zwischen den einzelnen Klassen der abstrakten Syntax ähnlich wie Kompositionen erlauben. Zweitens können spezielle Konzepte (vgl. Abschnitt 4.3) verwendet werden, um die Namensauflösung zu definieren. Existierende Lösungen für flache und hierarchische Namensräume werden in Kapitel 6 erklärt. Alternativ kann der Sprachentwickler, falls kein passendes Konzept existiert und eine Neuentwicklung zu aufwändig erscheint, diese manuell implementieren. Hierfür erzeugt MontiCore dokumentierte Schnittstellen, so dass eine spezifische manuelle Implementierung leicht integriert werden kann. Die Links werden dabei erst nach der Syntaxanalyse etabliert und nicht direkt nach der Erzeugung. Auf diese Art und Weise können Vorwärtsreferenzen in einer Sprache realisiert werden, was bedeutet, dass sich Bezeichner auf Elemente beziehen, die erst zu einem späteren Zeitpunkt in der Datei definiert werden.

Assoziationen sind geeignet, um Referenzen innerhalb eines Modells darstellen zu können. Für zusätzlich zu berechnende Elemente, wie zum Beispiel die Typisierung von Ausdrücken, eignen sich eher Attributgrammatiken [Knu68]. Der Grund hierfür ist, dass sich der Typ meistens nur schwer in der abstrakten Syntax identifizieren lässt, sondern eher nur aus ihr abgeleitet werden kann. Bei objektorientierten Sprachen ist die Beziehung noch re-

lativ direkt, so dass anstatt eines Typs auch die definierende Klasse verlinkt werden könnte. Sobald aber Generizität eine Rolle spielt, ist der Typ nicht mehr durch eine einfache Relation darzustellen. Für solche Fälle steht in MontiCore ein Attributgrammatikformalismus als Konzept zur Verfügung.

3.5 Weiterführende Elemente

Das Grammatikformat verfügt über weitere Elemente, die die Beschreibung von DSLs vereinfachen. Diese werden in diesem Abschnitt übersichtsartig dargestellt.

3.5.1 Optionen

Die Codegenerierung aus einer MontiCore-Grammatik kann durch die Angabe von Optionen beeinflusst werden. Diese werden in einem Block zusammengefasst, der mit dem Schlüsselwort `options` eingeleitet wird. Durch die Angabe von `lexer` und `parser` kann der Lookahead festgelegt werden. Der Lookahead bezeichnet die Anzahl der Zeichen beim Lexer bzw. Wörter beim Parser, nach denen eine eindeutige Produktionsauswahl erfolgt. Fehlen diese Angaben, werden standardmäßig drei Zeichen Lookahead für den Lexer und ein Wort Lookahead für den Parser verwendet. Zusätzlich können weitere Optionen direkt an Antlr[1] weitergereicht werden, indem diese als Zeichenkette nach dem Lookahead eingefügt werden. Zusätzlich zu den Optionen für Lexer und Parser gibt es die Möglichkeit, in einer Zeichenkette nach dem Schlüsselwort `header` Quellcode für den Antlr-Kopf zu übergeben. Abbildung 3.9 zeigt beispielhaft einen Optionenblock.

MontiCore-Grammatik

```
1 grammar Shop6 {
2
3   options {
4         lexer lookahead=2
5         parser lookahead=1
6   }
7
8   // ...
9
10 }
```

Abbildung 3.9: Definition von Optionen innerhalb einer Grammatik

Darüber hinaus können weitere Optionen innerhalb eines Optionenblocks in beliebiger Reihenfolge verwendet werden:

nows Standardmäßig werden Leerzeichen und Zeilenumbrüche während der lexikalischen Analyse ignoriert. Durch Setzen dieser Option können sie als Terminalzeichen in der Grammatik verwendet werden.

noslcomments Standardmäßig werden Zeichenketten, die mit „//“ beginnen, bis zum Zeilenende als Kommentar behandelt. Durch Setzen dieser Option wird dieses Verhalten deaktiviert und die Zeichen werden normal verarbeitet.

[1] (vgl. http://www.antlr2.org/doc/options.html für eine vollständige Auflistung)

nomlcomments Standardmäßig werden Zeichen innerhalb der Begrenzungen „/*“ und „*/“ als Kommentar behandelt. Dieses Verhalten wird durch das Setzen dieser Option deaktiviert und die Zeichen werden normal verarbeitet.

noanything Standardmäßig wird ein Identifier MCANYTHING generiert, der in keiner Parserproduktion benutzt wird. Dieses Vorgehen ist für die Einbettung verschiedener Sprachen ineinander notwendig, wie sie in Abschnitt 4.2 beschrieben wird. Dieses Verhalten wird mit Setzen der Option deaktiviert.

noident Standardmäßig wird ein Identifier IDENT generiert. Durch das Setzen der Option wird dieser Identifier nicht automatisch generiert.

nostring Standardmäßig wird ein Identifier STRING für Zeichenketten generiert. Durch das Setzen der Option wird dieser Identifier nicht automatisch generiert.

nocharvocabulary Standardmäßig wird das Unicode-Eingabealphabet zwischen den Zeichen \u0003 und \u7FFE für den Lexer verwendet. Dieses Verhalten wird mit Setzen der Option deaktiviert und kann dann in den Lexeroptionen ergänzt werden.

dotident Diese Option aktiviert einen erweiterten Identifier IDENT, der auch Punkte innerhalb eines Identifiers zulässt. Eine Deaktivierung der normalen Bezeichner über `noident` ist nicht notwendig.

compilationunit «IDENT» Diese Option sorgt für eine Einbindung der Grammatik in das DSLTool-Framework und erzeugt zusätzliche Produktionen für Paketinformationen und Imports innerhalb der Sprache. Der «IDENT» verweist auf eine Produktion der Grammatik, die ein Attribut Name vom Typ `java.lang.String` besitzt. Dieses ist z.B. der Fall, wenn eine Produktionskomponente `name:IDENT` verwendet wird. Diese Produktion wird nach dem Parsen von Paketnamen und Imports aufgerufen, bildet also die eigentliche Startproduktion der Grammatik.

header «String» Diese Option spezifiziert im «String» einen zusätzlichen Header, der in den generierten Parser eingebunden wird.

prettyprinter «QName» Diese Option spezifiziert in «QName» den vollqualifizierten Namen des PrettyPrinters, was eine automatische Kombination der PrettyPrinter bei Spracheinbettung erlaubt.

3.5.2 Prädikate

Durch die Bindung des MontiCore-Grammatikformats an den Parsergenerator Antlr stehen auch in MontiCore die Prädikate von Antlr zur Verfügung. Prädikate werden zur Ergänzung der Entscheidung innerhalb des Erkennungsprozesses benutzt und komplementieren das normale LL(k)-Verfahren mit festem Lookahead. Dadurch ist insbesondere ein lokales Backtracking möglich, was die Restriktionen eines LL-Parsers gegenüber einem LR-Parsers weitgehend aufhebt. Eine detaillierte Erklärung hierzu findet sich in [Par07].

Bei Prädikaten kann zwischen syntaktischen und semantischen Prädikaten unterschieden werden. Syntaktische Prädikate verwenden Teilausdrücke eines Produktionsrumpfs eingeschlossen in runden Klammern und um das Symbol `=>` ergänzt, um die Entscheidung, welche Alternative zu wählen ist, explizit zu formulieren. Die Abbildung 3.10 zeigt in Zeile 9 ein Beispiel für ein syntaktisches Prädikat. Die Produktion `PremiumClient` lässt sich nicht mit endlichem Lookahead von einer `Client`-Produktion unterscheiden. Der Grund

MontiCore-Grammatik

```
1 grammar Shop7 {
2
3   ShopSystem =
4     Name:IDENT Client* Order*;
5
6   Client =
7     "client" Name:IDENT* Address;
8
9   PremiumClient extends ("client" IDENT* NUMBER)=> Client =
10    "client" Name:IDENT* Discount:NUMBER Address;
11
12  Address =
13    Street:STRING Town:STRING;
14
15  interface Order;
16
17  OrderCreditcard implements {LT(1).getText().startsWith("cc")}? Order =
18    TransNumber:IDENT ClientName:IDENT BillingID:NUMBER;
19
20  OrderCash implements Order =
21    TransNumber:IDENT ClientName:IDENT Amount:NUMBER;
22 }
```

Abbildung 3.10: Verwendung von Prädikaten in MontiCore

hierfür ist die unbegrenzte Anzahl an `IDENT`s, die eine Unterscheidung mit festem Lookahead unmöglich machen, weil erst nach dem Erkennen der zu Entwicklungszeit unbekannten Anzahl an `IDENT` der Typ des Tokens danach den Unterschied ausmacht. Wird ein `STRING` (aus `Address`) erkannt, ist es ein `Client`, bei einer `NUMBER` handelt es sich um einen `PremiumClient`. Eine Unterscheidung beider Produktionen ist aufgrund der Vererbung notwendig, weil `Client` und `PremiumClient` erkannt werden, wenn das Nichtterminal `Client` wie zum Beispiel in Zeile 4 verwendet wird. Der Ausdruck `("client" IDENT* NUMBER)=>` bezeichnet somit das Unterscheidungsmerkmal, wann ein `PremiumClient` als `Client` erkannt werden soll. Wird dieser Ausdruck erkannt, wird von einem `PremiumClient` ausgegangen, andernfalls wird versucht, einen normalen `Client` zu erkennen. Somit lässt sich auch ein potentiell unendlicher Lookahead wie hier für `IDENT*` formulieren, welcher sich mit endlichem Lookahead nicht entscheiden lässt.

Semantische Prädikate hingegen erlauben die Verwendung eines Java-Ausdrucks zur Spezifikation einer eigenen Lookahead-Entscheidung. Semantische Prädikate werden durch geschweifte Klammern eingeschlossen und mit einem Fragezeichen abgeschlossen. Dabei kann der Inhalt über die syntaktischen Möglichkeiten hinaus verwendet werden. Im vorliegenden Beispiel unterscheidet das Prädikat `{LT(1).getText().startsWith("cc")}?` in Zeile 17 die verschiedenen Orders untereinander, so dass `CreditcardOrder` eine Transaktionsidentität haben müssen, die mit „cc“ beginnt. Eine Produktion wird hier also nur verwendet, falls der entsprechende Java-Ausdruck innerhalb des Prädikats nach wahr ausgewertet werden kann. Dabei wird die Funktion LT(n) genutzt, die das n-te Token vom aktuellen Erkennungspunkt aus zurückliefert.

Syntaktische und semantische Prädikate können nicht nur zur Unterscheidung von Produktionen bei Nichtterminalvererbung sondern auch am Anfang jeder Alternative innerhalb einer Produktion eingesetzt werden. Dabei ist zu beachten, dass nicht am Anfang einer Alternative eingesetzte semantische Prädikate als semantische Checks interpretiert werden, d. h. nicht zur Unterscheidung verschiedener Alternativen dienen, sondern bei der Auswertung nach *falsch* einen Erkennungsfehler auslösen.

3.5.3 Variablen und Produktionsparameter

Neben der Möglichkeit Elemente explizit zu benennen und so den Namen des Attributs anzugeben, kann auch ein Variablenname getrennt durch ein Gleichheitszeichen vor einem Terminal oder Nichtterminal verwendet werden. Die Variable enthält nach der Zuweisung den Wert des Terminals oder Nichtterminals. Bei der Verwendung eines kleenschen Sterns oder `+` an der Variablen wird eine Liste mit allen erkannten Instanzen abgelegt. Variablen werden nicht explizit definiert, sind jedoch getypt. Das bedeutet in diesem Zusammenhang, dass dieselben Typisierungs- und Ableitungsregeln wie für Attribute gelten. Variablen können, sofern die Typisierung stimmt, jedoch mehrfach einen Wert zugewiesen bekommen, wobei der alte Wert durch den neuen ersetzt wird. Der Gültigkeitsbereich einer Variablen ist auf eine Produktion beschränkt. In Abbildung 3.11 wird eine Variable `n` in Zeile 4 mit dem Wert der erkannten `IDENT` als Liste belegt.

Eine Anwendung für Variablen sind so genannte Produktionsparameter. Dabei besitzt eine Produktion eine Menge von Parametern, die am Produktionsanfang in speziellen Klammern (`[| |]`) angegeben werden. Diese Parameter müssen bei der Verwendung des Nichtterminals mit dem Wert von Variablen belegt werden, wie dieses in Zeile 4 durch die Variable `n` geschieht. Alternativ kann der Standardwert `null` verwendet werden. Bei der Spezifikation der Parameter wird die übliche Nichtterminalnotation verwendet, d. h. die Nichtterminale können benannt sein und werden somit direkt nach der Wertübergabe zu Attributen der Klasse. Es ist auch möglich, in den Produktionsparametern wiederum Variablen zu definieren und diese an eine weitere Produktion zu übergeben.

Der Wert einer Variablen wird bei der Wertübergabe automatisch gelöscht. Des Weiteren ist es nicht möglich, eine Variable gleichzeitig auch als Attribut zu nutzen. Durch diese beiden Maßnahmen ist sichergestellt, dass jeder erkannte Knoten nur einmal im AST auftaucht und Variablen nicht genutzt werden können, um Knoten zu duplizieren.

In Abbildung 3.11 befindet sich eine alternative Version des ShopSystems, bei dem im Sinne einer Linksfaktorisierung das Schlüsselwort `"client"` und der Name von `Client` und `PremiumClient` extrahiert wurden (Zeile 4). Die Prädikate der vorherigen Version sind damit überflüssig. Der Name wird der Produktion übergeben und dort durch die Notation als Attribut direkt zu einem Attribut der Klasse `Client` bzw. `PremiumClient`. Die abstrakte Syntax des ShopSystems bleibt dadurch unverändert, d. h. konkret ist der Name nicht ein Attribut der Klasse `ShopSystem`, auch wenn er innerhalb dieser Produktion erkannt wird, sondern ein Teil der Klasse `Client` bzw. `PremiumClient`.

Wie das Beispiel gut zeigt, müssen die Parameter von Unterproduktionen kompatibel zu den Produktionsparametern der Oberklassen sein. Konkret muss also jeder formale Parameter einer Unterklasse denselben Typ wie die Parameter in der Oberklasse haben. Eine Gleichbehandlung insofern, als die formalen Parameter als Variablen oder Attribute weiter verarbeitet werden, ist nicht notwendig.

Das Beispiel wurde so verändert, dass eine Produktion `ComplexOrder` eingeführt wurde, die eine Zusammenfassung verschiedener `Order` zu einer einzigen darstellt. Es soll aber erreicht werden, dass das Shopsystem nicht aus vielen `ComplexOrder` besteht, die ihrerseits nur eine einzige `Order` umfassen. Für diesen einfachen Fall soll weiterhin nur eine `OrderCash` oder `OrderCreditcard` entstehen. Die Variable `ret` hat eine spezielle Bedeutung, denn sie repräsentiert den Rückgabewert einer Produktion. Durch Wertzuweisungen an `ret` kann gezielt das Rückgabeobjekt einer Produktion verändert werden. In Zeile 18 wird der Rückgabewert so manipuliert, dass er sich direkt aus dem Wert der Produktion `Order` ergibt, wenn die erste Alternative ausgewählt wird. Durch das Schlüsselwort `returns` kann ein anderer Rückgabetyp als der eigentlich durch die Klassenproduktion de-

MontiCore-Grammatik

```
1 grammar Shop8 {
2
3   ShopSystem =
4     name:IDENT ("client" n=IDENT* Client[|n|])* ComplexOrder*;
5
6   Client [|name:IDENT*|] =
7     Address;
8
9   PremiumClient extends Client [|name:IDENT*|] =
10     discount:NUMBER;
11
12  Address =
13    street:STRING town:STRING;
14
15  interface Order;
16
17  ComplexOrder returns Order =
18    ret=Order | "{" Order ("," Order)* "}";
19
20  OrderCash implements Order =
21    "cashorder" clientName:IDENT amount:NUMBER;
22
23  OrderCreditcard implements Order =
24    "creditorder" clientName:IDENT billingID:NUMBER;
25 }
```

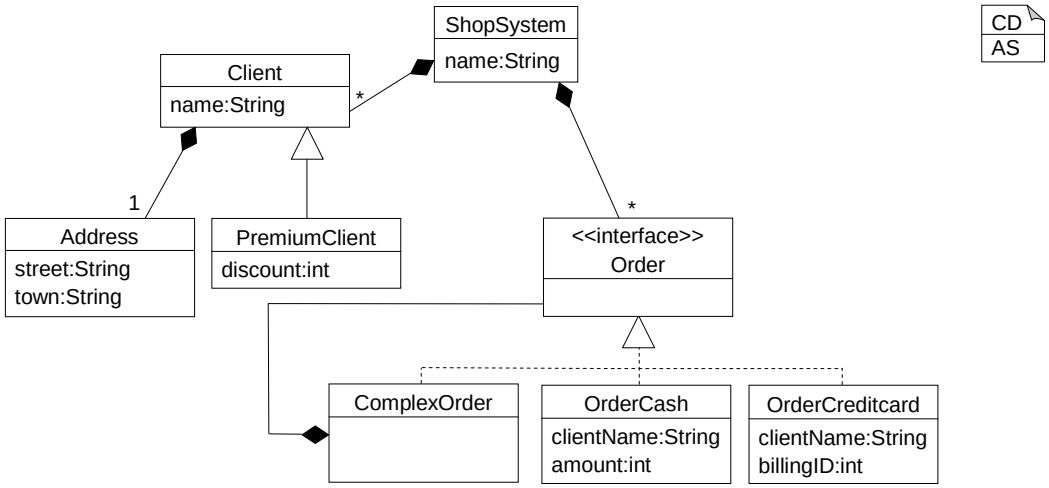


Abbildung 3.11: Verwendung von Variablen und Produktionsparametern

finierte Typ festgelegt werden. Die Festlegung mit `returns Order` schließt ein implizites `astimplements Order` (bzw. `astextends Order` falls `Order` eine Klasse wäre) mit ein, so dass die korrekte Typisierung automatisch sichergestellt wird. Es ist jedoch zu beachten, dass nun Attribute, die sich auf die Produktion `ComplexOrder` beziehen, den Typ hinter dem Schlüsselwort returns annehmen und nicht mehr den der Produktion selbst. Im Beispiel ist das `ShopSystem` daher eine Komposition von `Order`, obwohl das Nichtterminal `ComplexOrder` in Zeile 4 verwendet wird.

3.5.4 Aktionen

Aktionen sind Java-Statements, die ausgeführt werden, wenn eine bestimmte Stelle beim Parsen überschritten wird. Die Java-Statements werden ungeprüft in die Parsergenerierung und somit in den resultierenden Parser übernommen. Dabei ist zu beachten, dass während der Auswertung von Prädikaten keine Aktionen ausgeführt werden, sondern nur bei erfolgreicher Erkennung eines Ausdrucks. Aktionen werden durch geschweifte Klammern eingeschlossen und können an beliebigen Stellen innerhalb der Produktionen stehen. Initiale Aktionen unterscheiden sich von Aktionen dadurch, dass sie nur am Anfang eines Blocks stehen dürfen. Sie werden syntaktisch durch das Schlüsselwort `init` gefolgt von einer normalen Aktion konstruiert. Initiale Aktionen werden im Gegensatz zu normalen Aktionen auch bei der Auswertung von Prädikaten ausgeführt.

Eine kontrolliertere Variante der Aktionen stellt das Astscript dar, welches bestimmte Operationen anbietet, für die auch Kontextbedingungen geprüft werden können. Das Astscript wird durch das Schlüsselwort `astscript` gefolgt von den Anweisungen verwendet, die in geschweiften Klammern eingeschlossen werden. Dabei sind die folgenden Anweisungen implementiert:

Konstruktor: `!( «Zuweisung»* ) ;` Der Konstruktor erzeugt ein Objekt vom Typ der aktuellen Produktion und setzt den Rückgabewert auf dieses Objekt. Dabei können gleichzeitig verschiedene Zuweisungen an dieses Objekt gemacht werden, um die Attribute des neuen Objekts mit Werten von Variablen zu belegen.

Zuweisung: `«Att» [=|+=] «Var» ;` Dem Attribut `Att` wird der Wert der Variablen `Var` zugewiesen (`=`) oder an den aktuellen Wert angefügt, wenn es sich um eine Liste handelt (`+=`).

Setzen eines globalen Wertes: `push («ID»,«Var»)` Diese Anweisung fügt dem globalen Stack mit der Identität `ID` den Wert der Variablen `Var` zu. Die globalen Stacks sind mit `java.lang.String` typisiert, so dass für komplexe Datentypen die `toString()`-Methode verwendet wird. Diese Operation wird insbesondere bei der parametrisierten Einbettung wichtig, wie sie in Abschnitt 4.2 erklärt wird.

Entfernen eines globalen Wertes: `pop («ID»)` Diese Anweisung entfernt das oberste Datenelement des globalen Stacks mit der Identität `ID`.

Jeder Astscript-Block kann dabei am Anfang durch das Schlüsselwort `once` gekennzeichnet werden. Dadurch wird die Anweisung nur einmal innerhalb des Erkennungsprozesses einer Produktion ausgeführt und nicht wiederholt, wenn er sich zum Beispiel innerhalb eines Blocks mit kleenschem Stern befindet.

3.5.5 Mehrteilige Klassenproduktionen

In den bisher dargestellten Grammatiken wurde stets durch eine Klassenproduktion eine einzelne Klasse definiert. In speziellen Situationen, insbesondere für Ausdrücke (vgl. Abschnitt 3.5.6), ist es aber praktisch, diese strikte Eins-zu-Eins-Beziehung aufzubrechen und mehrere Produktionen zur Definition einer Klassen zu nutzen. Dazu kann nach dem Namen einer Klassenproduktion getrennt durch einen Doppelpunkt noch ein Bezeichner verwendet werden, der auf die entsprechende Klasse der abstrakten Syntax referenziert. Die Klasse wird aus der Vereinigung aller Produktionen gebildet, die diesen Namen verwenden. Innerhalb der Produktionsrumpf werden aber weiterhin die Produktionsnamen verwendet, um gezielt auf eine der Produktionen zu verweisen.

Die Abbildung 3.12 zeigt ein Beispiel, in dem die beiden Produktionen `Client` und `PremiumClient` verwendet werden, um den gemeinsamen Typ `Client` zu definieren. Die Attribute des Typs ergeben sich so, als wären alle Produktionen durch Alternativen verbunden worden. In diesem Zusammenhang beziehen sich AST-Regeln immer auf den Typ, so dass dort der Name des Typs und nicht einer der Produktionen verwendet werden muss.

MontiCore-Grammatik

```
1  grammar Shop9 {
2
3    ShopSystem =
4      (Client | PremiumClient)* ;
5
6    Client =
7      "client" name:IDENT Address;
8
9    PremiumClient:Client =
10     premium:["premiumclient"]
11     name:IDENT discount:NUMBER Address;
12
13 }
```

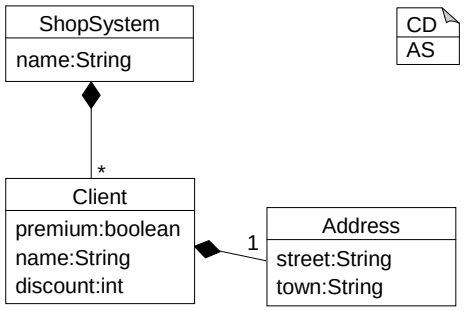


Abbildung 3.12: Mehrteilige Klassenproduktionen

Das Beispiel verdeutlicht nochmals die automatische Namensableitung für Kompositionen, die sich aus dem Namen der referenzierten Klasse ergibt und nicht aus dem Namen des Nichtterminals. Daher gibt es nur eine Kompositionsbeziehung zwischen `ShopSystem` und `Client`, da die beiden Nichtterminale `Client` und `PremiumClient` beide zur Komposition `Client` beitragen.

3.5.6 Ausdrücke

Ausdrücke in Programmier- und Modellierungssprachen unterscheiden sich von anderen Teilen der Sprache meistens dadurch, dass eine Vielzahl an AST-Elementen existiert, die fast beliebig zu einer Baumstruktur kombiniert werden können. In der konkreten Syntax werden wenige Schlüsselwörter verwendet und eine Klammerung kann aufgrund der unterschiedlich starken Bindung der Operationen oft vermieden werden. Bei der Realisierung von Ausdrucksstrukturen in MontiCore wurden zwei Ziele verfolgt:

- Die Baumstruktur soll nach dem Parsen nur AST-Knoten für wirklich aufgetretene Operatoren enthalten. Zwischenknoten wie eine Addition mit nur einem Argument, was bedeutet, dass tatsächlich keine Addition stattfindet, sondern der Knoten nur aufgrund der syntaktischen Struktur der Sprache erzeugt wird, sollten vermieden werden.
- Es soll möglich sein, Operationen verschiedener Prioritäten innerhalb einer Klasse zu realisieren. Dadurch lässt sich die Anzahl der beteiligten Klassen reduzieren und die Struktur verständlich halten. Somit können zum Beispiel die Addition und die Multiplikation zu einer Infix-Expression zusammengefasst werden.

Die Abbildung 3.13 zeigt die Realisierung einer einfachen Ausdruckssprache, die die üblichen Rechenoperationen für natürliche Zahlen zur Verfügung stellt. Dabei wird zunächst neben der lexikalischen Struktur die Schnittstellenproduktion `Expression` definiert. Die

MontiCore-Grammatik

```
1  grammar Expression {
2
3    token NUMBER = ('0'..'9')+ : int;
4
5    interface Expression;
6
7    AdditionSubtraction:InfixExp implements Expression returns Expression =
8      ret=MultDiv ( astscript {!(left=ret;);} op:["+"|"-"] right:MultDiv)*;
9
10   MultDiv:InfixExp returns Expression =
11     ret=Literal (astscript {!(left=ret;);} op:[MULT:"*"|DIV:"/"] right:Literal)*;
12
13   ast InfixExp =
14     left:Expression
15     right:Expression;
16
17   interface Literal astextends Expression;
18
19   NumLiteral implements Literal astimplements Expression  =
20     num:NUMBER;
21
22   Variable implements Literal astimplements Expression =
23     name:IDENT;
24
25 }
```

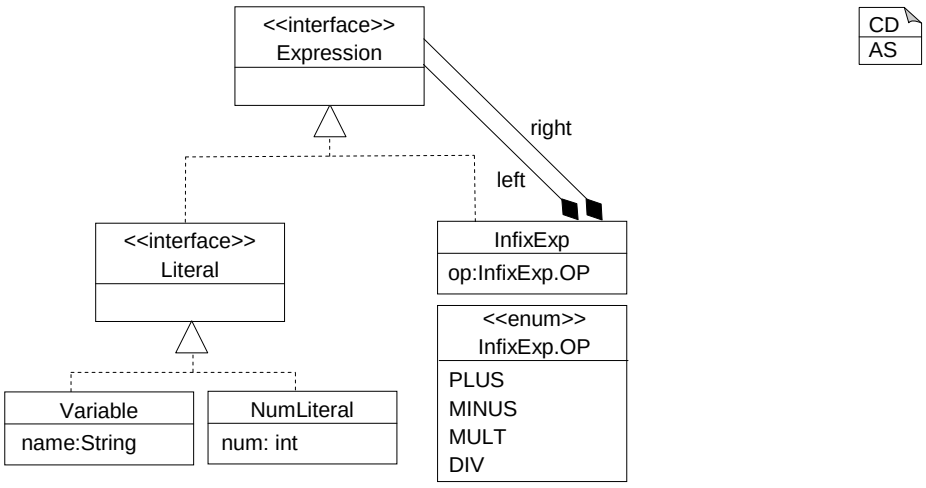


Abbildung 3.13: Realisierung von Ausdrücken in MontiCore

Klassenproduktion `AdditionSubtraction` ist die einzige Produktion, die diese auch in der konkreten Syntax implementiert (erkennbar durch `implements Expression`). Dadurch wird `Expression` direkt nach `AdditionSubtraction` abgeleitet. Die beiden Produktionen `AdditionSubtraction` und `MultDiv` definieren zusammen die Klasse `InfixExp`. Die Attribute `left` und `right` werden durch eine entsprechende AST-Produktion festgelegt. Der Operator ergibt sich aus den implizit angegebenen Werten, wobei für die Multiplikation und Division eigene Konstantennamen festgelegt wurden, weil sich die automatische Namensableitung am Namen des Zeichens orientiert und `STAR` und `SLASH` hier unpassend wären.

Bei der Realisierung von Ausdrücken ist die Beachtung der Priorität von Operatoren wichtig. Dabei wird zunächst mit der schwächsten Priorität begonnen, also dem Operator, der am wenigsten bindet. Die Produktionen haben dabei die folgende Form, wobei `«Px»` die aktuelle Operation mit Operatorsymbol `«OP»` bezeichnet und `«Px'»` die Operation, die den direkt stärker bindenden Operator verwendet:

```
ast «Px» = left:Expression right:Expression;
«Px» returns Expression =
    ret=«Px'» (astscript !(left=ret) op:["«OP»"] right:«Px'»)*.
```

Durch den Einsatz der Variablen `ret` und des Astscripts werden zwei Effekte erreicht. Erstens wird, wenn der Operator für die betrachtete Priorität nicht verwendet wird, direkt ein Ausdruck mit niedriger Priorität zurückgeliefert. Dabei entsteht kein Zwischenknoten. Zweitens, wenn der Operator auftritt, entsteht eine linksassoziative Baumstruktur durch die Instanzierung eines entsprechenden AST-Knotens und der gleichzeitigen Zuweisung des aktuellen Rückgabewerts als linkes Kind.

Über das beschriebene Schema lässt sich nun für jede Prioritätsstufe eine oder mehrere Klassen definieren, so dass eine Baumstruktur ohne unbesetzte Zwischenknoten entsteht. Ist es zusätzlich gewünscht, dass verschiedene Prioritätsstufen dieselbe Klasse verwenden, wie zum Beispiel die Zusammenfassung von Addition und Multiplikation zu einer `InfixExp`, dann können mehrteilige Klassenproduktionen verwendet werden. Das obige Schema ändert sich insofern, als die Namen der Produktionen unverändert bleiben, jedoch auf den beteiligten Stufen `InfixExp` als Produktionstyp hinzugefügt wird.

Die Abbildung 3.14 zeigt drei Objektstrukturen zu Ausdrücken, die beim Parsen entstehen. Dabei ist zu erkennen, dass keine Zwischenknoten aufgrund der syntaktischen Struktur der Sprache entstehen.

Die hier dargestellten Möglichkeiten erlauben die Realisierung von Ausdrucksstrukturen, wie sie so in keinem anderen modellbasierten Ansatz möglich sind. Die Konstruktion der Produktionen kann nach klaren Regeln durchgeführt werden. Aufbauend auf diesen Möglichkeiten zur Spezifikation kann ein Konzept entworfen werden, das die Spezifikation der Ausdrucksteilsprache mit Hilfe der Priorität, der Assoziativität und der Arität des Operators und dem Operatorsymbol erlaubt. Die technische Realisierung geschieht dann über die automatische Konstruktion der beschriebenen Produktionen.

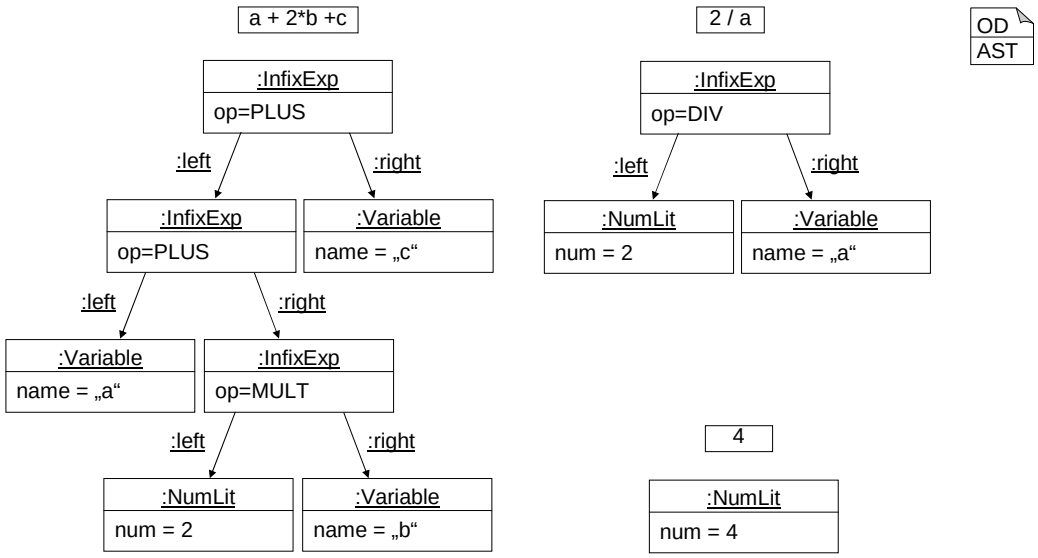


Abbildung 3.14: Beispiele für enstehende ASTs beim Parsen von Ausdrücken

3.5.7 Schlüsselwörter

Die Parsererzeugung von MontiCore basiert auf einem zweischrittigen Parseverfahren mit Lexer und Parser. Dabei ist es üblich, Schlüsselwörter einer Sprache durch den Lexer erkennen zu lassen. Diese sind dann keine validen Bezeichner mehr innerhalb der Sprache. Für Programmiersprachen ist dieses Verhalten durchaus erwünscht, weil so die Lesbarkeit der Instanzen erhöht wird und die Menge der validen Bezeichner nur unwesentlich eingeschränkt wird. Für DSLs kann dieses teilweise ein Problem sein, insbesondere wenn verschiedene DSLs kooperativ eingesetzt werden, weil sich so ein große Anzahl von Schlüsselwörter ergeben kann.

Daher erlaubt das MontiCore-Grammatikformat nach jedem Nichtterminal, das sich auf eine lexikalische Produktion bezieht, das Zeichen `&` einzufügen. Dieses ist besonders an den Stellen sinnvoll, an denen sich Bezeichner auf Elemente einer anderen DSL beziehen. Dadurch werden an dieser Stelle nicht nur Token erkannt, die zur lexikalischen Produktion konform sind, sondern auch alle Schlüsselwörter akzeptiert. Dieses Vorgehen erlaubt eine knappere Formulierung der Grammatik als die Schlüsselwörter an diesen Stellen einzeln als Alternativen aufzuführen.

3.5.8 Blockoptionen

Blockoptionen dienen dazu, den Parseprozess zu beeinflussen. Die wichtigste Option ist das so genannte Greedy- bzw. Nongreedy-Verhalten bei Mehrdeutigkeiten in einer Grammatik. Darunter versteht man zum Beispiel das Verhalten eines Parsergenerators, wenn er sich zwischen einer Fortsetzung innerhalb einer Umgebung mit kleenschem Stern oder optionalen Blöcken und der darauf folgenden syntaktischen Struktur entscheiden muss. Die bekannteste Problemstellung in diesem Zusammenhang ist das so genannte „Dangling-Else-Problem“: Bei der Eingabe `if(true) if(true) a() else b();` ist nicht klar, auf welche *if*-Anweisung sich das *else* beziehen soll, wobei in den meisten Programmiersprachen die erste Variante gültig ist:
(1) `if(true) { if(true) a() else b(); }` oder
(2) `if(true) { if(true) a() } else b();`

In einer Grammatik bedeutet ein solches Problem prinzipiell eine Mehrdeutigkeit, d. h. beim Erkennen der Produktionen ist am Anfang der Zeile 3 in Abbildung 3.15 unklar, ob der optionale Block betreten werden soll und so das *else* zum zweiten *if* gebunden werden soll (greedy) oder der Block zunächst nicht betreten wird und das *else* an das erste *if* gebunden wird (nongreedy). Antlr analysiert diese Mehrdeutigkeit und warnt prinzipiell immer bei einer solchen Produktionsstruktur. Das Parseverhalten ist ohne Angabe von Optionen dann greedy. Durch den Optionenblock `options {greedy=true;}` : werden die Warnungen entfernt. Durch den Optionsblock `options {greedy=false;}` : ändert sich das Parseverhalten und es wird ein Nongreedy-Verhalten durchgeführt, sofern der fest eingestellte Lookahead eine zweite Alternative erkennt.

MontiCore-Grammatik

```
1 IFStatement implements Statement =
2   "if" "(" Expression ")" Statement
3     ( options {greedy=true;} : "else" Statement)?;
```

Abbildung 3.15: „Dangling-Else-Problem“

3.6 Zusammenfassung

In diesem Kapitel wurde das MontiCore-Grammatikformat beschrieben. Es dient innerhalb der agilen modellgetriebenen Entwicklung zur Spezifikation der konkreten und abstrakten Syntax einer DSL. Die Struktur der abstrakten Syntax orientiert sich an objektorientierten Datenstrukturen und erlaubt so die einfache Umsetzung in eine Programmiersprache. Dadurch wird die Integration der Grammatiken in eine quellcodebasierte Entwicklung vereinfacht, und die Programmierung von Algorithmen und Codegenerierungen für DSLs ist auf eine effiziente Art möglich.

Mit der Vererbung von Nichtterminalen und der Verwendung von Schnittstellen, abstrakten Klassen und Assoziationen wurden Elemente der Metamodellierung in ein grammatikbasiertes Format integriert und gezeigt, wie sich diese gleichzeitig auf die konkrete und abstrakte Syntax auswirken. Zusätzlich wurden weiterführende Elemente entwickelt, die eine Erstellung einer Sprachdefinition vereinfachen. Mit mehrteiligen Klassenproduktionen wurde eine flexible Möglichkeit geschaffen, Ausdruckteilsprachen zu realisieren, die in anderen modellgetriebenen Ansätzen so nicht zu spezifizieren sind.

Kapitel 4

Modulare Entwicklung von domänenspezifischen Sprachen

Die modulare Entwicklung von Softwaresystemen hat sich innerhalb der Informatik als ein erfolgreiches Konzept zur Reduzierung der Komplexität einzelner Aufgaben etabliert und erlaubt das kooperative, zeitgleiche Erstellen von Software durch mehrere Personen. Diese Arbeitsteilung ermöglicht überhaupt erst die Erstellung komplexer Softwaresysteme. Grammatikbeschreibungssprachen unterscheiden sich in dieser Hinsicht kaum von Programmier- und Modellierungssprachen, so dass auch hier eine modulare Erstellung vorteilhaft ist.

Die Grundvoraussetzung für die Nutzung von Modularität ist das Frege-Prinzip, welches besagt, dass sich die Bedeutung des Kompositums nur aus der Bedeutung seiner Teile und der Kompositionsform ergibt. Dadurch lassen sich die einzelnen Teile getrennt betrachten und ihre Bedeutung geht bei der Integration nicht verloren. Die explizite Einführung von Modulschnittstellen, die im Sinne des „information hidings“ [Par72] nur einen Teil der Funktionalität außerhalb des Moduls verfügbar machen, erleichtert die kooperative Erstellung von Software. Für die Integration der verschiedenen Module ist dann nur die Kenntnis der Schnittstellen notwendig und kein Wissen über interne Details. Dieses ermöglicht in modernen Programmiersprachen die Erstellung von Klassenbibliotheken, die qualitätsgesicherte Module für den Entwickler zur Wiederverwendung bereitstellen.

Modularität innerhalb einer Sprachdefinition hilft, die Komplexität der Problemstellung dadurch zu verringern, dass nur kleine, in sich konsistente Elemente entwickelt werden müssen. Diese Module können dann getrennt verstanden, qualitätsgesichert und in verschiedenen Kontexten ohne Veränderung wieder verwendet werden, ohne dass der Entwickler Details kennen muss, die über eine Schnittstellenbeschreibung hinausgehen. Das MontiCore-Grammatikformat unterstützt drei Arten der Modularität, die für verschiedene Zwecke verwendet werden können:

Grammatikvererbung. Grammatiken können voneinander erben und dabei neue Produktionen hinzufügen oder existierende ersetzen. Somit können existierende Sprachen an neue Bedürfnisse angepasst werden, wobei nur die Modifikationen beschrieben werden müssen.

Einbettung. Spracheinbettung erlaubt die unabhängige Definition von Grammatikfragmenten mit nicht definierten Nichtterminalen. Die Fragmente können komponentenbasiert miteinander zu Sprachen kombiniert werden, indem nicht definierte Nichtterminale durch Nichtterminale anderer Fragmente ersetzt werden.

Konzepte. Das Grammatikformat selbst ist durch Konzepte erweiterbar, so dass zusätzliche Informationen in der Sprachdefinition aufgeführt werden und die Grundlage für weiterführende Generierungsschritte bilden können.

Die modulare Definition wirkt sich zunächst auf die abstrakte und konkrete Syntax der jeweiligen Sprachen aus, wie in den folgenden Abschnitten beschrieben wird. Weitere wichtige Elemente wie Kontextbedingungen, die Semantik der Sprache, Analysen und Codegenerierungen können auch modular entwickelt werden, damit sich nicht nur neue Sprachen aus bereits entwickelten Bausteinen zusammensetzen lassen, sondern auch darauf basierte Werkzeuge. In Kapitel 9 wird die modulare Entwicklung einiger weiterer Elemente besprochen.

Die Mechanismen greifen die in [MHS05] beschriebenen Entwurfsmuster zur Entwicklung von DSLs auf und bieten technische Möglichkeiten zu deren Realisierung. Zusätzlich ist die Sprachdefinition, wie in [FR07] empfohlen, durch die Konzepte erweiterbar. Sie kann neben der Definition der konkreten und abstrakten Syntax für bestimmte Anwendungsbereiche spezifische Elemente enthalten, die die Grundlage für weitere Generierungen bilden.

Dieses Kapitel basiert teilweise auf dem Konferenzpapier [KRV08]. Die Ausführungen wurden überarbeitet und an den aktuellen Stand der MontiCore-Entwicklung angepasst.

4.1 Grammatikvererbung

Grammatikvererbung kann benutzt werden, um eine existierende Sprache dadurch zu erweitern, dass der Unterschied zwischen der ursprünglichen und der neuen Sprache angegeben wird. Die ursprüngliche Sprachdefinition bleibt dabei unverändert und kann so für andere Projekte unverändert verwendet werden.

Ein leicht verständliches Beispiel für eine Spracherweiterung ist die Verwendung von SQL-Select-Anweisungen als Ausdrücke innerhalb einer GPL, wie es erfolgreich in [MBB06] gezeigt wurde. Unter der Annahme, dass bereits Grammatiken für beide Sprachen existieren, würde ein monolithischer Ansatz beide zu einer auch syntaktisch veränderten Grammatik kombinieren. Dieses Vorgehen steigert die Komplexität der Artefakte und macht eine getrennte Qualitätssicherung und Weiterentwicklung schwierig.

MontiCore verwendet mehrfache Grammatikvererbung. Dabei kann eine neue Grammatik gebildet werden, die von mehreren Sprachen erbt und somit alle Produktionen der Obergrammatiken übernimmt. Hierfür ist jedoch keine syntaktische Komposition beider Grammatiken notwendig, sondern Verweise auf die Obergrammatiken sind ausreichend. Auf diese Weise können unabhängig entwickelte Grammatiken kombiniert werden und so die konkrete und abstrakte Syntax einschließlich des Parsers und der AST-Klassen erzeugt werden.

Methodisch ist es nur sinnvoll, Grammatikvererbung anzuwenden, wenn die gewünschte Spracherweiterung ähnlich zur gewählten Grundsprache ist. Neue Produktionen können prinzipiell auch die lexikalische Analyse verändern, nur gilt dann diese Änderung sowohl für die neuen als auch die alten Teile. Im hier dargestellten Java/SQL-Beispiel fügen die SQL-Produktionen auch neue Schlüsselwörter wie `SELECT` und `FROM` zur kombinierten Sprache hinzu, die dann auch im Java-Anteil keine validen Bezeichner mehr sind. Diese Situation und die damit verbundenen Konsequenzen sind sehr ähnlich zur Einführung des Schlüsselworts `assert` und den nachfolgenden Änderungen am existierenden Quellcode bei Java 1.4. Wenn dieses Vorgehen schwerwiegende Probleme hervorruft oder sich zwei Sprachen in der lexikalischen Analyse gegenseitig stark widersprechen, sollte Einbettung verwendet werden. Dabei sind die einzelnen Lexer und Parser voneinander unabhängig, so dass die beschriebene Schlüsselwortproblematik nicht auftritt.

Einbettung kann mit der Grammatikvererbung derart kombiniert werden, dass durch die Vererbung ein neuer Erweiterungspunkt in Form eines externen Nichtterminals zur Sprache hinzugefügt wird. Dieser Erweiterungspunkt wird dann durch Einbettung genutzt, um zwei strukturell unterschiedliche Sprachen kombinieren zu können.

Die Abbildung 4.1 zeigt vereinfachend die Grammatikvererbung für die Kombination von Java und SQL. Um Java mit einer neuen Form von Expression (SQL Select Statement) zu erweitern, erbt die Grammatik `JavaSQL` von beiden Obergrammatiken, indem das Schlüsselwort `extends` benutzt wird. Danach werden durch Kommata getrennt die Obergrammatiken angegeben, wobei der vollqualifizierte Name der Grammatiken verwendet wird. Dieser ergibt sich aus der Paketbezeichnung, die wie in Java üblich am Anfang der Datei durch eine punktgetrennte Bezeichnung nach dem Schüsselwort `package` angegeben wird, und dem Namen nach dem Schlüsselwort `grammar`. Dabei fügt die Vererbung zu der Vereinigungsmenge aller Produktionen eine neue Produktion hinzu, die `SQLSelect` als eine spezielle Java-Expression definiert. In unserer Grammatik überschreibt die Produktion `SQLSelect` die geerbte Produktion `SQLSelect` aus der SQL-Grammatik durch Hinzufügen einer neuen Schnittstelle `Expression` aus der Java-Grammatik. Dabei entsteht eine Vererbungsbeziehung zwischen den Klassen `mc.sql.SQL.SQLSelect` und `mc.javasql.Java.SQLSelect`. Zusätzlich implementiert `mc.javasql.JavaSQL.SQLSelect` die Schnittstelle `mc.java.Java.Expression`.

MontiCore-Grammatik

```
1 package mc.javasql;
2
3 grammar JavaSQL extends
4   mc.java.Java, mc.sql.SQL {
5
6   SQLSelect implements Expression;
7
8 }
```

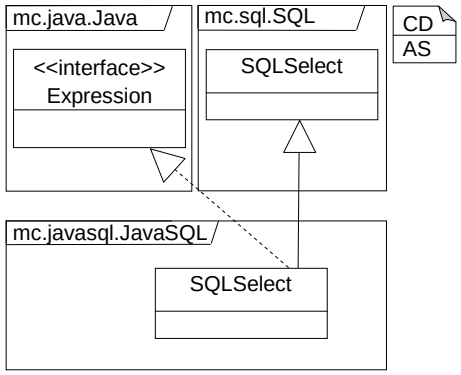


Abbildung 4.1: Mehrfache Grammatikvererbung

Für das Grammatikformat gilt generell, dass die Produktionen der Untergrammatik sich aus allen Produktionen der Obergrammatiken ergeben. Verwendet die Untergrammatik einen Produktionsnamen, der in einer Obergrammatik vorkommt, verschattet die neue Produktion die alte. Bei Parserproduktionen gibt es die Möglichkeit, einen neuen Produktionsrumpf zu spezifizieren. Dieser wird dann anstatt des Alten verwendet. In der abstrakten Syntax entsteht dann eine Unterklasse der aus der ursprünglichen Produktion abgeleiteten Klasse. In Abbildung 4.1 hingegen bleibt der Rumpf der Produktion `SQLSelect` unverändert, weil er nicht weiter beschrieben wird. Dieses bedeutet aber keine Epsilon-Produktion, sondern die Übernahme des Produktionsrumpfs aus der Obersprache, ohne diesen nochmals explizit aufführen zu müssen.

Die Grammatik in Abbildung 4.1 zeigt weiterhin, dass jede neue Produktion auch zu einer neuen Klasse in der abstrakten Syntax führt. MontiCore sichert, dass die Klasse, die durch die überschreibende Produktion definiert wird, eine Unterklasse der überschriebenen ist. Daher können alle Klassen, die sich auf die ursprüngliche beziehen (zum Beispiel durch Kompositionen und Assoziationen), unverändert bleiben. Dieser Ansatz hat Vorteile

gegenüber einer kompletten Neuerzeugung der AST-Klasssen für die neue Sprache, weil so Algorithmen für die Obersprache teilweise ohne Änderungen verwendet werden können.

Ein bekanntes Problem bei Mehrfachvererbung sind Namenskonflikte, wenn unterschiedliche Obergrammatiken dieselben Produktionsnamen verwenden. Beispielsweise in [Sim05] werden die Möglichkeiten für objektorientierte Sprachen ausführlich diskutiert. Für das MontiCore-Grammatikformat wurde folgende Strategie verwendet: In dem Fall, dass zwei oder mehr Obergrammatiken sich einen Produktionsnamen teilen, wird die Produktion aus der ersten Grammatik mit diesem Produktionsnamen verwendet (geordnete Vererbung). Treten dabei Konflikte aufgrund von sich widersprechenden Attributen oder der Restriktionen bezüglich der Mehrfachvererbung in den AST-Klassen auf, ist eine solche Sprachkombination ungültig. Beim Entwurf der oben dargestellten Lösung wurde sich explizit gegen eine Auflösungsstrategie wie in C++ entschieden, bei der Entwickler explizit auf eine Superklasse durch Angabe des Namens zugreifen können. Die Komplexität eines solchen Ansatzes würde dem Grundziel widersprechen, eine einfache Sprachdefinition zu erreichen. Alternativ wird die agile Entwicklung von DSLs empfohlen, bei der durch ein Refactoring einer der Obersprachen der Widerspruch beseitigt wird. Dadurch bleiben die Grammatiken gut lesbar und die Effekte der Grammatikvererbung verständlich.

4.1.1 Kontextbedingungen

Durch die Grammatikvererbung entstehen folgende zusätzliche Kontextbedingungen:

1. Produktionen dürfen nur von Produktionen des gleichen Typs überschrieben werden. Somit dürfen zum Beispiel Klassenproduktionen nur durch Klassenproduktionen und lexikalische Produktionen nur durch lexikalische Produktionen verschattet werden.

2. Beim Überschreiben einer Klassenproduktion entsteht in der abstrakten Syntax ein neuer Untertyp der entsprechenden Klasse. Die Oberproduktion darf nicht bereits Unterproduktionen durch Nichtterminalvererbung besitzen. Neue AST-Produktionen führen ebenfalls zu einer (impliziten) neuen Produktion mit demselben Produktionsrumpf wie in der Obergrammatik und somit zu einer neuen Unterklasse in der abstrakten Syntax.

3. Beim Überschreiben einer lexikalischen Produktion wird die lexikalische Erkennung ersetzt. Dabei muss sichergestellt sein, dass der dabei verwendete Java-Typ ein Untertyp aller lexikalischen Produktionen mit diesem Namen in allen Obergrammatiken ist. Ein Zusammenhang zwischen den regulären Ausdrücken der Produktionen muss nicht bestehen.

4. Beim Überschreiben von abstrakten und Schnittstellenproduktionen wird keine Unterklasse gebildet, sondern direkt die Klasse oder Schnittstelle aus der Obergrammatik übernommen. Daher darf auch keine neue Oberschnittstelle oder -klasse hinzugefügt werden. In Bezug auf die konkrete Syntax kann der Produktionsrumpf von der Obergrammatik übernommen werden oder auch überschrieben werden. Dieses hat dann nur Auswirkungen auf die konkrete Syntax.

5. Enumerationsproduktionen dürfen nicht überschrieben werden, weil für Enumerationen auf den meisten Plattformen keine Vererbung angeboten wird und so keine sinnvolle Vererbung realisiert werden kann. Aus demselben Grund ist auch die Verwendung von AST-Produktionen in Untergrammatiken für Enumerationsproduktionen nicht möglich.

6. Die Optionen werden zwischen den Grammatiken wie folgt vererbt:

   (a) Für Lexer- und Parser-Lookahead wird das Maximum aller Obergrammatiken verwendet, es sei denn, der Lookahead wird in der Grammatik überschrieben.

   (b) Für `header`-, `lexer`- und `parser`-Optionen wird die Konkatenation der entsprechenden Optionen in den Obergrammatiken verwendet, es sei denn, die Option wird in der Grammatik überschrieben.

   (c) Die Optionen `nows`, `noslcomment`, `nomlcomments`, `noanything`, `noident`, `nostring`, `noocharvocabulary` sind gesetzt, wenn sie in der Grammatik oder einer ihrer Obergrammatiken gesetzt sind.

   (d) Die Option `compilationunit` wird nicht vererbt. Dennoch stehen die erzeugten Produktionen, wenn sie in einer Obergrammatik verwendet wurden, über die normale Vererbung der Produktionen zur Verfügung.

   (e) Die `dotident`-Option wird nicht vererbt. (Die Produktion `IDENT` wird normal vererbt, so dass die Definition hinter `dotident` über die normale Vererbung von lexikalischen Produktionen zur Verfügung steht.)

   (f) Die `prettyprinter`-Option wird in der Grammatik definiert. Ist sie dort ausgelassen, wird sie von der ersten Obergrammatik übernommen, für die sie definiert ist.

4.1.2 Beispiel

Die Abbildungen 4.2 - 4.4 zeigen die Verwendung von Grammatikvererbung innerhalb des ShopSystems. Dabei wird die Sprachdefinition aus Abbildung 3.6 (Seite 47) als Obergrammatik verwendet und eine zusätzliche implementierende Klasse von `Order` hinzugefügt. `OrderDebitcard` stellt eine weitere Bezahlmöglichkeit für das Shopsystem dar. Die objektorientierte Notation innerhalb der Grammatik erlaubt somit, die Produktion `Order` unverändert zu lassen, obwohl für die konkrete Syntax jetzt hier die drei Varianten `OrderCash`, `OrderCreditcard` und `OrderDebitcard` geparst werden. Die Produktion `PremiumClient` wird durch eine gleichnamige Produktion überschrieben, die ein zusätzliches Attribut `birthdate` zur Sprache hinzufügt. Zusätzlich wird eine weitere lexikalische Produktion mit Namen `DATE` eingefügt, um Daten in spitzen Klammern zu parsen und im Datentyp `java.util.Date` zu speichern. Dabei wird der gesamte Text mit Ausnahme der Klammern genutzt, die durch die Ausrufezeichen nicht verwendet werden. In Zeile 10 der Abbildung 4.3 wird gezeigt, wie bei Umwandlungsfehlern eine Fehlermeldung ausgelöst werden kann. Das Klassendiagramm zeigt die abstrakte Syntax der entstehenden Sprachkombination. Dabei erzeugt der Parser der Sprache `ex.Shop10` Objektstrukturen, die zu dieser abstrakten Syntax konform sind. Aufgrund der Vererbung der konkreten Syntax entstehen dabei keine Instanzen von `Shop3.PremiumClient`, sondern nur welche vom Typ `ex.Shop10.PremiumClient`.

Dieses Beispiel zeigt insofern auch gut, dass kein direkter Zusammenhang zwischen den Sprachen besteht, die eine Grammatik und ihre Untergrammatik erzeugen, weil sowohl die lexikalische als auch die syntaktische Analyse grundlegend verändert werden können. Die abstrakte Syntax der Untergrammatik stellt jedoch eine Erweiterung der abstrakten Syntax der Obergrammatik dar, da keine Klassen entfernt werden können. Die modifizierte konkrete Syntax kann jedoch dafür sorgen, dass einige Klassen nicht mehr durch den Parser instanziert werden, sondern nur noch spezielle Unterklassen wie `PremiumClient` im dargestellten Beispiel.

MontiCore-Grammatik

```
1 grammar Shop3 {
2
3   // ...
4
5   PremiumClient extends Client =
6     "premiumclient" name:IDENT discount:NUMBER Address;
7 }
```

Abbildung 4.2: Auszug aus der Grammatik aus Abbildung 3.6 (Seite 47)

MontiCore-Grammatik

```
1 package ex;
2
3 grammar Shop10 extends Shop3 {
4
5   token DATE = "<"! NUMBER "." NUMBER "." NUMBER ">"! :
6     x-> java.util.Date : { java.util.Date d = null;
7       try {
8         d = java.text.DateFormat.getDateInstance().parse(x.getText());
9       } catch (java.text.ParseException e)
10        { throw new RecognitionException("Invalid date!"); }
11      return d;
12    };
13
14  OrderDebitcard implements Order =
15    "debitorder" clientName:IDENT cardNo:NUMBER;
16
17  PremiumClient =
18    "premiumclient" name:IDENT discount:NUMBER birthdate:DATE Address;
19 }
```

Abbildung 4.3: Erweiterung der Grammatik durch Grammatikvererbung

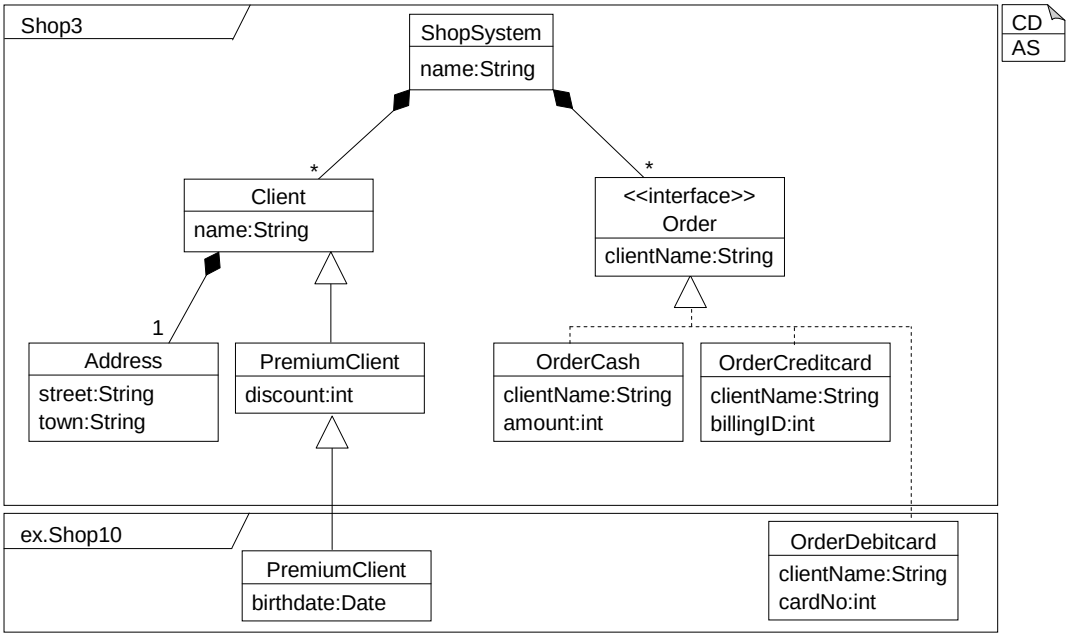


Abbildung 4.4: Abstrakte Syntax

4.1.3 Diskussion der Entwurfsentscheidungen

Bei der Ausgestaltung der Grammatikvererbung wurden drei Grundsätze beachtet:

- Bei der Übersetzung einer Grammatik sollen nur die Elemente der abstrakten Syntax neu erzeugt werden, für die auch explizit Produktionen durch den Entwickler angegeben wurden und nicht für geerbte Elemente. Dadurch lässt sich die aus einer Grammatik abgeleitete abstrakte Syntax so erzeugen, dass nur für die spezifizierten Produktionen Code generiert wird. Somit ist für den Grammatikentwickler nachvollziehbar, welche Klassen erzeugt werden.

- Überschriebene Produktionen sollen nur Elemente der abstrakten Syntax erzeugen, die Spezialisierungen der existierenden Elemente sind. Dadurch lassen sich Algorithmen für unveränderte Teile der Sprache, die auf der abstrakten Syntax programmiert wurden, unverändert nutzen.

- Eine Vererbung ohne die Angabe von neuen Elementen stellt eine (bis auf den Namen) unveränderte Grammatik dar. Dadurch lassen sich zu einer Sprache nachträglich modular Konzepte hinzufügen, indem eine Untersprache mit den Konzepten definiert wird. Die abstrakte und konkrete Syntax bleiben davon unberührt, sofern die Konzepte diese nicht beeinflussen.

Der erste Grundsatz hat zur Folge, dass die Menge der überschreibbaren Produktionen eingeschränkt werden muss (vgl. Punkt 2 auf Seite 66). Das Beispiel in Abbildung 4.5 zeigt die abstrakte Syntax, die aus der Grammatikvererbung einer Sprache Y, die von Sprache X erbt, entstanden ist. Dabei wird eine Klasse `A` überschrieben, die bereits eine Unterklasse `B` besitzt. Dieses führt zu der Situation, dass die Klasse `X.A` aufgrund der Grammatikvererbung einen neuen Subtyp `Y.A` in der abstrakten Syntax erhält. Dieser ist dann aber wiederum nicht mehr Oberklasse der Klasse `X.B`. Daher wäre dann ggf. eine Klasse `Y.B` einzuführen, damit auch die in der Sprache `X` definierte Unterklassenbeziehung für die davon abgeleitete Sprache `Y` gilt.

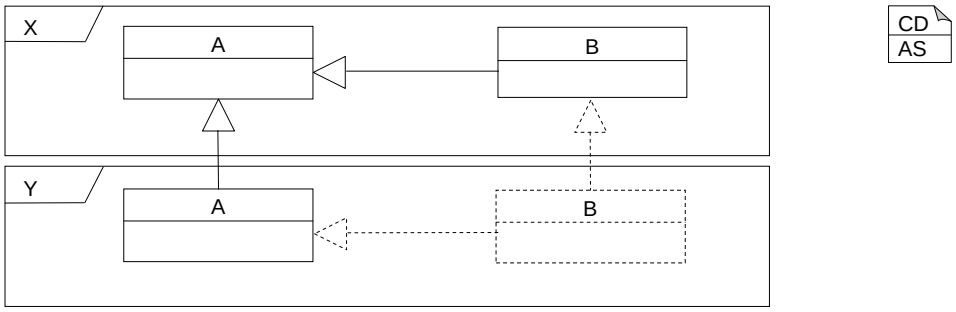


Abbildung 4.5: Problem bei der Grammatikvererbung von Klassenproduktion, die bereits Unterproduktionen haben

Dieses verletzt jedoch den ersten Grundsatz, so dass eine solche Grammatik als nicht wohlgeformt bezeichnet wird, weil zwei Probleme entstehen:

- Die Lösung verlangt Mehrfachvererbung in der abstrakten Syntax. Weil die MontiCore-AST-Klassen als Standard auf Java-Klassen abgebildet werden, wären diese Sprachen

nicht auf der Standardplattform realisierbar. Eine generelle Verwendung von Schnittstellen in der Standardabbildung würde diese Form der Vererbung ermöglichen. Diese Möglichkeit wurde jedoch verworfen, weil es keine großen Vorteile gegenüber der expliziten Einführung von Schnittstellen durch den Entwickler mittels Schnittstellenproduktionen bringt.

- Für einen Entwickler ist es nicht direkt nachvollziehbar, warum eine neue Klasse `Y.B` entsteht, obwohl die Grammatik `Y` keine Produktion `B` enthält.

Der zweite Grundsatz garantiert, dass sich existierende Algorithmen an die spezialisierten Sprachen anpassen lassen. Dieses ist insbesondere wichtig, weil die Implementierung von Algorithmen wie Codegenerierungen und Analysen einen entscheidenden Teil des Implementierungsaufwands einer DSL einnimmt. Im MontiCore-Framework gibt es unter anderem die folgenden Möglichkeiten zur Implementierung von Algorithmen, die eine nachträgliche Spezialisierung ermöglichen:

- Das Visitor-Muster ist innerhalb des MontiCore-Frameworks derart ausgestaltet, dass sich die Visitoren spezialisieren lassen und so an die spezialisierte Sprache angepasst werden können. Details hierzu finden sich in Abschnitt 9.3.10.
- MontiCore erlaubt die Spezifizierung von Attributen. Die Attributberechnungen sind dabei auf der abstrakten Syntax programmiert und können dadurch wie die Sprache spezialisiert werden.
- Die Programmiersprache Java verwendet dynamische Typbindung, was in Kombination mit Methoden in den AST-Klassen zu einer Spezialisierung der Algorithmen genutzt werden kann.

Zur Erfüllung des dritten Grundsatzes wurde insbesondere die Vererbung der Optionen so gestaltet, dass eine Ableitung ohne die Angabe zusätzlicher Optionen und Produktionen zur selben abstrakten und konkreten Syntax führt. Insbesondere erkennt der entstehende Parser dieselbe Sprache und erzeugt auf dieselbe Eingabe auch denselben AST. Dieses ermöglicht zum Beispiel, eine Sprache ohne eine Eclipse-Editorunterstützung zu beschreiben und so kommandozeilenbasierte Anwendungen zu entwickeln. Die Sprache kann dann nachträglich abgeleitet werden und die Untersprache mit dem entsprechenden Konzept zur Beschreibung der Eclipse-Editorunterstützung versehen werden. Da die abstrakte und konkrete Syntax dabei unverändert bleiben, können die Werkzeuge so transparent in Eclipse integriert werden.

Die Übersetzung einer MontiCore-Grammatik in ausführbaren Code erfordert den Quellcode aller Obergrammatiken. Bei der Implementierung von objektorientierten Compilern hingegen ist es möglich, dass für die Übersetzung einer Klasse nur die Schnittstelleninformationen der Oberklasse verwendet werden und nicht der komplette Quellcode. Die Compiler können die kompilierte Form einer Klasse und die darin eingebettete Schnittstelleninformation nutzen, um die syntaktische Korrektheit einer Unterklasse zu prüfen und diese zu übersetzen. Bei objektorientierter Programmierung lassen sich die Schnittstelleninformationen deutlich kompakter als die gesamte Implementierung formulieren, weil hierfür die Methodenrümpfe nicht notwendig sind. Die Schnittstelle beschränkt sich auf die Signatur der beteiligten Methoden und Attribute der Klassen. Bei Parsergeneratoren ist die Situation eine andere: Die Schnittstellenbeschreibung, die für eine Vererbung, wie sie in diesem Kapitel dargestellt wurde, notwendig ist, beschränkt sich nicht nur auf die vom

Entwickler formulierten Produktionsnamen. Vielmehr sind dazu First- und Follow-Mengen notwendig, die sich erst nach einer Analyse aus dem Rumpf der Produktion ergeben. Die Abbildung 4.6 zeigt einen Ausschnitt aus der Implementierung des Recursive-Descent-Parsers, wie Antlr sie erzeugt. Dabei wird klar, dass die First- und Follow-Mengen einer Produktion sich auch auf die Implementierung der anderen Produktionen auswirken. Wird also bei der Vererbung die Produktion `Y` überschrieben, müsste in diesem Beispiel auch der Parser für Produktion `X` angepasst werden, weil dort die Firstmenge der Produktion eingesetzt wird. Die notwendigen Änderungen sind somit nicht gut zu lokalisieren und im Extremfall über den kompletten Parser verteilt. Daher erzeugt die Parsergenerierung für jede Sprache einen eigenen Parser und stellt keine Verbindung bereits erzeugter Parser auf Quellcodeebene her.

MontiCore-Grammatik

```
1
2
3 X = "x" Y "x";
4
5
6 Y = "y";
7
8
```

Pseudocode

```
1 public X x() {
2
3   match("x");
4   if (LA(1)=="y") {
5     y();
6   }
7   match("x");
8 }
```

Abbildung 4.6: Recursive-Descent-Parsererzeugung

Somit lässt sich zusammenfassen, warum es innerhalb des MontiCore-Frameworks notwendig ist, die Obergrammatiken zur Quellcodeerzeugung im Klartext zur Verfügung zu haben:

- Die notwendige Schnittstellenbeschreibung einer MontiCore-Grammatik, um die Vererbung zu realisieren, ist nicht deutlich geringer als die eigentliche Grammatik selbst. Es ist nicht möglich, wie bei der objektorientierten Vererbung, größere Teile des Quellcodes auszulassen, weil diese zur Parsererzeugung notwendig sind.
- Die auf Antlr basierte Implementierung modularisiert die notwendigen Informationen nicht genug, um die Grammatikvererbung auf die Vererbung der Parserklasse zurückzuführen. Hierfür wäre die Implementierung eines eigenen Parsergenerators notwendig, die in dieser Arbeit vermieden wurde, weil so ein bereits qualitätsgesichertes Werkzeug eingesetzt werden konnte. Dadurch wurde der Aufwand der Implementierung verringert und gleichzeitig die Qualität der entstehenden Parser auf hohem Niveau garantiert.

4.2 Einbettung

DSLs werden typischerweise spezifisch für eine klar definierte und beschränkte Aufgabe entworfen. Daher ist es oft notwendig, mehrere Sprachen miteinander zu kombinieren, um alle Aspekte eines Softwaresystems beschreiben zu können. Ein Beispiel hierfür ist die OCL, die geeignet ist, Einschränkungen und logische Ausdrücke über Objekte zu formulieren. Die eigentlichen Objekte können jedoch nicht in der OCL selbst definiert werden, sondern hierfür ist eine andere Sprache notwendig. Für eine integrierte Entwicklung von Modellen ist es wünschenswert, die Einschränkungen räumlich nah zur Objektdefinition in derselben Datei aufführen zu können. Einbettung ist hier eine geeignete technische Möglichkeit, wenn die lexikalische Struktur der Hostsprache nicht ähnlich zur OCL ist und somit die Grammatikvererbung ungeeignet ist.

Das MontiCore-Grammatikformat erlaubt es, so genannte externe Nichtterminale durch das Schlüsselwort `external` zu definieren. An diesen Nichtterminalen können andere Sprachen verlinkt werden. Wird von einer Sprache in ihrer syntaktischen Analyse ein externes Nichtterminal erkannt, wird an dieser Stelle der Eingabe die lexikalische und syntaktische Analyse von der aktuellen auf die eingebettete Sprache gewechselt und das Parsen fortgesetzt. Am Ende der Einbettung, wenn der Erkennungsvorgang der eingebetteten Sprache beendet ist, wird wiederum der Erkennungsvorgang der äußeren Sprache mit ihrer lexikalischen und syntaktischen Analyse fortgesetzt. Der Prozess ist nicht auf zwei Stufen begrenzt, sondern auch die eingebettete Sprache kann ihrerseits andere Sprachen einbetten.

Externe Nichtterminale können als so genannte *bottom nonterminals* in Grammatikfragmenten [Läm01a] aufgefasst werden. Diese tauchen nur auf der rechten Seite einer Produktion auf und sind nicht selbst als Produktion definiert. MontiCore greift diese Terminologie auf und bezeichnet eine Grammatik mit externen Nichtterminalen als Fragment. Eine Sprache bezeichnet dann eine Grammatik ohne externe Nichtterminale oder eine Kombination verschiedener Fragmente, so dass für jedes externe Nichtterminal eine Sprache oder ein Fragment verlinkt wurde.

Die MontiCore-Infrastruktur kann Parser und abstrakte Syntaxklassen aus der Grammatik generieren, auch wenn dort externe Nichtterminale verwendet werden. Diese können zum Parsen verwendet werden, wobei ein nicht gebundenes externes Nichtterminal beim Erkennen zu einer Fehlermeldung führt. Die Parser können zur Entwicklungs- oder sogar zur Laufzeit kombiniert werden. Die resultierende Sprache kann als eine disjunkte Vereinigung aller Fragmente aufgefasst werden.

Die Auswahl eines eingebetteten Fragments kann nicht nur vom externen Nichtterminal bestimmt werden. Denn es ist möglich, auf Grundlage der bereits geparsten Eingabe einen Wert zu errechnen, der zur Auswahl zwischen verschiedenen Fragmenten verwendet werden kann.

4.2.1 Beispiel

Die Abbildungen 4.7 und 4.8 zeigen ein Beispiel für eine Einbettung. Dabei markiert das Schlüsselwort `external` die Nichtterminale `StatementCredit` und `StatementCash` als extern, welche durch geeignete Produktionen einer Grammatik einer eingebetteten Sprache ausgefüllt werden müssen. Das Nichtterminal `StatementCredit` spezifiziert zusätzlich, dass das eingebettete Fragment eine Instanz vom Typ `example.IStmt` erzeugen muss. Der Schrägstrich steht dabei wie auch in den AST-Regeln üblich für die Kennzeichnung eines Java-Typs. Durch diesen Mechanismus ist es möglich, von eingebetteten Grammatiken gewisse zur Verfügung zu stellende Methoden zu fordern und somit eine Interaktion zu rea-

lisieren. Dabei ist zu beachten, dass nicht direkt festgelegt wird, welche Sprache eingebettet wird, sondern nur deren Schnittstelle. Trifft der Parser beim Parsen eines Modells auf ein solches Nichtterminal, wird die zentrale Infrastruktur nach der Fortsetzung befragt. Diese Fortsetzung, also die Integration verschiedener Grammatiken, kann über eine Java-API oder über spezielle Sprachdateien erfolgen. Die genaue Auswahl der Sprache kann wie bei `StatementCash` auf eine einzelne Sprache beschränkt sein oder wie bei `StatementCredit` parametrisiert sein. Dabei wird nach dem Nichtterminal ein Attribut in spitzen Klammern angegeben. An dieser Stelle wird das Attribut `brand` der aktuellen AST-Klasse ausgewertet und der Parser ausgewählt, der unter dem aktuellen Wert des Attributs registriert ist.

MontiCore-Grammatik

```
1  grammar Shop11 {
2
3    ShopSystem =
4      name:IDENT Order*;
5
6    interface Order;
7
8    ast Order =
9      clientName:IDENT;
10
11   OrderCash implements Order =
12     "cashorder" clientName:IDENT amount:NUMBER StatementCash;
13
14   external StatementCash;
15
16   OrderCreditcard implements Order =
17     "creditorder" clientName:IDENT billingID:NUMBER make:IDENT StatementCredit<make>;
18
19   external StatementCredit /example.IStmt;
20 }
```

Abbildung 4.7: Erweiterung des ShopSystems durch Spracheinbettung

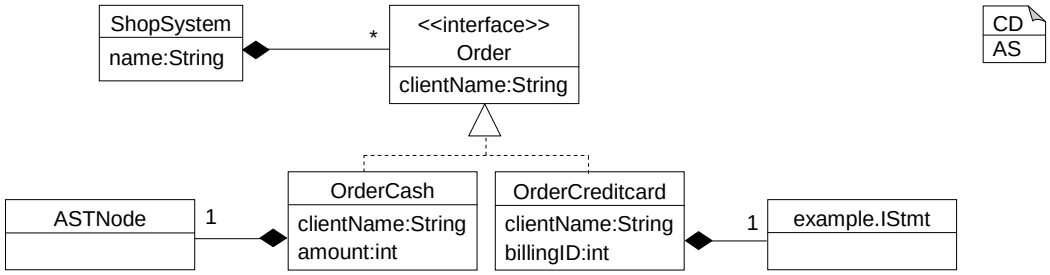


Abbildung 4.8: Aus Abbildung 4.7 abgeleitete abstrakte Syntax

Die Parametrisierung aufgrund von globalen Werten innerhalb eines Parsers wird in diesem Beispiel nicht gezeigt. Diese Werte können mit den in Abschnitt 3.5.4 dargestellten Befehlen gesetzt werden. Die Parametrisierung erfolgt, wie bei `StatementCredit` gezeigt, wobei jedoch zusätzlich das Schlüsselwort `global` vor dem Namen der globalen Datenstruktur angegeben werden muss. Dann wird der aktuelle Wert des globalen Stacks zur Auswahl des Parsers verwendet.

Die einzelnen Fragmente können über eine spezielle DSL, so genannte Sprachdateien, kombiniert werden. Die Definition der Sprachdateien befindet sich in Anhang D.2. Die Abbildung 4.9 zeigt eine solche Datei, die durch das Schlüsselwort `language` eingeleitet wird. Zunächst wird in Zeile 3 eine Root-Klasse definiert, die eine Sprache innerhalb des DSLTool-Frameworks definiert. In den Zeilen 5 - 13 wird die RootFactory beschrieben. Aus diesen Anweisungen werden Komponenten, die in das DSLTool-Framework, die in Kapitel 9 vorgestellte Referenzarchitektur für Generatoren, integriert werden können. Dort befindet sich eine detaillierte Beschreibung der Verantwortlichkeiten der einzelnen Systemteile.

Für die hier definierte Sprache wird durch den Stereotypen `<<start>>` das Fragment `Shop11` und dort durch einen Punkt getrennt die Produktion `ShopSystem` als Startproduktion festgelegt. Im Folgenden kann dieses Fragment als `shop` referenziert werden. In Zeile 9 wird definiert, dass die Produktion `Stmt` der Sprache `general.Statement` verwendet werden soll, wenn das externe Nichtterminal `StatementCash` angetroffen wird. In Zeile 11 wird festgelegt, dass für `StatementCredit` die Produktion `Statement` der Grammatik `brand.Visa` verwendet werden soll, wenn der Parameter mit dem Wert „visa“ belegt ist. Hingegen wird in Zeile 12 festgelegt, dass für den Wert „master“ die Produktion `Stmt` der Grammatik `brand.Master` verwendet werden soll.

MontiCore-Sprachdatei

```
1  language ShopSystem {
2
3    root ShopRoot<ShopSystem>;
4
5    rootfactory ShopRootFactory for ShopRoot<ShopSystem> {
6
7      Shop11.ShopSystem shop <<start>>;
8
9      general.Statement.Stmt s in shop.StatementCash;
10
11     brand.Visa.Statement s1 in shop.StatementCredit(visa);
12     brand.Master.Stmt s2 in shop.StatementCredit(master);
13   }
14 }
```

Abbildung 4.9: Kombination mittels einer Sprachdatei

4.2.2 Diskussion der Entwurfsentscheidungen

Bei der Realisierung der Spracheinbettung wurden die folgenden vier Grundsätze beachtet:

- Der Mechanismus erlaubt die Kombination beliebiger Sprachen miteinander. Insbesondere gibt es keinerlei Einschränkungen, dass die Sprachen eine ähnliche lexikalische oder syntaktische Struktur haben müssen oder nur ein gemeinsamer Satz an Schlüsselwörtern verwendet werden darf.

- Bei einer Kombination auftretende Mehrdeutigkeiten können zur Entwicklungszeit der Fragmente bezüglich jeder möglichen Einbettung analysiert werden.

- Die Kombination der eingebetteten Fragmente ist nach der Entwicklung möglich und nicht bereits zur Entwicklungszeit der Fragmente notwendig.

- Die Auswahl der eingebetteten Sprache kann durch den bereits erkannten Anteil der Datei erfolgen.

Der erste Grundsatz ist erfüllt, weil die Parsererzeugung auf einer etablierten LL(k)-Parsertechnologie basiert ist, die durch getrennte Lexer und Parser realisiert ist. Durch eine entsprechende Modifikation des Parsergenerators Antlr ist es aus der syntaktischen Analyse heraus möglich, den Lexer und Parser an dieser Stelle der Eingabe zu wechseln. Die etablierte Parsertechnologie garantiert einen gut verstandenen Erkennungsprozess, der dann auf die Eingabe stückweise angewendet wird. Somit sind alle Sprachen, die mit einem LL(k)-Parser erkennbar sind, auch in MontiCore eindeutig parsbar. Zusätzlich steht über die Prädikate im Extremfall ein vollständiges Backtracking zur Verfügung.

Der zweite Grundsatz lässt sich ebenfalls durch die Verwendung der bewährten Parsertechnologie sicherstellen. Der Verzicht auf eine komplexere scannerlose Parsertechnologie wie [Vis97] erlaubt die einfache Analyse der Grammatik zur Entwicklungszeit. Somit können Mehrdeutigkeiten mit herkömmlichen Algorithmen erkannt und die Entwickler auf diese hingewiesen werden. Dieses ist insbesondere bei externen Nichtterminalen wichtig, weil hier der Entwickler vor Mehrdeutigkeiten bezüglich jeder möglichen Sprache gewarnt wird. Prädikate können verwendet werden, um die zulässigen Sprachen so weit einzuschränken, dass keine Mehrdeutigkeit mehr auftreten kann. Das Prädikat beschreibt dann den syntaktischen Anfang, den alle an dieser Stelle eingebetteten Sprachen erfüllen müssen.

Der dritte Grundsatz wird dadurch erfüllt, dass es möglich ist, aus einem Fragment den Parser und die AST-Klassen zu erzeugen. Diese werden dann durch eine Sprachdatei oder eine Java-API konfiguriert. Der Lexer- und Parserwechsel ist so gestaltet, dass keine Modifikationen an den einmal erzeugten Klassen notwendig sind. Konkret bedeutet dieses, dass nach dem Entwurf der Sprache auch der kompilierte Parser ausreichend ist, um diesen mit anderen Grammatiken kombinieren zu können. Dieses erleichtert die Kombination verschiedener Sprachen, weil keine Kenntnis über interne Details notwendig ist. Somit können in sich abgeschlossene Sprachmodule entwickelt und qualitätsgesichert werden, die dann später in anderen Kontexten verwendet werden.

Der vierte Grundsatz wird dadurch erfüllt, dass der Mechanismus zur Erkennung, welche Sprache eingebettet ist, auf zwei Arten parametrisierbar ist:

- Es können syntaktische oder semantische Prädikate eingesetzt werden, um gezielt verschiedene externe Nichtterminale zu unterscheiden und so unterschiedliche eingebettete Sprachen auszuwählen.

- Es können Kontextinformationen genutzt werden, um über parametrisierbare externe Nichtterminale eine eingebettete Sprache auszuwählen.

Die erste Variante erlaubt über semantische Prädikate die Ausführung von beliebigem Java-Code. Dadurch kann auf den bereits geparsten Anteil zugegriffen werden. Die zweite Variante ermöglicht eine etwas restriktivere Auswahl, wobei Attribute der aktuelle AST-Klassen oder Peek-Werte von globalen Stacks verwendet werden können, um zwischen vordefinierten Alternativen wählen zu können.

Die derzeitige Ausgestaltung der Sprachdateien erlaubt nicht die Verwendung von bereits zusammengestellten Sprachen für eine Einbettung. Die in Kapitel 5 dargestellte Formalisierung und die Java-API, die die Realisierung übernimmt, erlauben dieses jedoch bereits.

4.3 Konzepte

Das MontiCore-Grammatikformat ist eine spezielle DSL zur Beschreibung der abstrakten und konkreten Syntax von Sprachen. Wird sie innerhalb einer modellgetriebenen Entwicklung eingesetzt, bildet sie die Grundlage für eine Vielzahl von Arten von Artefakten. Dabei lässt sich jedoch beobachten, dass einzelne Codegenerierungen aus der Sprachdefinition heraus durch zusätzliche Informationen parametrisiert werden müssen. Diese sollten direkt in der Sprachdefinition notiert werden können, um so eine kompakte zentrale Referenz zu erhalten. Zusätzlich können diese Konzepte dann mit der Sprache vererbt werden oder in einer Sprachkombination genutzt werden. Somit können sie über dieselben Wiederverwendungsmechanismen genutzt werden und zusammen mit den Sprachen in Bibliotheken abgelegt werden.

Durch die Konzepte kann das Grammatikformat als universelle Beschreibung von abstrakter und konkreter Syntax für spezielle Domänen derart angepasst werden, dass sich zusätzliche Komponenten aus der Sprachdefinition erzeugen lassen. Das MontiCore-Grammatikformat baut damit auf Erfahrungen mit der UML auf, die die Profilbildung einsetzt, um die Sprache an domänenspezifische Gegebenheiten anzupassen. Wie in [FR07] für die UML diskutiert, wurde die Definition der abstrakten und konkreten Syntax als kleinster gemeinsamer Kern einer Sprachdefinition identifiziert und weiterführende Elemente können als so genannte Konzepte genutzt werden.

Die Konzepte, die innerhalb des MontiCore-Frameworks entwickelt wurden und zur allgemeinen Verwendung zur Verfügung stehen, werden in Kapitel 6 beschrieben. Zusätzlich wird dort erklärt, wie die Konzepte in die Typanalyse des Grammatikformats integriert werden, damit auch hier die Kontextbedingungen geprüft werden können. Dazu kann die schrittweise aufgebaute Symboltabelle der Grammatik genutzt werden, um zum jeweiligen Stadium die entsprechenden Elemente in den Konzepten abzufragen. Zusätzlich wird erläutert, wie die Ableitung der abstrakten und konkreten Syntax durch die Konzepte beeinflusst werden kann.

4.3.1 Beispiel

Die Beschreibung von Assoziationen gehört zur Definition der abstrakten Syntax einer DSL und ist daher Bestandteil des Kerns des Grammatikformats, wie es in Kapitel 3 beschrieben wird. Die Abbildung 4.10 zeigt einen Ausschnitt des Verkaufssystems, in dem in den Zeilen 13 und 14 eine Assoziation `ClientOrder` definiert wird. Es existieren zwei navigierbare Assoziationsenden: Einerseits ist von einem `Client` aus über das Navigationsende `order` eine beliebige Anzahl an `Orders` zu erreichen. Der Name des Navigationsende ist hier nicht explizit definiert worden, sondern ergibt sich aus dem Namen der referenzierten Klasse. Andererseits ist von einer `Order` aus genau ein `Client` erreichbar.

Die MontiCore-Generierung umfasst neben den AST-Klassen, die die Navigationsenden beinhalten, und den Assoziationen selbst, auch einen Workflow, der die Kardinalitäten der Assoziationen überprüft.

Davon unabhängig müssen Algorithmen definiert werden, die nach der Erzeugung des ASTs die Links zwischen den Objekten etablieren, die zu den Assoziationen passen. Diese Etablierung gehört nicht direkt zur Beschreibung der konkreten und abstrakten Syntax einer Sprache und ist daher nicht durch das MontiCore-Grammatikformat spezifizierbar. Diese Algorithmen sind manuell programmierbar. Alternativ existieren Erweiterungen des Grammatikformats in Form von Konzepten, die es erlauben, die Etablierung von Links deklarativ zu beschreiben.

MontiCore-Grammatik

```
1  grammar Shop12 {
2
3    Client =
4      "client" name:IDENT Address;
5
6    interface Order;
7
8    ast Order =
9      clientName:IDENT;
10
11   // ...
12
13   association ClientOrder
14      Client 1 <-> * Order.orderingClient;
15
16   concept sreference {
17     ClientOrder: Order.clientName = Client.name;
18   }
19 }
```

Abbildung 4.10: Verwendung des Konzepts `sreference`

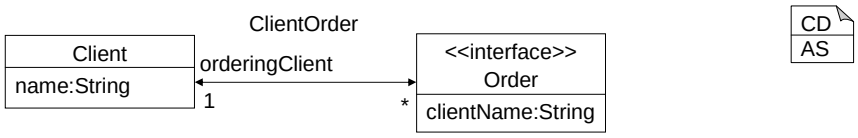


Abbildung 4.11: Auszug aus der aus Abbildung 4.10 abgeleiteten abstrakten Syntax

Die Abbildung 4.10 zeigt die Etablierung der Verbindungen zwischen Objekten für eine Assoziation mittels des Konzepts `sreference`. Dabei wird ein Link zwischen zwei Objekten etabliert, wenn die beiden Attribute `name` von `Client` und `clientName` von `Order` in ihrem Wert übereinstimmen. Aus der Spezifikation wird ein Workflow generiert, der nach dem Parsen ausgeführt werden kann und dann die Links erzeugt.

Für die Konzepte sollten vom Entwickler Kontextbedingungen implementiert werden, damit der DSL-Entwickler auf seine Eingabe bezogene Fehlermeldungen erhält. Dafür bietet das MontiCore-Framework die Symboltabelle der Grammatik, die für das im Beispiel dargestellte Konzept `sreference` die folgenden Bedingungen prüft:

- Existiert eine Assoziation mit Namen `ClientOrder`?
- Sind `Client` und `Order` Klassen der abstrakten Syntax und Teil der Assoziation `ClientOrder`?
- Ist `clientName` ein Attribut von `Order` und `name` ein Attribut von `Client`?

4.3.2 Entwicklung von Konzepten

Konzepte stellen einen Erweiterungspunkt des MontiCore-Frameworks dar. Für den Konzeptentwickler sind zwei Implementierungsaufgaben zu erledigen: Erstens die Überprüfung von Kontextbedingungen und zweitens die Implementierung der Codegenerierung. Die Realisierung der Codegenerierung kann dabei im Wesentlichen auf zwei Arten erfolgen:

- Realisierung als eingebettete DSL, das heißt die Rückführung des Konzepts auf Elemente des Grammatikformats. Dabei werden die entsprechenden Produktionen erzeugt und dann die Symboltabelle automatisiert erzeugt.

- Realisierung als eigenständige Generierung, die zum Beispiel die Symboltabelle als Unterstützung nutzt.

Unabhängig von der Codegenerierung sollten die Kontextbedingungen immer auf der ursprünglichen Eingabe des DSL-Entwicklers ausgegeben werden. Nur so kann dieser seine Fehler korrigieren, ohne Details der Codegenerierung zu kennen. Für den Konzept-Entwickler ist es wichtig, die dabei auftretenden Kontextbedingungen leicht programmieren zu können. Insbesondere sollte es nicht notwendig sein, die Prinzipien der Grammatikvererbung zu verstehen, sondern die Menge der Produktionen einer Grammatik wird durch die Symboltabelle der Grammatik bestimmt. Die Symboltabelle und die einzelnen Phasen beim Aufbau werden in Abschnitt 6.3.1 erklärt. Im Wesentlichen lassen sich die folgenden Aktionen durchführen:

- Hinzufügen von Produktionen und Assoziationen zur Grammatik zur Realisierung einer eingebetteten DSL.

- Hinzufügen von Produktionen oder Assoziationen zur Symboltabelle.

- Überprüfen von Produktionen oder Assoziationen in der Symboltabelle.

- Hinzufügen von Attributen zur Symboltabelle.

- Überprüfen von Attributen mittels der Symboltabelle.

4.3.3 Diskussion der Entwurfsentscheidungen

Die Konzepte sind innerhalb des MontiCore-Grammatikformats über Einbettung realisiert. Dadurch ergeben sich keine Einschränkungen bei der Realisierung der Konzepte. Diese Freiheit birgt gleichzeitig die Gefahr, dass die Konzepte nicht die Notationskonventionen des Grammatikformats respektieren. Hierzu ist eine kritische Beurteilung seitens des Entwicklers notwendig.

Die Symboltabelle wird, wie in Abschnitt 6.3.1 beschrieben, schrittweise aufgebaut. Dabei wird zwischen dem unmodifizierten AST, der beispielsweise für das Prettyprinting verwendet wird, und dem modifizierten AST unterschieden, der auch die von Konzepten und Optionen erzeugten Produktionen enthält. Die Symboltabelle enthält entsprechende Callback-Methoden zum Aufbau der Elemente und zur Überprüfung der Kontextbedingungen. Durch die strenge Reglementierung ist sichergestellt, dass verschiedene Konzepte unabhängig voneinander entwickelt werden können, auch wenn sie sich auf dieselben Elemente einer Grammatik beziehen.

Die Implementierung einer eingebetteten DSL kann noch komfortabler ausgestaltet werden. Dabei ergeben sich derzeit Schwierigkeiten, wenn die neu hinzugefügten Produktionen oder Assoziationen die Kontextbedingungen der Grammatik verletzen. Dann werden allgemeine Fehlermeldungen erzeugt, die sich nicht auf die Eingabe des DSL-Entwicklers beziehen. Die Realisierung von *src infos* wie in [Tra08] dargestellt könnte eine geeignete Lösung sein, weil so die Elemente immer zum ursprünglichen Erzeuger zurückverfolgt werden können und dieser eine entsprechende Fehlermeldung generieren kann.

4.4 Zusammenfassung

In diesem Kapitel wurden Modularitätskonzepte der MontiCore-Grammatik dargestellt, die eine strukturierte Wiederverwendung von Sprachen ermöglichen:

- Sprachvererbung, die bisher nur für die konkrete Syntax publiziert wurde, wurde so erweitert, dass sie sich auch auf die abstrakte Syntax einer Sprache auswirkt.
- Mit der Spracheinbettung wurde ein weiterer Mechanismus vorgestellt, der eine Komposition von Sprachen aus bestehenden bereits kompilierten Sprachkomponenten erlaubt. Dabei wurde insbesondere ein Mechanismus geschaffen, der gut fundierte Parsertechnologien verwendet, aber trotzdem eine Kompositionalität auch auf lexikalischer Ebene erreicht.
- Das Grammatikformat ist so entwickelt worden, dass eine Sprachdefinition modular ergänzt werden kann. Die Erweiterungen können in die Typprüfung des Grammatikformats mit einbezogen werden.

Kapitel 5

Formalisierung

In diesem Kapitel werden das MontiCore-Grammatikformat und die Sprachdateien formalisiert, um deren Bedeutung unabhängig von der Implementierung im MontiCore-Framework eindeutig festzulegen. Dazu wird eine auf die Kernelemente reduzierte abstrakte Syntax als algebraischer Datentyp dargestellt. Die Beschränkung auf die Kernelemente erlaubt eine kompaktere Darstellung der semantischen Abbildungen, als dieses mit dem vollständigen Sprachumfang möglich wäre, ohne dabei wesentliche Konstrukte auszulassen.

Zunächst werden die notwendigen Kontextbedingungen formalisiert, die für wohlgeformte Grammatiken und Sprachdateien erfüllt sein müssen. Somit wird die Menge der Modelle auf die Grammatiken und Sprachdateien eingeschränkt, für die sich eine Semantik definieren lässt.

Die Ableitung der konkreten und abstrakten Syntax einer DSL aus den Grammatiken und Sprachdateien wird dabei mit Hilfe zweier denotationeller Semantiken erklärt:

- Als semantische Domäne für die abstrakte Syntax einer Sprache wird eine Variante der UML/P-Klassendiagramme verwendet, die ihrerseits in [CGR08] im Zuge der semantischen Fundierung der UML erklärt wird. Durch die semantische Abbildung lässt sich somit erkennen, welche Klassenstrukturen aus einer Grammatik erzeugt werden. Eine Sprachdatei erzeugt keine Datenstrukturen, sondern kombiniert nur existierende, so dass hier keine Abbildung notwendig ist.

- Als semantische Domäne für die konkrete Syntax wird eine hier entwickelte Variante der *Extended-Backus-Naur-Form* (EBNF) verwendet, die als X-Grammatik bezeichnet wird, und den zweistufigen Erkennungsprozess mit Lexer und Parser erkennen lässt. Mittels einer Ableitungsrelation wird definiert, wie die validen Sätze einer Sprache erzeugt werden. Die konkrete Syntax einer DSL wird somit durch die Überführung einer MontiCore-Grammatik oder einer Sprachdatei in eine X-Grammatik erläutert.

Das Kapitel ist wie folgt gegliedert: In Abschnitt 5.1 werden Grundlagen erklärt, die eine kompakte Darstellung der Zusammenhänge innerhalb der folgenden Abschnitte erlaubt. In Abschnitt 5.2 wird die vereinfachte abstrakte Syntax des Grammatikformats mit einer mengentheoretischen Notation dargestellt. In Abschnitt 5.3 werden die wichtigsten Kontextbedingungen erläutert, die für wohlgeformte Grammatiken und Sprachdateien erfüllt sein müssen, damit für sie eine sinnvolle Semantik abgeleitet werden kann. Dabei werden Hilfsdatenstrukturen und -abbildungen verwendet, die in Abschnitt 5.4 erklärt werden. Die vereinfachte abstrakte Syntax der UML/P-Klassendiagramme wird in Abschnitt 5.5 dargestellt. In Abschnitt 5.6 wird die Ableitung der abstrakten Syntax einer Sprache aus einer Grammatik beschrieben, indem ein entsprechendes UML/P-Klassendiagramm angegeben

wird. In Abschnitt 5.7 wird erklärt, wie dieses Klassendiagramm auf verschiedene Weisen in Quellcode umgesetzt werden kann. Die Art der Umsetzung kann dabei die Kontextbedingungen beeinflussen. In Abschnitt 5.8 wird die abstrakte Syntax von Sprachdateien angegeben. In Abschnitt 5.9 werden die Kontextbedingungen der Sprachdateien kompakt formuliert, wobei weitere Hilfsdatenstrukturen und -abbildungen aus Abschnitt 5.10 verwendet werden. In Abschnitt 5.11 werden X-Grammatiken als semantische Domäne für die konkrete Syntax definiert. Die semantische Abbildung von Grammatiken und Sprachdateien in diese Domäne wird in Abschnitt 5.12 erläutert. Der Abschnitt 5.13 fasst dieses Kapitel kurz zusammen. Der Anhang E ergänzt die Darstellungen, indem die in diesem Kapitel definierten Funktionen, Mengen, Prädikate, Symbole und Datentypen aufgelistet werden.

5.1 Grundlagen

Die folgenden Ausführungen orientieren sich an [EMGR+01] und vereinfachen die Semantikdefinition des Grammatikformats, indem grundlegende Notationen und Datentypen erklärt werden.

Definition 5.1.1 (Wörter und Wortmengen)**.** Sei $\mathcal{A}$ eine beliebige Menge. Dann wird definiert:

- Das ausgezeichnete Element $\epsilon \notin \mathcal{A}$ heißt leeres Wort (über $\mathcal{A}$).
- Für $n \in \mathbb{N} \setminus \{0\}$ und $a_1, \ldots, a_n \in \mathcal{A}$ beliebig, heißt $a_1 a_2 \ldots a_n$ nichtleeres Wort über $\mathcal{A}$.
- Die Menge aller Wörter über $\mathcal{A}$ wird als $\mathcal{A}^*$ bezeichnet.
- Die Menge aller nichtleeren Wörter über $\mathcal{A}$ wird als $\mathcal{A}^+$ bezeichnet.

Definition 5.1.2 (Stringkonkatenation)**.** Die Stringkonkatenation $\cdot$ wird zur einfacheren Definition der Sprache einer Grammatik wie folgt erweitert. Das Symbol $\wp$ bezeichnet dabei die Potenzmenge.

$$\begin{array}{ll} \cdot : \mathcal{A}^* \times \mathcal{A}^* \to \mathcal{A}^* & w_1 \cdot w_2 =_{\text{def}} w_1 w_2 \\ \cdot : \mathcal{A}^* \times \wp(\mathcal{A}^*) \to \wp(\mathcal{A}^*) & w_1 \cdot S =_{\text{def}} \{w_1 w_2 \mid w_2 \in S\} \end{array}$$

Dabei gilt insbesondere $\forall w \in \mathcal{A}^* : w \cdot \emptyset = \emptyset$.

Definition 5.1.3 (Algebraische Signatur)**.** Sei S eine beliebige Menge, deren Elemente als Sorten bezeichnet werden. Sei OP die Liste der Operationssymbole. Jedes Operationssymbol $f \in OP$ besitzt eine Operationsdeklaration $f : s_1 \times \ldots \times s_n \to s$, wobei $n \in \mathbb{N}$ und $s, s_1, \ldots, s_n \in S$. Dann heißt das Paar (S, OP) algebraische Signatur.

Definition 5.1.4 (Signatur der Grammatikausdrücke Σ_G)**.** Im Folgenden bezeichnet die algebraische Signatur $\Sigma_G = (\{exp\}, \{\text{concat}, \text{alt}, \text{kleene}\})$ die Signatur der Ausdrücke in einer Grammatik, wobei die folgenden Operationsdeklarationen gelten:

$$\begin{array}{ll} \text{concat:} & exp \times exp \to exp \\ \text{alt:} & exp \times exp \to exp \\ \text{kleene:} & exp \to exp \end{array}$$

Definition 5.1.5 (Einsortige algebraische Signatur mit Variablen)**.** Sei (S, OP) eine algebraische Signatur, wobei S nur aus einer Sorte besteht. Sei X eine Menge, deren Elemente als Variablen bezeichnet werden, wobei gilt $X \cap OP = \emptyset$, so dass sich die Variablen eindeutig von den Konstanten (nullstellige Operationen) in OP unterscheiden. Dann heißt das Tripel (S, OP, X) einsortige algebraische Signatur mit Variablen.

Definition 5.1.6 (Allgemeine Terme)**.** Sei $\Sigma = (S, OP, X)$ eine einsortige algebraische Signatur mit Variablen. Die Menge der Terme über X, $\mathcal{T}_\Sigma(X)$, bezeichnet die kleinste Teilmenge von $(S \cup OP_B \cup X \cup \{„(“, „)“, „,“\})*$, für die die folgenden Bedingungen gelten:

(1) Variablen sind Terme.
$\forall x \in X : x \in \mathcal{T}_\Sigma(X)$

(2) Konstanten sind Terme.
$\forall c :\to s \in OP : c \in \mathcal{T}_\Sigma(X)$

(3) Die Anwendung von Operationen auf Terme erzeugt weitere Terme.
$\forall f : (s_1 \times ... \times s_n) \to s \in OP,\ t_1, ..., t_n \in \mathcal{T}_\Sigma(X) : f(t_1, ..., t_n) \in \mathcal{T}_\Sigma(X)$

Die Definition 5.1.6 beschreibt alle Sätze, die sich aus der Signatur und den Variablen durch die generische Operationsanwendung bilden lassen. Im Folgenden wird aus Gründen der Lesbarkeit jedoch oft eine Infix-Schreibweise mit spezifischen Operationssymbolen verwendet, die sich aber formal auf diese generische Schreibweise zurückführen ließe. Zum Beispiel wird für die Alternative die übliche Schreibweise $a \mid b$ anstatt der hier formal definierten Notation $\text{alt}(a, b)$ verwendet.
Im Folgenden werden algebraische Datentypen verwendet, wobei auf eine detaillierte Definition über die Signatur hinaus verzichtet wird, weil die Operationen nicht von in der Literatur wie [EMGR+01] gebräuchlichen Definitionen abweichen.

Definition 5.1.7 (*Stack* $\langle$*Value*$\rangle$)**.** Ein Stack ist ein First-In-First-Out-Stapelspeicher mit Werten aus der Menge *Value*.

empty	$\to$ *Stack* $\langle$*Value*$\rangle$	Typkonstruktor.
peek	*Stack* $\langle$*Value*$\rangle \to$ *Value*	Liefert das oberste Datenelement.
push	*Stack* $\langle$*Value*$\rangle \times$ *Value* $\to$ *Stack* $\langle$*Value*$\rangle$	Fügt ein Datenelement hinzu.
tail	*Stack* $\langle$*Value*$\rangle \to$ *Stack* $\langle$*Value*$\rangle$	Liefert den Stack ohne das oberste Datenelement.

Definition 5.1.8 (*Map* $\langle$*ID*, *Value*$\rangle$)**.** Eine Map speichert Wertpaare zwischen *ID* und *Value*.

empty	$\to$ *Map* $\langle$*ID*, *Value*$\rangle$	Typkonstruktor.
get	*Map* $\langle$*ID*, *Value*$\rangle \times$ *ID* $\to$ *Value*	Liefert Wert zum Schlüssel.
set	*Map* $\langle$*ID*, *Value*$\rangle \times$ *ID* $\times$ *Value* $\to$ *Map* $\langle$*ID*, *Value*$\rangle$	Setzt Wert für Schlüssel.

Definition 5.1.9 (*List* $\langle$*Value*$\rangle$)**.** Eine Liste beinhaltet Werte vom Typ *Value*.

new	$\to$ *List* $\langle$*Value*$\rangle$	Typkonstruktor.
append	*List* $\langle$*Value*$\rangle \times$ *Value* $\to$ *List* $\langle$*Value*$\rangle$	Fügt einen Wert an die Liste an.
head	*List* $\langle$*Value*$\rangle \to$ *Value*	Liefert den ersten Wert.
tail	*List* $\langle$*Value*$\rangle \to$ *List* $\langle$*Value*$\rangle$	Liefert die Liste ohne den ersten Wert.

Bei der Verwendung der Datentypen wird oftmals eine objektorientierte Schreibweise verwendet, d. h. das erste Element einer Operation wird vor den Operatornamen, getrennt durch einem Punkt, gestellt. Somit sind zum Beispiel append(l, a) und $l.\,\text{append}(a)$ alternative Schreibweisen für das Anfügen eines Elements a an eine Liste l.

Definition 5.1.10 (Funktionen)**.** Für eine partielle Funktion $f : A \to B$ bezeichnen $\text{dom}(f) = A$ die Quelle (Domain) und $\text{codom}(f) = B$ die Zielmenge (Codomain). Die Bildmenge einer partiellen Funktion f wird als $\text{range}(f) = \{f(a) \mid a \in A\}$ bezeichnet. Der Definitionsbereich $\text{def}(f) = \{a \mid f(a) = b, b \in B\}$ bezeichnet die Teilmenge von A für die f definiert ist, so dass $f : \text{def}(f) \to B$ eine (totale) Funktion ist.

5.2 Abstrakte Syntax des Grammatikformats

In diesem Abschnitt wird die abstrakte Syntax des MontiCore-Grammatikformats als algebraischer Datentyp beschrieben. Die Grundlagendefinitionen aus Abbildung 5.1 beinhalten einige elementare Definitionen, die in sowohl in der MontiCore-Grammatik als auch in den Sprachdateien (vgl. Abschnitt 5.8) verwendet werden. Die Abbildung 5.2 zeigt die abstrakte Syntax der verwendeten Aktionssprache innerhalb einer MontiCore-Grammatik. Die Abbildung 5.3 zeigt die abstrakte Syntax des Grammatikformats, die die Grundlage für die Semantikdefinition bildet. Falls die Menge nicht direkt in einer Abbildung definiert ist, verweist der Index auf die Nummer der entsprechenden Abbildung.

Zur Definition der abstrakten Syntax des MontiCore-Grammatikformats wurden zunächst Vereinfachungen an der Sprachdefinition durchgeführt. Die hier dargestellte Variante des Grammatikformats ist die Essenz der Sprache, die eine kompakte Semantikdefinition ermöglicht. In Anhang D.1 wird das MontiCore-Grammatikformat und die verwendete Essenz als MontiCore-Grammatik dargestellt. Dabei wird in Abschnitt D.1.5 detailliert dargestellt, welche Elemente ausgelassen wurden. Die wichtigsten Änderungen sind dabei die folgenden:

- Auslassen von „syntaktischem Zucker“ des MontiCore-Grammatikformats, der sich diese nicht auf die dargestellten Aspekte der Formalisierung auswirkt oder durch die Verwendung anderer Konkstrukte erreicht werden kann:
  - Keine `astimplements` und `astextends` in Parserproduktionen, weil hierfür AST-Regeln verwendet werden können
  - Keine Verwendung von Iterationen wie `*` und `+` bei Nichtterminalen, weil die Nichterminale in Klammern verwendet werden können
  - Auslassen der Iterationen `?` und `+`, weil diese auf `*` und die Verwendung von leeren Alternativen zurückgeführt werden können

Direction	=	$\{\leftarrow, —, \leftrightarrow, \rightarrow\}$
QName	=	List $\langle$*Name*$\rangle$
JType	=	$\ll$ Java-Klasse oder Schnittstelle $\gg$
Card	=	*Min* $\times$ *Max*
Min	=	$\mathbb{N}$
Max	=	$\mathbb{N} \cup \{*\}$
Name	=	$\ll$ Menge von Bezeichnern $\gg$
String	=	$\ll$ Zeichenkette $\gg$

Abbildung 5.1: Gemeinsame Mengen der verschiedenen Sprachen

$Action$	$=$	$Push \cup Pop \cup Constr$
$Push$	$=$	$StackName \times VarName$
Pop	$=$	$StackName$
$Constr$	$=$	$\text{List}\,\langle Stmt \rangle$
$Stmt$	$=$	$Assign \cup Append$
$Assign$	$=$	$AttName \times VarName$
$Append$	$=$	$AttName \times VarName$
$AttName$, $StackName$, $VarName$	$=$	$Name_{5.1}$

Abbildung 5.2: Aktionssprache Astscript

$Grammar$	$=$	$QName_{5.1} \times Name_{5.1} \times Alphabet \times \text{List}\,\langle Grammar \rangle \times \wp(Prod) \times \wp(ASTRule) \times \wp(Association)$
$Prod$	$=$	$LexProd \cup EnumProd \cup ExternalProd \cup ParserProd$
$ParserProd$	$=$	$InterfaceProd \cup AbstractProd \cup ClassProd$
$LexProd$	$=$	$Name_{5.1} \times RegularExpression \times JType_{5.1} \times MappingJava$
$EnumProd$	$=$	$Name_{5.1} \times \wp(Constant)$
$ExternalProd$	$=$	$Name_{5.1} \times JType_{5.1}$
$InterfaceProd$	$=$	$IType \times \wp(Inheritance) \times \text{List}\,\langle Parameter \rangle$
$AbstractProd$	$=$	$CType \times \wp(Inheritance) \times \wp(Implementation) \times \text{List}\,\langle Parameter \rangle$
$ClassProd$	$=$	$Name_{5.1} \times CType \times CType \times \wp(Inheritance) \times \wp(Implementation) \times \text{List}\,\langle Parameter \rangle \times Body$
$Parameter$	$=$	$Name_{5.1} \times ProdElementType \times Name_{5.1} \times Card_{5.1}$
$ProdElementType$	$=$	$\{\text{AST}, \text{Ignore}, \text{VarAssign}\}$
$Body$	$=$	$\mathcal{T}_{\Sigma_G}(ProdElement \cup Action_{5.2})$
$ProdElement$	$=$	$Name_{5.1} \times ProdElementType \times ProdElementRef$
$ProdElementRef$	$=$	$Constant \cup NonTerminal \cup Terminal$
$Constant$	$=$	$Name_{5.1} \times String_{5.1}$
$NonTerminal$	$=$	$Name_{5.1} \times ParserParameter^{\text{opt}} \times \text{List}\,\langle VariableName \rangle$
$Terminal$	$=$	$String_{5.1}$
$ParserParameter$	$=$	$ParserMode \times Name_{5.1}$
$ParserMode$	$=$	$\{\text{global}, \text{ast}\}$
$ASTRule$	$=$	$Type \times \wp(CType) \times \wp(IType) \times \wp(AttributeInAST)$
$AttributeInAST$	$=$	$Name_{5.1} \times AttType \times Card_{5.1}$
$AttType$	$=$	$Type \cup JType_{5.1}$
$Association$	$=$	$Name_{5.1} \times AssociationEnd \times Direction_{5.1} \times AssociationEnd$
$AssociationEnd$	$=$	$Type \times Name_{5.1} \times Card_{5.1}$
$Type$	$=$	$CType \cup IType$
$CType$, $IType$, $Inheritance$, $Implementation$,		
$VariableName$	$=$	$Name_{5.1}$
$RegularExpression$	$=$	$\ll$ Regulärer Ausdruck $\gg$
$MappingJava$	$=$	$\ll$ Abbildung eines Strings auf einen Java-Typen $\gg$
$Alphabet$	$=$	$\ll$ Alphabet $\gg$

Abbildung 5.3: Vereinfachte abstrakte Syntax des MontiCore-Grammatikformats

  - Auslassen der Iteration `&`: Die Schlüsselwörter können einzeln aufgeführt werden
  - Auslassen von Assoziationsblöcken: Alternativ können Assoziationen verwendet werden
  - Auslassen von Konstantengruppen: Alternativ können Konstanten verwendet werden

- Vereinfachte Betrachtung bestimmter Aspekte der Sprache:
  - Ausschließliche Verwendung einer eingeschränkten Aktionssprache anstatt beliebiger Java-Anweisungen
  - Vereinfachtes Lexerformat

- Ausgelassene Aspekte der Formalisierung, die sehr implementierungsabhängig sind und daher nicht zur Semantik hinzugefügt wurden:
  - Grammatik-Optionen
  - Konzepte (weil deren Bedeutung individuell definiert werden muss)
  - Methodendefinitionen in den AST-Regeln.
  - Syntaktische und semantische Prädikate
  - Init-Aktionen
  - Externe Implementierung von AST-Klassen

Die vereinfachte Sprachdefinition wird kanonisch mittels der im Folgenden beschriebenen Regeln, die sich an [CGR08] orientieren, auf eine einfache mengentheoretische Beschreibung abgebildet. Weil die MontiCore-Grammatiksprache im Bootstrapping-Verfahren entwickelt wird, kann die Grammatiksprache dabei wie jede andere Sprache behandelt werden.

- Produktionen werden auf Mengen abgebildet.

- Die Elemente des Produktionskörpers bilden Tupel, die die Struktur der Elemente festlegen.
  Grammatik `X = A B C`
  Mengen $X = A \times B \times C$

- Der optionale Name von Nichtterminalen wird ignoriert.
  Grammatik `X = a:A b:B C`
  Mengen $X = A \times B \times C$

- Schnittstellenproduktionen und Alternativen werden auf die (disjunkte) Mengenvereinigung zurückgeführt.
  Grammatik `interface X; A implements X; B implements X;`
  oder
  Grammatik `X = A | B;`
  Mengen $X = A \cup B$

- Optionale Elemente `X?` werden zu einer Menge, die ein spezifisches Element $\epsilon \notin X$ enthält, das die Nichtexistenz repräsentiert. X^{opt} stellt dabei eine Kurzform von $X \cup \{\epsilon\}$ dar.

- Wiederholungen durch einen kleenschen Stern in der Grammatik werden durch die entsprechende Potenzmenge oder eine Liste dargestellt.

Grammatik	`X = A*`
Mengen	$X = \wp(A)$ oder $X = \text{List}\,\langle A \rangle$

Im Folgenden wird aus Gründen einer übersichtlichen und verständlichen Notation auf spezifische Komponenten eines Tupels aufgrund seines Namens (getrennt durch einen Punkt) referenziert. Zum Beispiel verweist $g.Name$ auf den Namen der Grammatik g und ist somit eine Kurzform für die Projektion $\pi_2(g)$ auf die zweite Komponente des Tupels. Bei zwei gleichnamigen Komponenten wird zusätzlich eine Nummer verwendet, das heißt $Association.AssociationEnd_1$ ist eine Kurzform für die Projektion auf die zweite Komponente und $Association.AssociationEnd_2$ auf die vierte Komponente (also die zweite vom Typ *AssociationEnd*). Für die erste Komponente kann auf die Nummerierung verzichtet werden. Bei der Referenzierung können Ketten entstehen, wobei bei mengenwertigen Ausdrücken Flattening eingesetzt wird, das heißt $g.ASTRule.IType$ bezeichnet die Menge aller *IType*, die in AST-Produktionen verwendet werden, und ist keine Menge von Mengen. Für Komponenten, die aus disjunkten Teilmengen bestehen, kann auch direkt der Name einer Teilmenge verwendet werden. So ist $g.LexProd$ eine abkürzende Schreibweise für $\{l \mid l \in g.Prod,\ l \in LexProd\}$. Für Komponenten, die nur aus einer Subkomponente bestehen, kann auf eine Referenzierung der Subkomponente verzichtet werden. So ist zum Beispiel für $i \in InterfaceProd$ die Bezeichnung $i.IType$ eine Kurzschreibweise für $i.IType.Name$, falls aus dem Kontext klar wird, dass Namen verwendet werden.

Eine Alternative zu dem hier beschriebenen Vorgehen wäre gewesen, die aus dem MontiCore-Grammatikformat induzierte abstrakte Syntax in Form von UML/P-Klassendiagrammen zu verwenden. Dieses wurde verworfen, weil genau dieser Ableitungsprozess in diesem Abschnitt definiert werden soll. Dieser Kreisschluss, der allen Systemen, die im Bootstrapping-Verfahren entwickelt werden, zugrunde liegt, wird mit dem hier eingesetzten Verfahren aufgebrochen und eine einfachere Theorie verwendet, um die Bedeutung des Grammatikformats zu erklären.

5.3 Kontextbedingungen einer MontiCore-Grammatik

Die semantischen Abbildungen in diesem Kapitel lassen sich nur für wohlgeformte Grammatiken definieren. Die folgende Aufzählung listet die Kontextbedingungen übersichtsartig auf, die sich aufgrund der in diesem Kapitel formalisierten Essenz ergeben. Die detaillierten Definitionen finden sich in Abschnitt 5.4.

Eine Grammatik ist wohlgeformt, wenn folgende Bedingungen gelten:

1. Eindeutigkeit der Produktionsnamen unter Vererbung, das heißt jeder Produktionsname wird nur einmal verwendet (vgl. Definition 5.4.2)

2. Eindeutigkeit der Produktionsreferenzen, das heißt an jeder Stelle der Grammatik, an der Produktionsnamen verwendet werden dürfen, wird nur auf existierende Produktionen verwiesen (vgl. Definition 5.4.3)

3. Eindeutigkeit der Typnamen unter Vererbung, das heißt jeder Typ wird nur durch eine Produktion oder eine Menge von Klassenproduktionen definiert (vgl. Definition 5.4.4)

4. Eindeutigkeit der Typreferenzen, das heißt an jeder Stelle der Grammatik, an der Typnamen verwendet werden dürfen, wird nur auf existierende Typen verwiesen (vgl. Definition 5.4.5)

5. Für jede Produktion der Grammatik lässt sich der definierte Typ (DType) und der returnierte Typ (Type) ohne Typfehler bestimmen (vgl. Definitionen 5.4.9 und 5.4.11)

6. Für jeden Typ darf maximal eine AST-Regel existieren (vgl. Definition 5.4.13)

7. Die Grammatik muss korrekt typisierte Produktionen besitzen, das heißt die Vererbungsbeziehungen und die Typen der Paramter müssen korrekt sein, wobei hier noch keine Attribute betrachtet werden (vgl. Definition 5.4.16)

8. Die Operation att_g^T muss sich für jeden Typ einer Grammatik gültig sein, sich also die Attribute für jeden Typen korrekt berechnen lassen (vgl. Definition 5.4.22)

9. Die Operation $\text{att}_g^{T'}$ muss für jeden Typ einer Grammatik gültig sein und somit die die Vererbung respektieren (vgl. Definition 5.4.22)

10. Die Assoziationsnamen müssen eindeutig sein und sich von den Produktionsnamen unterscheiden (vgl. Definitionen 5.4.28 und 5.4.29)

11. Die Namen der navigierbaren Assoziationsenden dürfen nicht mit den Namen der Attribute und Kompositionen kollidieren (vgl. Definition 5.4.31)

12. Die Variablen innerhalb einer Grammatik müssen wohldefiniert sein, so dass sich eindeutig ein Typ bestimmmen lässt (vgl. Definition 5.4.33)

13. Die Nichtterminale innerhalb einer Grammatik müssen valide sein, das heißt alle Nichtterminale müssen bei der Verwendung von Produktionsparametern die Typisierung einhalten (vgl. Definition 5.4.37)

5.4 Hilfsstrukturen für das MontiCore-Grammatikformat

Die Semantikabbildung lässt sich nur für Grammatiken sinnvoll definieren, die bestimmte Kontextbedingungen erfüllen und daher wohlgeformt sind. In diesem Abschnitt werden die notwendigen Hilfsdatenstrukturen und -abbildungen definiert.

5.4.1 Namen

Innerhalb einer MontiCore-Grammatik wird je ein Namensraum für Produktionen und Typen verwendet. Dabei besitzen Produktionen und Typen jeweils einen eindeutigen Namen. Beide Namensräume sind weitesgehend identisch, weil mit der Ausnahme von mehrteiligen Klassenproduktionen jede Produktion einen gleichnamigen Typen definiert. Im Gegensatz zu BNF-artigen Notationen, wo die mehrfache Verwendung eines Nichtterminals auf der linken Seite einer Produktion eine alternative Ableitung darstellt, darf in dieser ENBF-artigen Notation jedes Nichtterminal nur einmal links erscheinen.

Definition 5.4.1 (Eindeutigkeit von Produktionsnamen)**.** Die Namen der Produktionen einer Grammatik g sind eindeutig, wenn $\text{PN}_g \subset \wp(\mathit{Name} \times \mathit{Grammar} \times \mathit{Prod})$ eine partielle Funktion ist ($\text{PN}_g : \mathit{Name} \to \mathit{Grammar} \times \mathit{Prod}$).

$$\begin{aligned}\mathrm{PN}_g \;=_{\text{def}}\; & \{\ (p.Name, g, p) \mid p \in g.LexProd\ \} \cup \\ & \{\ (p.Name, g, p) \mid p \in g.EnumProd\ \} \cup \\ & \{\ (p.Name, g, p) \mid p \in g.ExternalProd\ \} \cup \\ & \{\ (p.IType.Name, g, p) \mid p \in g.InterfaceProd\ \} \cup \\ & \{\ (p.CType.Name, g, p) \mid p \in g.AbstractProd\ \} \cup \\ & \{\ (p.Name, g, p) \mid p \in g.ClassProd\ \}\end{aligned}$$

Diese Eigenschaft muss für die geordnete Grammatikvererbung so erweitert werden, dass sie auch bei Berücksichtigung der Obergrammatiken und der geerbten Produktionen gilt. Dabei wird die Menge der Produktionen durch die Gesamtmenge aller Produktionen aller Grammatiken bestimmt, wobei bei Namensgleichheit zunächst die Produktionen der Grammatik selbst verwendet werden und dann erst die Produktion aus der ersten Obergrammatik, in der eine Produktion mit diesem Namen definiert ist.

Definition 5.4.2 (Eindeutigkeit von Produktionsnamen unter Vererbung)**.** Die Namen der Produktionen einer Grammatik g mit den Obergrammatiken $g.Grammar = [g_1, \ldots, g_m]$ sind eindeutig unter Vererbung, wenn $\mathrm{PN}'_g \subset \wp(Name \times Grammar \times Prod)$ eine partielle Funktion ist ($\mathrm{PN}'_g : Name \to Grammar \times Prod$).

$$\begin{aligned}\mathrm{PN}'_g =_{\text{def}} \mathrm{PN}_g \cup \textstyle\bigcup_{i\in\{1,\ldots,m\}} \{\ (n', g', p') \quad \mid \quad & (n', g', p') \in \mathrm{PN}'_{g_i}\ , \\ & \nexists (n'', g'', p'') \in \mathrm{PN}_g \cup \textstyle\bigcup_{j<i} \mathrm{PN}'_{g_j} : n' = n''\}\end{aligned}$$

Mit Hilfe der Eindeutigkeit von Produktionsnamen unter Vererbung lässt sich definieren, wann eine Grammatik eindeutig bezüglich der Produktionsreferenzen definiert ist. Diese Eigenschaft bezeichnet, dass innerhalb einer Grammatik nicht auf undefinierte Produktionen verwiesen wird, sondern alle referenzierten Produktionen direkt in der Grammatik definiert sind oder durch Vererbung zur Verfügung stehen.

Definition 5.4.3 (Eindeutigkeit der Produktionsreferenzen)**.** Eine MontiCore-Grammatik g ist eindeutig bezüglich der Produktionsreferenzen definiert, wenn ihre Obergrammatiken $g.Grammar = [g_1, \ldots, g_m]$ eindeutig bezüglich der Produktionsreferenzen definiert sind und zusätzlich die folgenden Bedingungen gelten:

(1) Implementierungsbeziehungen existieren.
$$\begin{aligned}\forall n \;\in\; & g.InterfaceProd.Inheritance.Name \;\cup \\ & g.AbstractProd.Implementation.Name \;\cup \\ & g.ClassProd.Implementation.Name :\end{aligned}$$

(a) *Referenzierte Produktionen existieren.*
$n \in \mathrm{def}(\mathrm{PN}'_g)$

(b) *Referenzierte Produktionen sind Schnittstellenproduktionen.*
$\mathrm{PN}'_g(n).Prod \in \mathrm{PN}'_g(n).Grammar.InterfaceProd$

(2) Vererbungsbeziehungen existieren.
$\forall n \in g.AbstractProd.Inheritance.Name \cup g.ClassProd.Inheritance.Name :$

(a) *Referenzierte Produktionen existieren.*
$n \in \mathrm{def}(\mathrm{PN}'_g)$

(b) *Referenzierte Produktionen sind abstrakte oder Klassenproduktionen.*
$\mathrm{PN}'_g(n).Prod \in \mathrm{PN}'_g(n).Grammar.AbstractProd \cup \mathrm{PN}'_g(n).Grammar.ClassProd$

(3) Nichtterminale verweisen auf Produktionen
Für ein $n \in NonTerminal$, das im Körper oder in den Produktionsparametern einer Klassenproduktion vorkommt, muss gelten: $n.Name \in \mathrm{def}(\mathrm{PN}'_g(n))$

5.4.2 Typisierung

Der zweite Namensraum sind die Typnamen innerhalb einer Grammatik, die zum Beispiel in AST-Produktionen und Assoziationen verwendet werden. Die einzige Unterschied zu den Produktionsnamen ist hier, dass bei mehrteiligen Klassenproduktionen mehrere Produktionen zu einem Typ beitragen können.

Definition 5.4.4 (Eindeutigkeit von Typnamen unter Vererbung)**.** Sind die Produktionsnamen einer Grammatik g unter Vererbung eindeutig, bilden die Typnamen eine partielle Funktion $\mathrm{TN}'_g : Name \to \wp(Grammar \times Prod)$. Sofern die Typnamen von Klassenproduktionen nicht mit anderen Produktionsnamen interferieren, d. h. zusätzlich gilt $\#\,\mathrm{TN}'_g(n) > 1 \Rightarrow \forall (g,p) \in \mathrm{TN}'_g(n) : p \in ClassProd$, sind die Typnamen eindeutig unter Vererbung.

$$\begin{aligned}\mathrm{TN}'_g \;=_{\mathrm{def}}\; & \{\ (n,\gamma,p) \mid (n,\gamma,p) \in \mathrm{PN}'_g,\ p \notin \gamma.ClassProd\ \} \cup \\ & \{\ (n.CType_1.Name,\gamma,p) \mid (n,\gamma,p) \in \mathrm{PN}'_g,\ p \in \gamma.ClassProd\ \}\end{aligned}$$

Mit Hilfe der Eindeutigkeit von Typnamen unter Vererbung lässt sich definieren, wann eine Grammatik korrekt bezüglich der Typenreferenzen definiert ist, also nicht auf undefinierte Typen verwiesen wird.

Definition 5.4.5 (Eindeutigkeit der Typreferenzen)**.** Eine Grammatik g mit Obergrammatiken $g.Grammar = [g_1, \ldots, g_m]$ ist eindeutig bezüglich der Typenreferenzen definiert, wenn alle Obergrammatiken $g_1, \ldots, g_m$ eindeutig bezüglich der Typenreferenzen definiert sind und die folgenden Bedingungen gelten:

(1) Verweise auf allgemeine Typen
$$\begin{aligned}\forall n \;\in\; & g.ASTRule.Type.Name \;\cup \\ & g.Association.AssociationEnd_1.Type.Name \;\cup \\ & g.Association.AssociationEnd_2.Type.Name \;\cup \\ & g.ClassProd.Parameter.Name_2 \;\cup \\ & \{t \mid t \in g.ASTRule.AttributeInAST.AttType \cap Type\} :\end{aligned}$$

(a) *Referenzierter Typ existiert.*
$n \in \mathrm{def}(\mathrm{TN}'_g)$

(b) *Referenzierter Typ muss durch Parserproduktion definiert sein.*
$\forall (\pi,\gamma) \in \mathrm{TN}'_g(n) : \pi \in \gamma.ParserProd$

(2) Implementierte Schnittstellen
$\forall n \in g.ASTRule.IType.Name :$

(a) *Referenzierter Typ existiert.*
$n \in \mathrm{def}(\mathrm{TN}'_g)$

(b) *Referenzierter Typ muss durch Schnittstellenproduktion definiert sein.*
$\forall(\pi, \gamma) \in \mathrm{TN}'_g(n) : \pi \in \gamma.InterfaceProd$

(3) Oberklassen
$\forall n \in g.ASTRule.CType.Name$:

(a) *Referenzierter Typ existiert.*
$n \in \mathrm{def}(\mathrm{TN}'_g)$

(b) *Referenzierter Typ muss durch abstrakte Produktion oder Klassenproduktionen definiert sein.*
$\forall(\pi, \gamma) \in \mathrm{TN}'_g(n) : \pi \in \gamma.AbstractProd \cup \gamma.ClassProd$

(4) Verweise auf Java-Typen
Die folgenden Java-Datentypen müssen existieren.

(a) $g.LexProd.JType$

(b) $g.ExternalProd.JType$

(c) $\{t \mid t \in g.ASTRule.AttributeInAST.AttType \cap JType\}$

5.4.3 Subtypisierung

Definition 5.4.6 (Abbildung von Namen auf vollqualifizierte Javatypen durch qname)**.** Die Funktion qname : $List\,\langle Name\rangle \times Name \to JType$ bildet eine Paketbezeichnung und einen Namen auf einen vollqualifizierten Typ ab: $([n_1, ..n_m], n) \mapsto n_1 \cdot \text{“.”} \cdot n_2 ... \cdot n_m \cdot \text{“.”} \cdot n$

Definition 5.4.7 (Obertyp)**.** Das Symbol $\top$ bezeichnet einen validen Obertyp aller Klassen und Schnittstellen, die aus den Produktionen einer Grammatik abgeleitet werden, und wird auch als `ASTNode` bezeichnet.

Definition 5.4.8 (Typfehler)**.** Aus technischen Gründen wird ein Typ $\tilde{\tau}$ verwendet, der Supertyp aller Typen (inklusive $\top$) ist und einen Typfehler darstellt. $\tilde{\tau}$ ist nicht zu verwechseln mit der Rolle der Klasse *Object* in objektorientierten Programmiersprachen, der einen validen Obertyp aller Klassen darstellt und hier als $\top$ bezeichnet wird.

Für die Produktionen einer Grammatik wird zunächst über die Abbildung DType festgelegt, welche Typen sie definieren. Für Enumerationsproduktionen, Schnittstellenproduktionen und abstrakte Produktionen gilt, dass diese nur einmal definiert werden dürfen. Ist eine Enumerations-, Schnittstellen- oder abstrakte Produktion mit gleichem Namen in zwei Obergrammatiken vorhanden, dann muss sie vom gleichen Typ sein. Dieses trifft typischerweise dann zu, wenn beide Obergrammatiken ihrerseits eine gemeinsame Obergrammatik besitzen, in der die entsprechende Produktion definiert ist.

Die Gründe für diese Art der Festlegung liegen bei Enumerationsproduktionen bei der fehlenden Möglichkeit zur Vererbung. Bei Schnittstellen- und abstrakten Produktionen greift das Entwurfsprinzip, dass nur für in einer Grammatik tatsächlich spezifizierte Produktionen eine Klasse oder Schnittstelle in der dazugehörigen abstrakten Syntax existieren soll. Für sie gibt es praktisch immer schon eine implementierende Klasse bzw. eine Unterklasse in der Grammatik, in der sie definiert sind, weil diese meistens der Grund für diese Art der Produktion ist. Könnten die Schnittstellen- und abstrakten Produktionen überschrieben werden, müsste für die implementierenden Klassen bzw. Unterklassen jeweils einen neue Klasse enstehen, ohne dass eine Produktion explizit aufgeführt würde.

Bei Klassen ist Überschreiben prinzipiell möglich, sofern keine Unterklassen existieren. Bei der Verwendung von mehreren Obergrammatiken muss jedoch vermieden werden,

dass dabei gleichnamige Klassen aus verschiedenen Obergrammatiken zusammengeführt werden, weil sonst ohne explizite Produktion eine Unterklasse entstehen müsste.

Definition 5.4.9 (Typabbildung DType_g)**.** Die definierten Typen einer Grammatik g mit den Obergrammatiken $g.Grammar = [g_1, \ldots, g_m]$ werden durch die partielle Funktion $\mathrm{DType}_g : Name \to JType$ festgelegt, die einen Produktionsnamen auf einen vollqualifizierten Java-Datentyp abbildet, den sie definiert. Die Funktion DType erfordert die Eindeutigkeit der Produktionsnamen unter Vererbung. $\mathrm{DType}_g(n)$ ist für $\mathrm{PN}'_g(n) = (\gamma, p)$ wie folgt definiert:

(1) Lexikalische Produktionen
$p \in \gamma.LexProd : \mathrm{DType}_g(n) =_{\mathrm{def}} p.JType$

(2) Enumerationsproduktion, Schnittstellenproduktionen und abstrakte Produktionen
Enumerationsproduktion, Schnittstellenproduktionen und abstrakte Produktionen dürfen nur in einer einzigen Grammatik definiert sein. Diese kann jedoch (transitiv) mehrfache Obergrammatik der betrachteten Grammatik sein:
$p \in \gamma.EnumProd \cup \gamma.InterfaceProd \cup \gamma.AbstractProd$:

$$\mathrm{DType}_g(n) =_{\mathrm{def}} \begin{cases} t & : \bigcup_{0 \le i \le m} T_i = \{t\} \\ \tilde{\tau} & : \text{sonst} \end{cases}$$

$$T_i = \begin{cases} \mathrm{qname}(g.QName, n) & : i = 0, n \in \mathrm{def}(\mathrm{PN}_g) \\ \mathrm{DType}_{g_i}(n) & : i > 0, n \in \mathrm{def}(\mathrm{PN}'_{g_i}) \\ \emptyset & : \text{sonst} \end{cases}$$

(3) Externe Produktionen
$p \in ExternalProd : \mathrm{DType}_g(n) =_{\mathrm{def}} p.JType$

(4) Klassenproduktionen
Zur Bestimmung des Typs einer Klassenproduktion müssen alle Regeln berücksichtigt werden, die den Typ definieren. Dabei muss beachtet werden, dass entweder eine Regel in der aktuellen Grammatik existiert, die den Typ definiert, oder alle Produktionen aus derselben Obergrammatik stammen.
$p \in \gamma.ClassProd,\ X = \mathrm{TN}'_g(p.CType_1),\ qn = \mathrm{qname}(g.QName, p.CType_1.Name)$

$$\mathrm{DType}_g(n) =_{\mathrm{def}} \begin{cases} qn & : \exists(\gamma, p') \in X : \gamma = g \\ \mathrm{DType}_\gamma(n) & : \exists i \in \{1, \ldots, m\} : X = \mathrm{TN}'_{g_i}(p.CType_1) \\ \tilde{\tau} & : \text{sonst} \end{cases}$$

Die Funktion DType ist für Produktionsnamen aus $\mathrm{def}(\mathrm{PN}_g)$ definiert. In der semantischen Abbildung ist jedoch auch an manchen Stellen notwendig, für Typreferenzen den entsprechenden Java-Datentyp zu bestimmen. Diese Abbildung ist mit DType nahezu identisch und unterscheidet sich nur für mehrteilige Klassenproduktionen, weil hier die Zuordnung nicht eins zu eins ist. Die Abbildung DType^{TN} wandelt DType entsprechend ab.

Definition 5.4.10 (Typabbildung DType_g^{TN})**.** Die durch Typreferenzen definierten Typen werden durch die partielle Funktion $\mathrm{DType}_g^{TN} : Name \to JType$ festgelegt. Die Funktion DType_g^{TN} erfordert die Eindeutigkeit der Produktionsnamen unter Vererbung. DType_g^{TN} ist wie DType_g definiert, mit dem einzigen Unterschied, dass für Klassenproduktionen der definierende Typ anstatt der Produktionsnamen verwendet wird. $\mathrm{DType}_g^{TN}(n)$ ergibt sich für $g \in Grammar,\ g.Grammar = [g_1, \ldots, g_m]$ bei Klassenproduktionen aus dem definierten Typ aller definierenden Klassenproduktionen. Durch die Konstruktion von DType_g ist

sichergestellt, dass für Klassenproduktionen alle Elemente aus $\mathrm{TN}'_g(n)$ zum gleichen Typ führen. Bei anderen Arten von Produktionen besteht $\mathrm{TN}'_g(n)$ nur aus einem Element.

$$DType_g^{TN}(n) =_{\text{def}} t, \text{ wobei } \{t\} = \bigcup_{(\gamma,p)\in \mathrm{TN}'_g(n)} \begin{cases} DType_g(p.Name) & : \ p \in \gamma.ClassProd \\ DType_g(n) & : \ \text{sonst} \end{cases}$$

Für die Produktionen einer Grammatik wird die Abbildung Type_g definiert, welche die durch die Produktionen returnierten Typen beschreibt. Diese Funktion stimmt mit DType nur für Klassenproduktionen nicht überein, weil DType auch einen Subtyp zurückliefern kann.

Definition 5.4.11 (Typabbildung Type_g)**.** Die partielle Funktion $\mathrm{Type}_g : Name \to JType$ bildet einen Produktionsnamen in einer Grammatik g auf einen vollqualifizierten Java-Datentyp ab. Die Funktion Type erfordert die Eindeutigkeit der Produktionsnamen unter Vererbung. Type ist wie DType definiert, mit dem einzigen Unterschied, dass für Klassenproduktionen der Rückgabewert und nicht der definierte Typ verwendet wird. Für alle anderen Elemente gelten unverändert die Angaben aus Definition 5.4.9, so dass gilt $\mathrm{Type}_g(n) =_{\text{def}} \mathrm{DType}_g(n)$ mit der Ausnahme:

$\mathrm{Type}_g(n)$ ist für $g \in Grammar,\ g.Grammar = [g_1, \ldots, g_m],\ \mathrm{PN}'_g(n) = (\gamma, p)$ wie folgt definiert:

(4) Klassenproduktionen
$p \in \gamma.ClassProd : \mathrm{Type}_g(n) =_{\text{def}} \mathrm{DType}_g^{TN}(p.CType_2.Name)$

Definition 5.4.12 (Typ eines formalen Parameters)**.** Für einen formalen Parameter pr wird $\mathrm{Type}_g^{Pr}(pr)$ als Typ des Parameters bezeichnet:
$\mathrm{Type}_g^{Pr}(pr) =_{\text{def}} \mathrm{DType}_g^{TN}(pr.Name_2)$.

Definition 5.4.13 (Hilfsfunktion ast_g)**.** Für eine Grammatik g, für die sich die Typen der Produktionen ohne Typfehler bestimmen lassen, lässt sich die Menge wie folgt definieren: $\mathrm{ast}_g =_{\text{def}} \{(a.Type.Name, a) \mid a \in g.ASTRule\}$. Für eine wohlgeformte Grammatik ist $\mathrm{ast}_g(n)$ eine partielle Funktion $Name \to ASTRule$, weil nur eine einzige AST-Produktion für jeden Typ erlaubt ist. Für Produktionen, für die keine AST-Regel definiert ist, liefert die Funktion $\mathrm{ast}_g(n)$ die leere Menge als Rückgabewert.

Die Untertyprelation $\succ_g$ bezeichnet die Vererbungs- und Implementierungsbeziehungen, die sich aufgrund der Struktur einer Grammatik ergeben. Sie definiert somit, welche Typen aufgrund der Produktionen der Grammatik voneinander erben.

Definition 5.4.14 (Untertyprelation $\succ_g$ für eine Grammatik)**.** Für eine Grammatik g mit Obergrammatiken $g.Grammar = [g_1, \ldots, g_m]$ ist die Relation $\succ_g \subset \wp(JType \times JType)$ die kleinste Menge, die die folgenden Eigenschaften erfüllt. Dabei bedeutet $a \succ_g b$ dass a ein direkter Untertyp von b ist, wobei der verwendete Operator an das Vererbungssymbol in UML-Klassendiagrammen erinnern soll. Die Untertyprelation $\succ_g^*$ bezeichnet den reflexiven und transitiven Abschluss von $\succ_g$.

(1) Grammatikvererbung

(a) *Vererbungsrelation in Obergrammatiken*
Für $g.Grammar = [g_1, \ldots, g_m]$ gilt $\forall i \in \{1, \ldots, m\} : \succ_{g_i} \subset \succ_g$

(b) *Überschreiben von Klassenproduktionen*
Für $n \in \mathrm{def}(DType_g), \mathrm{PN}'_g(n).Prod \in ClassProd$
$\forall_{i\in\{1,\ldots,m\}} \mathrm{PN}'_{g_i}(n).Prod \in ClassProd \Rightarrow \mathrm{DType}_g(n) \succ_g \mathrm{DType}_{g_i}(n)$

(2) Java-Datentypen
Wenn x und y Javatypen sind, wie sie in lexikalischen oder externen Produktionen verwendet werden, also nicht aus Produktionen erzeugt sind, sondern nur dort verwendet werden und in einer Subtypbeziehung zueinander stehen, gilt: $x \succ_g y$.

(3) Schnittstellenproduktionen
Für $n \in \text{def}(DType_g), i = \text{PN}'_g(n).Prod \in InterfaceProd$
$\forall t \in SI_1 \cup SI_2 : DType_g(n) \succ_g t$, wobei

(a) Vererbung in Schnittstellenproduktion definiert
$SI_1 =_{\text{def}} \{\text{DType}_g(x.Name) \mid x \in i.Inheritance\}$

(b) Vererbung in AST-Regel definiert
$SI_2 =_{\text{def}} \{\text{DType}_g^{TN}(x.Name) \mid x \in ast(g,n).IType\}$

(4) Abstrakte Produktionen und Klassenproduktionen
Für $n \in \text{def}(DType_g), p = \text{PN}'_g(n).Prod \in AbstractProd \cup ClassProd$
$\forall t \in SI_1 \cup SI_2 \cup SC_1 \cup SC_2 : DType_g(n) \succ_g t$, wobei

(a) Schnittstellen in abstrakter Produktion definiert
$SI_1 =_{\text{def}} \{\text{DType}_g(x.Name) \mid x \in p.Implementation\}$

(b) Schnittstellen in AST-Regel definiert
$SI_2 =_{\text{def}} \{\text{DType}_g^{TN}(x.Name) \mid x \in ast(g,n).IType\}$

(c) Vererbung in abstrakter Produktion definiert
$SC_1 = \{\text{DType}_g(x.Name) \mid x \in p.Inheritance\}$

(d) Vererbung in AST-Regel definiert
$SC_2 = \{\text{DType}_g^{TN}(x.Name) \mid x \in ast(g,n).CType\}$

Die Parserproduktionen einer Grammatik, also die Schnittstellenproduktionen, abstrakte Produktionen oder Klassenproduktionen, verfügen über formale Parameter. Diese müssen zu den formalen Parametern anderer wie die der verschatteten Produktionen passen. Damit diese Eigenschaft formuliert werden kann, wird zunächst das Prädikat $p \succ^{\pi}_{g,g'} p'$ definiert, dass wahr ist, wenn die formalen Parameter von p eine Verfeinerung der formalen Parameter von p' darstellen.

Definition 5.4.15 (Korrekte Verfeinerung formaler Parameter $\succ^{\pi}_{g,g'}$)**.** Eine Parserproduktion $o \in g.ParserProd$ einer Grammatik g verfeinert die formalen Parameter einer zweiten Parserproduktion p einer Grammatik g' korrekt, wenn für $o.Parameter = [o_1, \ldots, o_l]$ und $p.Parameter = [p_1, \ldots, p_{l'}]$ die folgenden Bedingungen gelten (kurz: $o \succ^{\pi}_{g,g'} p$):

1. $l = l'$
2. $\forall i \in \{1, \ldots, l\} : \text{Type}^{Pr}_{g'}(p_i) \succ^*_g \text{Type}^{Pr}_g(o_i)$
3. $\forall i \in \{1, \ldots, l\} : o_i.Card.Min \leq p_i.Card.Min$
4. $\forall i \in \{1, \ldots, l\} : o_i.Card.Max \geq p_i.Card.Max$

Die Verfeinerung ist hier so definiert, dass beim Parsen kein Fehler aufgrund verletzter Kardinalitäten oder Typfehler auftreten kann, weil die Untertypen stets gleichgroße oder größere Kardinalitäten zulassen müssen und nur denselben oder einen Obertyp verwenden dürfen. Die formalen Parameter werden in den Produktionen, je nach *ProdElementType* zu

einem Attribut oder zu einer Variablen. Werden sie zu einem Attribut, greifen hier zusätzlich die im Folgenden dargestellten Regeln zur Attributbildung, die nur die Verwendung eines Typs oder eines Subtyps erlauben. Somit ergibt sich dann effektiv, dass für diese Parameter, wenn sie zum selben Attribut beitragen, nur derselbe Typ verwendet werden darf. Dies ist jedoch nicht zwangsläufig so ist, weil es keinerlei Einschränkungen in Bezug auf die Verwendung der Parameter gibt. So können zwei Produktionen o und p die Parameter korrekt verfeinern, die eine diese jedoch als Variablen und die andere diese als Attribute verwenden.

Auf diesen Definitionen aufbauend lässt sich erklären, wann eine Grammatik korrekt typisierte Produktionen besitzt. Diese Eigenschaft bedeutet, dass die durch die Produktionen definierten Typen stimmig sind, aber noch keine komplette Typsicherheit vorhanden ist, da hierfür noch die Attribute betrachtet werden müssen.

Definition 5.4.16 (Korrekt typisierte Produktionen)**.** Eine Grammatik g hat korrekt typisierte Produktionen, wenn alle Obergrammatiken $g.Grammar = [g_1, ..., g_m]$ korrekt typisierte Produktionen besitzen und folgende Bedingungen für $(n, g, p) \in \mathrm{PN}_g$ gelten:

(1) Lexikalische Produktionen
Lexikalische Produktionen verschatten nur andere lexikalische Produktionen und geben einen Subtyp zurück. Für $p \in g.LexProd$ muss gelten:

(a) $\forall i \in \{1, ..., m\} : (n, g', p') \in \mathrm{PN'}_{g_i} \Rightarrow p' \in g'.LexProd$

(b) $\forall i \in \{1, ..., m\} : (n, g', p') \in \mathrm{PN'}_{g_i} \Rightarrow \mathrm{DType}_g(p.Name) \succ_g^* \mathrm{DType}_{g'}(p'.Name)$

(2) Enumerationsproduktionen
Enumerationsproduktionen dürfen keine andere Produktion verschatten.
$p \in EnumProd \Rightarrow \forall i \in \{1, ..., m\} : \nexists (n, g', p') \in \mathrm{PN'}_{g_i}$

(3) Externe Produktionen
Für $p \in g.ExternalProd$ müssen die folgenden Bedingungen gelten:

(a) Externe Produktionen verschatten nur andere externe Produktionen.
$\forall i \in \{1, ..., m\} : (n, g', p') \in \mathrm{PN'}_{g_i} \Rightarrow p' \in g'.ExternalProd$

(b) Externe Produktionen returnieren einen Subtyp der verschatteten Produktion.
$\forall i \in \{1, ..., m\} : (n, g', p') \in \mathrm{PN'}_{g_i} \Rightarrow \mathrm{DType}_g(p.Name) \succ_g^* \mathrm{DType}_{g'}(p'.Name)$

(c) Der definierte Typ muss ein Untertyp des gemeinsamen Obertyps aller Klassen und Schnittstellen sein.
$\mathrm{DType}_g(p.Name) \succ_g^* \top$

(4) Schnittstellenproduktionen
Für $p \in g.InterfaceProd$ müssen die folgenden Bedingungen gelten:

(a) Schnittstellenproduktionen dürfen keine andere Produktion verschatten.
$\forall i \in \{1, ..., m\} : \nexists (n, g', p') \in \mathrm{PN'}_{g_i}$

(b) Die Parameter müssen eine Verfeinerung der Parameter der Oberproduktion sein:
$\forall x \in p.Inheritance : p \succ_{g,g'}^{\pi} \mathrm{PN'}_{g'}(x).Prod$ wobei $g' = \mathrm{PN}_g'(x).Grammar$

(5) Abstrakte Produktionen
Für $p \in g.AbstractProd$ müssen die folgenden Bedingungen gelten:

(a) Abstrakte Produktionen dürfen keine andere Produktion verschatten.
$\forall_{i \in \{1,\dots,m\}} : \nexists (n, g', p') \in \mathrm{PN}'_{g_i}$

(b) Die Parameter müssen eine Verfeinerung der Parameter Oberproduktion oder implementierten Schnittstelle sein:
$\forall x \in p.Inheritance \cup p.Implementation : p \succ^{\pi}_{g,g'} \mathrm{PN}'_{g'}(x).Prod$
wobei $g' = \mathrm{PN}'_{g}(x).Grammar$

(6) Klassenproduktionen
Für $p \in g.ClassProd$ müssen die folgenden Bedingungen gelten:

(a) Klassenproduktionen verschatten nur andere Klassenproduktionen oder abstrakte Produktionen.
$\forall i \in \{1, \dots, m\} : (n, g', p') \in \mathrm{PN}'_{g_i} \Rightarrow p' \in g'.ClassProd \cup g'.AbstractProd$

(b) Klassenproduktionen returnieren einen Subtyp des returnierten Typs der verschatteten Produktion. Der definierte Typ ist ebenfalls ein Subtyp, was jedoch durch die Konstruktion von $\succ_g$ gesichert ist.
$\forall i \in \{1, \dots, m\} : (n, g', p') \in \mathrm{PN}'_{g_i} \Rightarrow \mathrm{Type}_g(p.Name) \succ^*_g \mathrm{Type}_{g'}(p'.Name)$

(c) Verschattete Klassenproduktionen dürfen keine Untertypen in Obergrammatiken haben.
$\forall i \in \{1, \dots, m\} : \forall (n', g', p') \in \mathrm{PN}'_{g_i} : n \neq n' \Rightarrow \mathrm{DType}_{g_i}(n') \not\succ^*_g \mathrm{DType}_{g_i}(n)$

(d) Der definierte Typ muss ein Subtyp des returnierten Typs sein.
$\mathrm{DType}_g(n) \succ^*_g \mathrm{Type}_g(n)$

(e) Der returnierte Typ muss die Vererbungsbeziehungen erfüllen.
$\forall x \in p.Implementation \cup p.Inheritance : \mathrm{Type}_g(n) \succ^*_g \mathrm{Type}_g(x)$

(f) Die Parameter müssen eine Verfeinerung der Parameter der verschatteten Produktion sein:
$\forall i \in \{1, \dots, m\} : (n, g', p') \in \mathrm{PN}'_{g_i} \Rightarrow p \succ^{\pi}_{g,g_i} p'$

(g) Die Parameter müssen eine Verfeinerung der Parameter Oberproduktion oder implementierten Schnittstelle sein:
$\forall x \in p.Inheritance \cup p.Implementation : p \succ^{\pi}_{g,g'} \mathrm{PN}'_{g'}(x).Prod$
wobei $g' = \mathrm{PN}'_{g}(x).Grammar$

(7) Untertyprelation $\succ^*_g$

(a) Die Untertyprelation $\succ^*_g$ muss zyklenfrei sein.

(b) Schnittstellen dürfen nicht Untertyp einer (abstrakten) Klasse sein.

(8) Variationspunkt 1: Mehrfachvererbung in $\succ^*_g$:
Eine der folgenden Bedindungen muss für $\succ^*_g$ gelten:

(a) Mehrfachvererbung zwischen Klassentypen erlaubt

(b) Mehrfachvererbung zwischen Klassentypen nicht erlaubt.

5.4.4 Attribute und Kompositionen

Die Attribute und Kompositionsbeziehungen in der abstrakten Syntax ergeben sich aus den Produktionskörpern der Grammatik. Die Datenstruktur D wird zur Speicherung der Zwischenergebnisse verwendet, die sich aus Teilausdrücken von Produktionskörpern ergeben.

Definition 5.4.17 (Datenstruktur D)**.** Die Datenstruktur D hat die folgende Struktur.

D	$=$	$\wp(Attrib)$
$Attrib$	$=$	$Name_{5.1} \times \wp(Kind) \times Min_{5.1} \times Max_{5.1}$
$Kind$	$=$	$JType_{5.1} \cup ConstantValue \cup ProdReference \cup TypeReference$
$ConstantValue$, $ProdReference$, $TypeReference$	$=$	$Name_{5.1}$

Die Datenstruktur D wird im Folgenden für die primitiven Elemente eines Produktionskörpers definiert. Damit diese Definition entlang der Struktur einer Produktion in Definition 5.4.19 leichter formuliert werden kann, werden für D Hilfsoperationen benötigt, die der Grundstruktur einer Produktion mit kleenschen Sternen, Alternativen, Sequenzen, der Überschreibung durch Unterklassen (override) und der Überschreibung durch AST-Regeln (astoverride) entsprechen. Die beiden letzten Operationen unterscheiden sich durch die Behandlung von Konstanten, weil für Konstantenattribute entschieden wurde, unabhängig von der automatisch abgeleiteten Kardinalität, mehrfache Werte nur zuzulassen, wenn dieses explizit durch eine AST-Regel festgelegt wurde.

Definition 5.4.18 (Operationen auf D)**.** Auf der Datenstruktur D sind die folgenden Operationen definiert:

(1) Rechenregeln für $\mathbb{N} \cup \{*\}$
Dabei gelten für die verwendeten Zahlwerte aus $\mathbb{N} \cup \{*\}$ die üblichen Rechenregeln aus $\mathbb{N}$ und zusätzlich:
$\forall x \in \mathbb{N} \cup \{*\} : * + x = x + * = * = \max(x, *) = \max(*, x)$
$\forall x \in \mathbb{N} \cup \{*\} : \min(x, *) = \min(*, x) = x$

(2) Kleenescher Stern
kleene: $D \to D : d_1 \mapsto d$ mit

$$d = \{(n, K, 0, *) \mid (n, K, min, max) \in d_1\}$$

(3) Alternativen
alt: $D \times D \to D : d_1 \times d_2 \mapsto d$ mit

$$\begin{aligned} d = {} & \{(n, K, 0, max) \mid (n, K, min, max) \in d_1, \nexists (n, K', min', max') \in d_2\} \cup \\ & \{(n, K, 0, max) \mid (n, K, min, max) \in d_2, \nexists (n, K', min', max') \in d_1\} \cup \\ & \{(n, K, min, max) \mid (n, K_1, min_1, max_1) \in d_1, (n, K_2, min_2, max_2) \in d_2, \\ & \quad K = K_1 \cup K_2, min = \min(min_1, min_2), max = \max(max_1, max_2)\} \end{aligned}$$

(4) Sequenzen
seq: $D \times D \to D : d_1 \times d_2 \mapsto d$ mit

$$\begin{aligned} d = {} & \{(n, K, min, max) \mid (n, K, min, max) \in d_1, \nexists (n, K', min', max') \in d_2\} \cup \\ & \{(n, K, min, max) \mid (n, K, min, max) \in d_2, \nexists (n, K', min', max') \in d_1\} \cup \\ & \{(n, K, min, max) \mid (n, K_1, min_1, max_1) \in d_1, (n, K_2, min_2, max_2) \in d_2, \\ & \quad K = K_1 \cup K_2, min = min_1 + min_2, max = max_1 + max_2\} \end{aligned}$$

(5) Override
override: $D \times D \to D : d_1 \times d_2 \mapsto d$ mit

$$\begin{aligned} d \;=\; & \{(n, K, min, max) \mid (n, K, min, max) \in d_1\} \cup \\ & \{(n, K, min, max) \mid (n, K, min, max) \in d_2, \nexists (n, K', min', max') \in d_1\} \end{aligned}$$

(6) Astoverride
astoverride: $D \times D \to D : d_1 \times d_2 \mapsto d$ mit

$$\begin{aligned} d \;=\; & \{(n, K, min, max) \mid (n, K, min, max) \in d_1\} \cup \\ & \{(n, K, min, max) \mid (n, K, min, max) \in d_2, \nexists (n, K', min', max') \in d_1, \\ & \qquad\qquad \exists k \in K : k \notin \mathit{ConstantValue}\} \cup \\ & \{(n, K, 0, 1) \mid (n, K, min, max) \in d_2, \nexists (n, K', min', max') \in d_1, \\ & \qquad\qquad \forall k \in K : k \in \mathit{ConstantValue}\} \end{aligned}$$

Definition 5.4.19 (Funktion att_g)**.** Die Funktion att_g dient zur Berechnung der Datenstruktur aus den Produktionen einer Grammatik g. Die Funktion ist dabei rekursiv definiert und verwendet die Operationen aus Definition 5.4.18.

(1) Schnittstellenproduktionen
Für $i \in \mathit{InterfaceProd}$ gilt:
$\mathrm{att}_g(i) =_{\mathrm{def}} \emptyset$

(2) Abstrakte Produktionen
Für $a \in \mathit{AbstractProd}$ gilt:
$\mathrm{att}_g(a) =_{\mathrm{def}} \emptyset$

(3) Klassenproduktionen
Für $p, p_1, \ldots, p_n \in \mathit{ClassProd}$ gilt:

(a) $\mathrm{att}_g(p) =_{\mathrm{def}} \mathrm{seq}(x, \mathrm{att}_g(p.\mathit{Body}))$, wobei $x = \mathit{seq}(\mathrm{att}_g(l_1), \mathit{seq}(..., \mathrm{att}_g(l_n))...)$
für $p.\mathit{ProdElement} = [l_1, \ldots, l_n] \in \mathit{List}\,\langle \mathit{Parameter} \rangle$

(b) $\mathrm{att}_g(\{p_1, \ldots, p_n\}) =_{\mathrm{def}} \mathrm{alt}(\mathrm{att}_g(p_1), \mathrm{alt}(\ldots \mathrm{att}_g(p_n)))$

(4) Formale Parameter
Für $p \in \mathit{Parameter}$ gilt:

(a) Für $p.\mathit{ProdElementType} = \mathrm{AST}$ gilt:
$\mathrm{att}_g(p) =_{\mathrm{def}} \{(p.\mathit{Name}_1, \{p.\mathit{Name}_2\}, p.\mathit{Card}.\mathit{Min}, p.\mathit{Card}.\mathit{Max})\}$

(b) Für $p.\mathit{ProdElementType} \in \{\mathrm{VarAssign}, \mathrm{Ignore}\}$ gilt:
$\mathrm{att}_g(p) =_{\mathrm{def}} \emptyset$

(5) Struktur der Terme
Für $t, t_1, t_2 \in \mathcal{T}_{\Sigma_G}(\mathit{ProdElement} \cup \mathit{Action})$ gilt:

(a) $\mathrm{att}_g(t_1|t_2) =_{\mathrm{def}} \mathrm{alt}(\mathrm{att}_g(t_1), \mathrm{att}_g(t_2))$

(b) $\mathrm{att}_g(t_1 t_2) =_{\mathrm{def}} \mathrm{seq}(\mathrm{att}_g(t_1), \mathrm{att}_g(t_2))$

(c) $\mathrm{att}_g(t^*) =_{\mathrm{def}} \mathrm{kleene}(\mathrm{att}_g(t))$

(6) ProdElements
Für $x \in ProdElement$ gilt:

(a) Für $x.ProdElementType = \text{AST}$ gilt:

i. $c = x.ProdElementRef \in Constant : \text{att}_g(x) =_{\text{def}} \{(x.Name, \{z\}, 1, 1)\}$ mit $z = c.Name \in ConstantValue$

ii. $n = x.ProdElementRef \in NonTerminal, \text{PN}'_g(n).Prod \notin LexProd :$ $\text{att}_g(x) =_{\text{def}} \{(x.Name, \{z\}, 1, 1)\}$ mit $z = n.Name \in ProdReference$

iii. $n = x.ProdElementRef \in NonTerminal, p = \text{PN}'_g(n).Prod \in LexProd :$ $\text{att}_g(x) =_{\text{def}} \{(x.Name, \{z\}, 1, 1)\}$ mit $z = p.JType \in JType$

iv. $t = x.ProdElementRef \in Terminal : \text{att}_g(x) =_{\text{def}} \{(x.Name, \{z\}, 1, 1)\}$ mit $z = \texttt{java.lang.String} \in JType$

(b) Für $x.ProdElementType \in \{\text{VarAssign}, \text{Ignore}\}$ gilt:
$\text{att}_g(x) =_{\text{def}} \emptyset$

(7) Actions
Für $x \in Action$ gilt:
$\text{att}_g(x) =_{\text{def}} \emptyset$

(8) AST-Regeln
Für $p \in ASTRule, p.AttributeInAST = \{a_1, \dots, a_n\}$ gilt:
$\text{att}_g(p) =_{\text{def}} \text{seq}(\text{att}_g(a_1), \text{seq}(\dots, seq(\text{att}_g(a_n)) \dots)$

(9) Attribute
Für $a \in AttributeInAST$ gilt:
$\text{att}_g(a) =_{\text{def}} \{(a.Name, \{a.AttType\}, a.Card.Min, a.Card.Max)\}$

(10) Leere Menge
$\text{att}_g(\emptyset) =_{\text{def}} \emptyset$

Die Datenstruktur D lässt sich für beliebige Grammatiken und Produktionen berechnen. Wichtig ist jedoch, dass sich daraus auch eindeutig Attribute ableiten lassen und die Elemente der Produktionen nicht widersprüchlich sind. Ein Widerspruch stellt zum Beispiel ein Attribut dar, das sowohl Konstantenwerte als auch Referenzen auf andere Produktionen als Typ hat. Für Verweise auf andere Produktionen muss sich ein Typ bestimmen lassen, der Obertyp aller anderen ist. Dadurch sind alle Zuweisungen an das Attribut typsicher. Aufgrund der Zyklenfreiheit der Vererbungsrelation ist dieser Typ eindeutig, sofern ein solcher Typ existiert. Daher wird im Folgenden definiert, wann eine Datenstruktur wohldefiniert ist, was zunächst nicht die Vererbung mit einschließt.

Definition 5.4.20 (Wohldefiniertheit von Datenstrukturen)**.** Eine Datenstruktur D ist innerhalb einer Grammatik g für einen Datentyp $t \in JType$ wohldefiniert, wenn alle Attribute in $D.Attrib$ wohldefiniert sind.

Definition 5.4.21 (Wohldefiniertheit von Attributen)**.** Ein Attribut $a \in Attrib$ ist innerhalb einer Grammatik g für einen Datentyp $t \in JType$ wohldefiniert, wenn die folgenden Bedingungen gelten:

(1) Art des Attributs ist eindeutig
$\forall x \in a.Kind : x \in JType$ oder
$\forall x \in a.Kind : x \in ConstantValue$ oder
$\forall x \in a.Kind : x \in ProdReference \cup TypeReference$

(2) Typisierung der Attribute
$\forall z \in a.Kind : z \in JType \Rightarrow \exists y \in a.Kind : \forall x \in a.Kind : x \succ_g^* y$

Anmerkung: Für wohldefinierte Attribute wird y als Typ des Attributs bezeichnet: $\text{Type}_g^{Att}(a,t) =_{\text{def}} y$.

(3) Typisierung der Konstanten
Für implizite Konstanten wird der Typ des Attributs wie folgt festgelegt:
$\text{Type}_g^{Att}(a,t) =_{\text{def}} \text{qname}(t, a.Name)$.

(4) Typisierung der Kompositionen
$$\text{th}(x) =_{\text{def}} \begin{cases} \text{Type}_g(x.Name) & : \quad x \in ProdReference \\ \text{DType}_g^{TN}(x.Name) & : \quad x \in TypeReference \end{cases}$$
$\forall x \in a.Kind : x \in ProdReference \cup TypeReference \Rightarrow$
$\quad \exists y \in a.Kind : \forall x \in a.Kind : \text{th}(x) \succ_g^* \text{th}(y)$

Anmerkung: Für wohldefinierte Attribute wird $\text{th}(y)$ als Typ des Attributs bezeichnet: $\text{Type}_g^{Att}(a,t) =_{\text{def}} \text{t}(y)$.

(5) Kardinalitäten
$a.Min \leq a.Max$

Neben der Wohldefiniertheit der Datenstrukturen und ihrer Attribute, die eine einzelne Produktion definieren, sind zwei Eigenschaften entscheidend:

- Die AST-Regeln dürfen existierende Attribute nicht auf eine unzulässige Art und Weise ändern. Dieses stellt die Gültigkeit der Operation astoverride sicher. Die Kardinalitäten werden hier nicht betrachtet, weil die AST-Regeln sowohl einschränken dürfen, so dass nur die Modelle als valide erkannt werden, die die engere Auswhal erfüllt, als auch erweitern dürfen, um die abstrakte Syntax nutzbarer zu machen, wobei der Parser dann nur eine Teilmenge der gültigen Modelle erzeugen kann.
- Unterklassen, Unterschnittstellen und implementierende Klassen müssen den jeweiligen Obertyp respektieren. Dieses stellt die Gültigkeit der Operation override sicher.

Definition 5.4.22 (Gültigkeit der Operation astoverride)**.** Für zwei Datenstrukturen d_1 und d_2 innerhalb der Grammatiken g_1 bzw. g_1 für die Datentypen $t_1, t_2 \in JType$ ist die Operationen astoverride(d_1, d_2) gültig, falls d_1 und d_2 wohldefiniert sind und für alle Attribute $a_1 \in d_1$ und $a_2 \in d_2$ mit $a_1.Name = a_2.Name$ gilt:

(1) Dieselbe Art des Attributs
$\forall x \in a_1.Kind : x \in JType \Rightarrow \forall y \in a_2.Kind : y \in JType$ und
$\forall x \in a_1.Kind : x \in ConstantValue \Rightarrow \forall y \in a_2.Kind : y \in ConstantValue$ und
$\forall x \in a_1.Kind : x \in ProdReference \cup TypeReference \Rightarrow$
$\quad \forall y \in a_2.Kind : y \in ProdReference \cup TypeReference$

(2) Typisierung der Attribute: Verallgemeinerung durch AST-Regel erlaubt
$\forall x \in a_1.Kind \cup a_2.Kind : x \in JType \cup ProdReference \cup TypeReference \Rightarrow$
$\text{Type}_{g_2}^{Att}(a_2, t_2) \succ_g^* \text{Type}_{g_1}^{Att}(a_1, t_1)$

Neben der Wohldefiniertheit der Datenstrukturen, die eine einzelne Produktion definieren, ist es wichtig zu prüfen, ob die Attribute und Kompositionen auch die auftretenden Vererbungsbeziehungen respektieren. Dieses stellt die Gültigkeit der Operationen override sicher. Dabei gibt es verschiedene Variationspunkte, die von der Implementierung der Zielplattform abhängen.

Definition 5.4.23 (Gültigkeit der Operation override)**.** Für zwei Datenstrukturen d_1 und d_2 innerhalb der Grammatiken g_1 bzw. g_1 für die Datentypen $t_1, t_2 \in JType$ ist die Operation override(d_1, d_2) gültig, falls d_1 und d_2 wohldefiniert sind und für alle Attribute $a_1 \in d_1$ und $a_2 \in d_2$ mit $a_1.Name = a_2.Name$ gilt:

(1) Dieselbe Art des Attributs
$\forall x \in a_1.Kind : x \in JType \Rightarrow \forall y \in a_2.Kind : y \in JType$ und
$\forall x \in a_1.Kind : x \in ConstantValue \Rightarrow \forall y \in a_2.Kind : y \in ConstantValue$ und
$\forall x \in a_1.Kind : x \in ProdReference \cup TypeReference \Rightarrow$
$\forall y \in a_2.Kind : y \in ProdReference \cup TypeReference$

(2) Subtypkompatiblität der Attribute

Variationspunkt 2: Subtypisierung der Attribute:
Für eine gültige Operation override(d_1, d_2) muss eine der folgenden Eigenschaften für $a_1.Kind, a_2.Kind \in JType$ gelten:

(a) *Subtypisierung von Attributen*
$\text{Type}_{g_1}^{Att}(a_1, t_1) \succ_g^* \text{Type}_{g_2}^{Att}(a_2, t_2)$.
(*Anmerkung*: $\text{Type}_{g_1}^{Att}(a_1, t_1) \succ_g^* \text{Type}_{g_2}^{Att}(a_2, t_2)$ ist kein Schreibfehler, weil im Gegensatz zur Wohldefiniertheit von override hier kein Obertyp aller Attribute gesucht wird, sondern die Komposition verfeinert wird.)

(b) *Subtypisierung von Attributen mit Kardinalität 1*
Für $a_1.Max = a_2.Max = 1$ muss gelten $\text{Type}_{g_1}^{Att}(a_1, t_1) \succ_g^* \text{Type}_{g_2}^{Att}(a_2, t_2)$, sonst muss gelten $\text{Type}_{g_1}^{Att}(a_1, t_1) = \text{Type}_{g_2}^{Att}(a_2, t_2)$.

(c) *Keine Subtypisierung von Attributen*
$\text{Type}_{g_1}^{Att}(a_1, t_1) = \text{Type}_{g_2}^{Att}(a_2, t_2)$.

Variationspunkt 3: Kardinalität der Attribute:
Für eine gültige Operation override(d_1, d_2) muss eine der folgenden Eigenschaften für $a_1.Kind, a_2.Kind \in JType$ gelten:

(a) *Unbeachtet*
Keine Einschränkung.

(b) *Keine Mischung*
$a_1.Max > 1 \Rightarrow a_2.Max > 1, a_1.Max = 1 \Rightarrow a_2.Max = 1$

(c) *Genaue Übereinstimmung*
$a_1.Min = a_2.Min,\ a_1.Max = a_2.Max$

(3) Subtypkompatiblität der Konstanten

Variationspunkt 4: Subtypisierung der Konstanten:
Für eine gültige Operation override(d_1, d_2) muss eine der folgenden Eigenschaften für $a_1.Kind, a_2.Kind \in ConstantValue$ gelten:

(a) *Keine Typisierung von Konstanten*
Keine Einschränkung.

(b) *Keine Subtypisierung von Konstanten*
$\text{Type}_{g_1}^{Att}(a_1, t_1) = \text{Type}_{g_2}^{Att}(a_2, t_2)$.

(*Anmerkung:* Es wurde auf keiner Plattform eine wirkliche Untertypbildung für Konstanten identifiziert, so dass neben dem Verzicht auf Subtypisierung nur eine generische Realisierung über primitive Datentypen wie `boolean` und `int` möglich ist. Diese Form der Konstanten hat keine Einschränkung bezüglich der Typisierung der Konstanten.)

Variationspunkt 5: Kardinalität der Konstanten:
Für eine gültige Operation override(d_1, d_2) muss eine der folgenden Eigenschaften für $a_1.Kind, a_2.Kind \in ConstantValue$ gelten:

(a) *Unbeachtet*
Keine Einschränkung.

(b) *Keine Mischung*
$a_1.Max > 1 \Rightarrow a_2.Max > 1, a_1.Max = 1 \Rightarrow a_2.Max = 1$

(c) *Genaue Übereinstimmung*
$a_1.Min = a_2.Min,\ a_1.Max = a_2.Max$

(4) Subtypkompatiblität der Kompositionen

Variationspunkt 6: Subtypisierung der Kompositionen:
Für eine gültige Operation override(d_1, d_2) muss eine der folgenden Eigenschaften für $a_1.Kind, a_2.Kind \in ProdReference \cup TypeReference$ gelten:

(a) *Subtypisierung von Kompositionen*
$\text{Type}_{g_1}^{Att}(a_1, t_1) \succ_g^* \text{Type}_{g_2}^{Att}(a_2, t_2)$.

(b) *Subtypisierung von Kompositionen mit Kardinalität 1*
Für $a_1.Max = a_2.Max = 1$ muss gelten $\text{Type}_{g_1}^{Att}(a_1, t_1) \succ_g^* \text{Type}_{g_2}^{Att}(a_2, t_2)$, sonst muss gelten $\text{Type}_{g_1}^{Att}(a_1, t_1) = \text{Type}_{g_2}^{Att}(a_2, t_2)$.

(c) *Keine Subtypisierung von Kompositionen*
$\text{Type}_{g_1}^{Att}(a_1, t_1) = \text{Type}_{g_2}^{Att}(a_2, t_2)$.

Variationspunkt 7: Kardinalität der Kompositionen:
Für eine gültige Operation override(d_1, d_2) muss eine der folgenden Eigenschaften für $a_1.Kind, a_2.Kind \in ProdReference \cup TypeReference$ gelten:

(a) *Unbeachtet*
Keine Einschränkung.

(b) *Keine Mischung*
$a_1.Max > 1 \Rightarrow a_2.Max > 1, a_2.Max = 1 \Rightarrow a_2.Max = 1$

(c) *Genaue Übereinstimmung*
$a_1.Min = a_2.Min, a_1.Max = a_2.Max$

Definition 5.4.24 (Hilfsfunktion Prods_g)**.** Für einen Typ $t \in JType$ bezeichnet die Funktion $\text{Prods}_g : JType \to Grammar \times Prod$ die Grammatik-Produktions-Paare, die zur Definition eines Typs geführt haben.
$\text{Prods}_g(t) =_{\text{def}} \{(\gamma, \pi) \mid (\gamma, \pi) \in \text{range}(\text{TN}'_g),\ (\gamma, \pi) = \text{PN}'_\gamma(n), \text{DType}(n) = t\}$

Unter Nutzung der vorherigen Definitionen und Hilfsfunktionen lässt sich die Funktion att_g^T für Typen definieren. Diese dient dann zur Berechnung der Attribute eines Typs innerhalb der Ableitung der abstrakten Syntax. Hier werden nun die AST-Regeln miteinbezogen, wobei jeweils die Gültigkeit von override und astoverride genutzt wird, um die Gültigkeit von att_g^T zu prüfen. Diese Ableitung betrachtet zunächst alle Klassen und Schnittstellen unabhängig von ihren Oberklassen und implementierte Schnittstellen, weil diese die Ausgestaltung der Attribute nicht beeinflussen.

Definition 5.4.25 (Berechnung der Datenstruktur für Typen mit att_g^T)**.** Für einen Typ $t \in JType$, der aus einer Parserproduktion einer Grammatik g entstanden ist, gilt:

(1) Schnittstellenproduktionen
Für $\text{Prods}_g(t) = \{(\gamma, \pi)\}, \pi \in \gamma.InterfaceProd$ gilt:
($\text{Prods}_g(t)$ enthält aufgrund der Konstruktion von DType nur ein Element.)

(a) $\text{att}_g^T(t) =_{\text{def}} \text{astoverride}(\text{att}_g(\text{ast}_\gamma(\pi.IType.Name)), \text{att}_g(\pi))$

(b) $\text{att}_g^T(t)$ ist gültig, wenn $\text{att}_g(\text{ast}_\gamma(\pi.IType.Name)))$, $\text{att}_g(\pi)$ und $\text{astoverride}(\text{att}_g(\text{ast}_\gamma(\pi.IType.Name)), \text{att}_g(x.Prod))$ wohldefiniert sind und die Operation $\text{astoverride}(\text{att}_g(\text{ast}_\gamma(\pi.IType.Name)), \text{att}_g(x.Prod))$ gültig ist.

(2) Abstrakte Produktionen
Für $\text{Prods}_g(t) = \{(\gamma, \pi)\}, \pi \in \gamma.AbstractProd$
($\text{Prods}_g(t)$ enthält aufgrund der Konstruktion von DType nur ein Element.)

(a) $\text{att}_g^T(t) =_{\text{def}} \text{astoverride}(\text{att}_g(\text{ast}_\gamma(\pi.CType.Name)), \text{att}_g(\pi))$

(b) $\text{att}_g^T(t)$ ist gültig, wenn $\text{att}_g(\text{ast}_\gamma(\pi.CType.Name)))$, $\text{att}_g(\pi)$ und $\text{astoverride}(\text{att}_g(\text{ast}_\gamma(\pi.CType.Name)), \text{att}_g(x.Prod))$ wohldefiniert sind und die Operation $\text{astoverride}(\text{att}_g(\text{ast}_\gamma(\pi.CType.Name)), \text{att}_g(x.Prod))$ gültig ist.

(3) Klassenproduktionen
Für $\text{Prods}_g(t) = \{(\gamma_1, \pi_1), \ldots, (\gamma_m, \pi_m)\}, n = \pi_1.CType_1 = \ldots = \pi_m.CType_1$ gilt:
(n ist aufgrund der Konstruktion von DType eindeutig.)

(a) $\text{att}_g^T(t) =_{\text{def}} \text{astoverride}(\text{att}_g(\text{ast}_g(n)), \text{att}_g(\{\pi_1, \ldots, \pi_m\}))$

(b) $\text{att}_g^T(t)$ ist gültig, wenn $\text{att}_g(\text{ast}_g(n))$, $\text{att}_g(\{\pi_1, \ldots, \pi_m\})$ und $\text{astoverride}(\text{att}_g(\text{ast}_g(n)), \text{att}_g(\{\pi_1, \ldots, \pi_m\})))$ wohldefiniert sind und die Operation $\text{astoverride}(\text{att}_g(\text{ast}_g(n)), \text{att}_g(\{\pi_1, \ldots, \pi_m\})))$ gültig ist.

Definition 5.4.26 (Berechnung der transitiven Datenstruktur für Typen mit $\text{att}_g^{T'}$)**.** Die Funktion $\text{att}_g^{T'}$ bezeichnet die transitiven Attribute für die Typen, was die Attribute der Oberklassen mit einschließt.

(1) Schnittstellenproduktionen
Für eine Schnittstelle t und den Schnittstellentypen $\{i | t \succ_g i\} = \{i_1, \ldots, i_n\}$ gilt:

(a) $\text{att}_g^{T'}(t) =_{\text{def}} \text{override}(\text{att}_g(t), \text{alt}(\text{att}_g^{T'}(i_1), alt(\ldots \text{att}_g^{T'}(i_n))))$

(b) $\text{att}_g^{T'}(t)$ ist gültig, wenn $\text{att}_g(t)$ und $\text{att}_g^{T'}(i_1), \ldots \text{att}_g^{T'}(i_n)$ wohldefiniert und die Operation $\text{override}(\text{att}_g(t), \text{alt}(\text{att}_g^{T'}(i_1), alt(\ldots \text{att}_g^{T'}(i_n))))$ gültig ist.

(2) Abstrakte Produktionen und Klassenproduktionen
Für $\{i | t \succ_g i\} = \{i_1, \ldots i_n\}$ Schnittstellentypen und $\{c | t \succ_g c\} = \{c_1, \ldots c_m\}$ Klassentypen gilt:

(a) $\text{att}_g^{T'}(t) =_{\text{def}} \text{override}(att(t), \text{alt}(\text{att}_g^{T'}(c_1), ... \text{alt}(\text{att}_g^{T'}(c_m))))$

(b) $\text{att}_g^{T'}(t)$ ist gültig, wenn $\text{att}_g(t)$, $\text{att}_g^{T'}(i_1)$, ... $\text{att}_g^{T'}(i_n)$ und $\text{att}_g^{T'}(c_1)$, ... $\text{att}_g^{T'}(c_m)$ wohldefiniert sind. Zusätzlich müssen die Schnittstellen implementiert werden, also die Operation $override(\text{att}_g^{T'}(t), \text{att}_g^{T'}(i_1)), ..., \text{override}(\text{att}_g^{T'}(t), \text{att}_g^{T'}(i_n))$ gültig sein.

5.4.5 Assoziationen

Die Namen der Assoziationen einer Grammatik müssen ebenso wie die Produktionen eindeutig sein und bilden somit einen eigenen Namensraum.

Definition 5.4.27 (Eindeutigkeit von Assoziationsnamen)**.** Die Namen der Assoziationen einer Grammatik sind eindeutig, wenn $\text{AN}_g \subset \wp(Name \times Association)$ eine partielle Funktion ist $(\text{AN}_g : Name \to Association)$. $\text{AN}_g =_{\text{def}} \{ (a.Name, a) \mid a \in g.Association \}$

Definition 5.4.28 (Eindeutigkeit von Assoziationsnamen unter Vererbung)**.** Die Namen der Assoziationen einer Grammatik g mit den Obergrammatiken $g.Grammar = [g_1, ..., g_m]$ sind eindeutig unter Vererbung, wenn $\text{AN}'_g \subset \wp(Name \times Association$ eine partielle Funktion ist $(\text{AN}'_g : Name \to Association)$. $\text{AN}'_g =_{\text{def}} \text{AN}_g \cup \bigcup_{i \in \{1, ..., m\}} \text{AN}'_{g_i}$

Darüber hinaus sollen die Namen der Assoziationen nicht mit den Namen der Produktionen kollidieren, um Missverständisse zu vermeiden und den Zielplattformen mehr Freiheiten bei der Implementierung zu geben.

Definition 5.4.29 (Disjunktion von Assoziationsnamen und Produktionsnamen)**.** Die Namen der Assoziationen und Produktionen sind disjunkt, wenn gilt $\text{def}(\text{PN}'_g) \cap \text{def}(\text{AN}'_g) = \emptyset$.

Aus derselben Argumentation heraus müssen die Namen der navigierbaren Assoziationsenden nicht mit Attributen kollidieren. Da Assoziationsenden auch in Unterklassen zur Verfügung stehen, muss diese Eigenschaft auch bei vererbten Klassen geprüft werden.

Definition 5.4.30 (Relation AE_g)**.** Die Relation $AE_g \subset \wp(JType \times Name \times Association)$ ist für eine Grammatik g die kleinste Menge mit folgenden beiden Eigenschaften:

(1) $\forall t \in \text{range}(\text{DType}_g), a \in \text{range}(\text{AN}'_g)$:
$t \succ_g^* a.AssociationEnd_2.Type, a.Direction \in \{\leftarrow, \leftrightarrow\}$
$\Rightarrow (t, a.AssociationEnd_1.Name, a) \in AE_g$

(2) $\forall t \in \text{range}(\text{DType}_g), a \in \text{range}(\text{AN}'_g)$:
$t \succ_g^* a.AssociationEnd_1.Type, a.Direction \in \{\rightarrow, \leftrightarrow\}$
$\Rightarrow (t, a.AssociationEnd_2.Name, a) \in AE_g$

Definition 5.4.31 (Disjunktion von Attributen und Assoziationsnamen)**.** Die Namen der Attribute und Assoziationsenden einer Grammatik g sind disjunkt, wenn

(1) es keine zwei gleichnamigen navigierbaren Asssoziationsenden in einer Klasse gibt, also die Relation AE_g ein partielle Funktion $\text{AE}_g : JType \times Name \to Association$ ist, und

(2) die Navigationsenden unterschiedlich von den Attributen sind:
$\forall t \in JType : (n, T, min, max) \in \text{att}_g^{T'}(t) \Rightarrow \nexists (t, n, a') \in AE_g$.

5.4.6 Variablen

Neben den Attributen gibt es in der MontiCore-Grammatik auch Variablen, die Attributen sehr ähnlich sind, weil sie auch an keiner Stelle explizit definiert werden müssen, sondern sich ihr Typ automatisch aus den Zuweisungen ergibt. Variablennamen müssen sich dabei (auch wenn es keine technischen oder logischen Gründe gibt) von Attribut-, Kompositions- und Assoziationsendennamen unterscheiden, um Verwechselungen zu vermeiden. Im Gegensatz zu Attributen findet jedoch keine Vererbung von Variablen statt. Variablen sind vom Gültigkeitsbereich auf eine einzelne Produktion begrenzt und sind auch nicht bei mehrteiligen Klassenproduktionen in mehreren Produktionen gültig.

Zur Berechnung der Typisierung wird wie für Attribute die Datenstruktur D aus Definition 5.4.17 eingesetzt, wobei die Kardinalitäten keine Rolle spielen, sondern nur die Typisierung berechnet wird. Die Typableitung erfolgt durch die Funktion var, die sich von der Funktion att_g nur dadurch unterscheidet, dass bei Produktionselementen anstatt der Attribute die Variablenzuweisungen und das Anfügen von Werten betrachtet werden.

Definition 5.4.32 (Funktion var)**.** Die Funktion var dient zur Berechnung der Datenstruktur D aus den Produktionen einer Grammatik. Die Funktion ist dabei rekursiv definiert und verwendet die Operationen aus Definition 5.4.18.

(1) Klassenproduktionen
Für $p, p_1, \ldots p_n \in \mathit{ClassProd}$ gilt:

(a) $\text{var}(p) =_{\text{def}} \text{seq}(x, \text{var}(p.\mathit{Body}))$, wobei $x = \mathit{seq}(\text{var}(l_1), \mathit{seq}(..., \text{var}(l_n))...)$ für $p.\mathit{Parameter} = [l_1, \ldots l_n] \in \mathit{List}\,\langle \mathit{Parameter} \rangle$

(b) $\text{var}(\{p_1, \ldots p_n\}) =_{\text{def}} \text{alt}(\text{var}(p_1), \text{alt}(\ldots \text{var}(p_n)))$

(2) Formale Parameter
Für $p \in \mathit{Parameter}$ gilt:

(a) Für $p.\mathit{ProdElementType} = \text{VarAssign}$ gilt:
$\text{var}(p) =_{\text{def}} \{(p.\mathit{Name}_1, \{p.\mathit{Name}_2\}, p.\mathit{Card}.\mathit{Min}, p.\mathit{Card}.\mathit{Max})\}$

(b) Für $p.\mathit{ProdElementType} \in \{\text{AST}, \text{Ignore}\}$ gilt:
$\text{var}(p) =_{\text{def}} \emptyset$

(3) Struktur der Terme
Für $t, t_1, t_2 \in \mathcal{T}_{\Sigma_G}(\mathit{ProdElement} \cup \mathit{Action})$ gilt:

(a) $\text{var}(t_1|t_2) =_{\text{def}} \text{alt}(\text{var}(t_1), \text{var}(t_2))$

(b) $\text{var}(t_1 t_2) =_{\text{def}} \text{seq}(\text{var}(t_1), \text{var}(t_2))$

(c) $\text{var}(t^*) =_{\text{def}} \text{kleene}(\text{var}(t))$

(4) ProdElements
Für $x \in \mathit{ProdElement}$ gilt:

(a) Für $x.\mathit{ProdElementType} = \text{VarAssign}$ gilt:

i. $c = x.\mathit{ProdElementRef} \in \mathit{Constant}$:
$\text{var}(x) =_{\text{def}} \{(x.\mathit{Name}, \{z\}, , \text{VarAppend1}, 1)\}$ mit
$z = c.\mathit{Name} \in \mathit{ConstantValue}$

ii. $n = x.\mathit{ProdElementRef} \in \mathit{NonTerminal}$:
$\text{var}(x) =_{\text{def}} \{(x.\mathit{Name}, \{z\}, 1, 1)\}$ mit
$z = n.\mathit{Name} \in \mathit{ProdReference}$

iii. $t = x.ProdElementRef \in Terminal$:
$\text{var}(x) =_{\text{def}} \{(x.Name, \{z\}, 1, 1)\}$ mit
$z = \texttt{java.lang.String} \in JType$

(b) Für $x.ProdElementType \in \{\text{AST}, \text{Ignore}\}$ gilt:
$\text{var}(x) =_{\text{def}} \emptyset$

(5) Aktionen
Für $x \in Action$ gilt:
$\text{var}(x) =_{\text{def}} \emptyset$

Die Variablen sind wohldefiniert, wenn die Namen der Variablen nicht mit anderen Namen innerhalb einer Produktion kollidieren. Variablenzuweisungen sind nur für Nichtterminale und Terminale erlaubt.

Definition 5.4.33 (Wohldefiniertheit von Variablen)**.** Die Variablen einer Grammatik g sind wohldefiniert, wenn gilt:

(1) Variablenzuweisungen sind entweder nur für Nichtterminale oder nur für Terminale erlaubt.
$\forall p \in \text{range}(\text{PN}'_g).Prod \cap g.ClassProd$:
$\text{var}(p).Kind \in \wp(ProdReference \cup TypeReference) \cup \wp(JType)$

(2) Typisierung der Variablen aus Terminalen und lexikalischen Produktionen.
$\forall p \in \text{range}(\text{PN}'_g).Prod \cap g.ClassProd$:
$\forall x \in var(p).Kind : x \in JType \Rightarrow \exists y \in \text{var}(p).Kind : \forall x \in a.Kind : x \succ_g^* y$

Anmerkung: Für wohldefinierte Variablen wird y als Typ der Variablen bezeichnet: $\text{Type}_g^{Var}(x.Name, p) =_{\text{def}} y$.

(3) Typisierung der Variablen aus Kompositionen.
$\forall p \in \text{range}(\text{PN}'_g).Prod \cap g.ClassProd$:
$\forall x \in \text{var}(p).Kind : x \in ProdReference \cup TypeReference \Rightarrow$
$\exists y \in a.Kind : \forall x \in d.Kind : \text{th}(x) \succ_g^* \text{th}(y)$

wobei

$$\text{th}(x) =_{\text{def}} \begin{cases} \text{Type}_g(x.Name) & : \quad x \in ProdReference \\ \text{DType}_g^{TN}(x.Name) & : \quad x \in TypeReference \end{cases}$$

Anmerkung: Für wohldefinierte Variablen wird $\text{th}(y)$ als Typ der Variablen bezeichnet: $\text{Type}_g^{Var}(x.Name, p) =_{\text{def}} \text{th}(y)$.

(4) Variablennamen müssen sich von Attributen und Assoziationsenden unterscheiden.
$\forall p \in \text{range}(\text{PN}'_g).Prod \cap ClassProd \Rightarrow$

(a) $\forall \delta_1 \in \text{att}_g^{T'}(\text{DType}_g(p.Name)), \delta_2 \in \text{var}(p) : \delta_1.Name \neq \delta_2.Name$
(b) $\forall \delta \in \text{var}(p) : \nexists(\text{DType}_g(p.Name), \delta.Name) \in \text{def}(AE_g)$

Aufbauend auf den vorherigen Definitionen muss nun für eine wohlgeformte Grammatik gelten, dass bei jeder Verwendung von Nichtterminalen die Typisierung und die Kardinalität von Produktionsparametern eingehalten wurde. Das bedeutet, dass bei jeder Verwendung eines Nichtterminals für die als Parameter verwendeten Variablen geprüft wird, welche minimale und welche maximale Belegung aus der Struktur der Produktion an dieser Stelle gefolgert werden kann. Diese Kardinalität muss innerhalb der definierten Grenzen des formalen Parameters liegen. Zur Berechung wird zunächst die Datenstruktur *CurVar* und die dazu passenden Operationen definiert.

Definition 5.4.34 (Datenstruktur *CurVar*)**.** Die aktuelle Variablenbelegung bezeichnet die minimale und maximale Elementanzahl an einer bestimmten Stelle innerhalb einer Produktion $CurVar = \wp(Name \times \mathbb{N} \times \mathbb{N} \cup \{*\})$. Für *CurVar* sind die folgenden Operationen definiert.

(1) Minimum
$\min\colon CurVar \times Name \to \mathbb{N}$ mit
$$\min(v,n) =_{\text{def}} \begin{cases} i & : \ \exists (n,i,a) \in v \\ 0 & : \ \text{sonst} \end{cases}$$

(2) Maximum
$\max\colon CurVar \times Name \to \mathbb{N} \cup \{*\}$ mit
$$\max(v,n) =_{\text{def}} \begin{cases} a & : \ \exists (n,i,a) \in v \\ 0 & : \ \text{sonst} \end{cases}$$

(3) Kleenescher Stern
$\text{kleene}\colon CurVar \times CurVar \times CurVar \to CurVar$
$kleene(c_1,c_2,c_3) = \{(n,i,a) \mid n \in c_1.Name \cup c_2.Name \cup c_3.Name\}$ mit
$$i =_{\text{def}} \begin{cases} \min(\min(c_1,n),\min(c_2,n)) & : \ \min(c_2,n) \leq \min(c_3,n) \\ 0 & : \ \text{sonst} \end{cases}$$
$$a =_{\text{def}} \begin{cases} \max(\max(c_1,n),\max(c_2,n)) & : \ \max(c_2,n) \geq \max(c_3,n) \\ * & : \ \text{sonst} \end{cases}$$

(4) Alternativen
$\text{alt}\colon CurVar \times CurVar \to CurVar : c_1 \times c_2 \mapsto c$ mit
$$\begin{aligned} c \ = \ & \{(n,0,max) \mid (n,min,max) \in c_1, \nexists (n,min',max') \in c_2\} \cup \\ & \{(n,0,max) \mid (n,min,max) \in c_2, \nexists (n,K',min',max') \in d_1\} \cup \\ & \{(n,min,max) \mid (n,min_1,max_1) \in c_1, (n,min_2,max_2) \in c_2, \\ & \qquad min = \min(min_1,min_2), max = \max(max_1,max_2)\} \end{aligned}$$

(5) Sequenzen
$\text{seq}\colon CurVar \times CurVar \to CurVar : c_1 \times c_2 \mapsto c$ mit
$$\begin{aligned} c \ = \ & \{(n,min,max) \mid (n,min,max) \in c_1, \nexists (n,min',max') \in c_2\} \cup \\ & \{(n,min,max) \mid (n,min,max) \in c_2, \nexists (n,min',max') \in c_1\} \cup \\ & \{(n,min_1+min_2,max_1+max_2) \mid (n,min_1,max_1) \in c_1, (n,min_2,max_2) \in c_2\} \end{aligned}$$

(6) Override
$\text{seq}\colon CurVar \times CurVar \times Name \to CurVar : c_1 \times c_2 \times n \mapsto c$ mit
$$\begin{aligned} c \ = \ & \{(n',min,max) \mid (n',min,max) \in c_1, \nexists (n,min',max') \in c_2, n \neq n'\} \cup \\ & \{(n',min,max) \mid (n',min,max) \in c_2, \nexists (n',min',max') \in c_1 \text{ oder } n' = n\} \cup \\ & \{(n',min_1+min_2,max_1+max_2) \mid (n',min_1,max_1) \in c_1, (n',min_2,max_2) \in c_2\} \end{aligned}$$

Die Funktion vars dient zur Berechnung der Datenstruktur aus den Produktionen einer Grammatik. Dabei wird auch die Relation aller Variablenvorkommen innerhalb der Produktionen als Teilmenge von $\wp(NonTerminal \times CurVar)$ berechnet. Diese Teilmenge bezeichnet die Menge der Nichtterminale mit an dieser Stelle gültigen minimalen und maximalen Anzahl an Objekten in den jeweiligen Variablen. Die Funktion vars ist rekursiv

definiert und verwendet die Operationen aus der Definition 5.4.34. Dabei ist zu beachten, dass jede Verwendung einer Variablen als Regelparameter, diese Variablen zurücksetzt. Dieses Verhalten ist notwendig, um sicherzustellen, dass jeder Knoten nur einmal innerhalb des ASTs vorkommt und die Variablen nicht zur Duplizierung von Knoten genutzt werden.

Definition 5.4.35 (Funktion vars)**.** Die Funktion vars : $CurVar \times \wp(NonTerminal \times CurVar) \times \top \rightarrow CurVar \times \wp(NonTerminal \times CurVar)$ dient zur Berechnung der Kardinalität der Variablen innerhalb der einzelnen Stellen der Grammatik.

(1) Klassenproduktionen
Für $p \in ClassProd, p.ProdElement = [l_1, \ldots, l_m]$ gilt:
$\text{vars}(v, i, p) =_{\text{def}} \text{vars}(x, p.Body)$, wobei $x = \text{vars}(\ldots \text{vars}(\text{vars}(i, l_1), l_2) \ldots l_m)$

(2) Formale Parameter
Für $p \in Parameter, x = p.ProdElement$ gilt:

(a) Für $x.ProdElementType = \text{VarAssign}$ gilt:
$\text{vars}(v, i, x) =_{\text{def}} (\text{override}(v, \{(x.Name, p.Card.Min, p.Card.Max)\}, x.Name), i)$

(b) Für $x.ProdElementType \in \{\text{AST}, \text{Ignore}\}$ gilt:
$\text{vars}(v, i, x) =_{\text{def}} (v, i)$

(3) Struktur der Terme
Für $t, t_1, t_2 \in \mathcal{T}_{\Sigma_G}(ProdElement)$ gilt:

(a) $\text{vars}(v, i, t_1|t_2) =_{\text{def}} (\text{alt}(v_1, v_2), i_1 \cup i_2)$ wobei
$(v_1, i_1) = \text{vars}(i, t_1)$
$(v_2, i_2) = \text{vars}(i, t_2)$

(b) $\text{vars}(v, i, t_1 t_2) =_{\text{def}} \text{vars}(\text{vars}(i, t_1))$

(c) $\text{var}(v, i, t^*) =_{\text{def}} (\text{kleene}(i, \text{vars}(\emptyset, t), \text{vars}(\text{vars}(\emptyset, t), t), i)$

(4) ProdElement
Für $x \in ProdElement, (v', i') = \text{vars}(v, i, x.ProdElementRef)$ gilt:

(a) Für $x \in ProdElement,\ x.ProdElementType = \text{VarAssign}$ gilt:
$\text{vars}(v, i, x) =_{\text{def}} (\text{override}(v', \{(x.Name, 1, 1)\}, x.Name), i')$

(b) Für $x \in ProdElement,\ x.ProdElementType \in \{\text{AST}, \text{Ignore}\}$ gilt:
$\text{vars}(v, i, x) =_{\text{def}} (v', i')$

(5) ProdElementRef
Für $x \in ProdElementRef$ gilt:

(a) Für $x \in NonTerminal, x.Name_2 = [n_1, \ldots n_n]$ gilt:
$\text{vars}(v, i, x) =_{\text{def}} (\text{override}(\ldots \text{override}(v, \{(n_1, 0, 0)\}, n_1), (n_2, 0, 0) \ldots n_n), i \cup (x, v))$

(b) Für $x \in Constant \cup Terminal$ gilt: $\text{vars}(v, i, x) =_{\text{def}} (v, i)$

(6) Actions
Für $x \in Action$ gilt:

(a) Für $x \in Constr, x.Stmt = [s_1, \ldots s_n] gilt :$
$\text{vars}(v, i, x) =_{\text{def}} \text{vars}(\ldots \text{vars}(v, i, s_1), \ldots s_n)$

(b) Für $x \in Push \cup Pop$ gilt:
$\text{vars}(v, i, x) =_{\text{def}} (v, i)$

(7) Statements
Für $x \in Action$ gilt:
$\text{vars}(v, i, x) =_{\text{def}} (\text{override}(v, \{(x.Name_2, 0, 0)\}, x.Name_2), i)$

Mittels der Funktion var kann nun die Menge CV berechnet werden, die die Menge der Nichtterminale mit an dieser Stelle gültigen minimalen und maximalen Anzahl an Objekten in den jeweiligen Variablen bezeichnet. Diese ergibt sich aus der Menge I, die für alle Produktionen berechnet wird.

Definition 5.4.36 (Paare von Nichtterminalen und aktuell gültigen Variablenbelegungen)**.** Die Funktion $CV_g : ClassProd \rightarrow \wp(NonTerminal \times CurVar)$ bezeichnet alle Paare von Nichtterminalen und aktuell gültigen Variablenbelegungen einer Klassenproduktion in einer Grammatik g.
$CV_g(p) =_{\text{def}} \text{vars}(\emptyset, \emptyset, p).CurVar$

Eine Grammatik g verfügt nur über valide Nichtterminale, wenn alle Nichtterminale bei der Verwendung von Produktionsparametern die Typisierung einhalten, indem nur Variablen verwendet werden, die ein Subtyp der formalen Parameter sind, und die Variablen zu diesem Zeitpunkt innerhalb des vorgegebenen Bereichs sind. Als ein Spezialfall kann anstatt eines Variablennamens auch das reservierte Wort „null“ verwendet werden, wobei dann der Parameter auch die Kardinalität 0 erlauben muss.

Definition 5.4.37 (Valide Nichtterminale)**.** Eine Grammatik g verfügt über valide Nichtterminale, wenn gilt:
$\forall p \in \{x | \text{range}(PN'_g).Prod, x \in ClassProd\}$:
$\forall (n, c) \in CV_g(p)$ mit
$n.VariableName = [n_1, ..., n_m]$
$\text{PN}'_g(n.Name).Prod.Parameter = [p_1, ..., p_m]$:
$\forall i \in \{1, \ldots, m\}$ gilt eine der folgenden Bedingungen:

1. *Normale Variable*
   (a) $\text{Type}_g^{Var}(n_i, p) \succ_g^* \text{Type}_g^{Pr}(p_i)$
   (b) $\min(c, n_i) \geq p_i.Card.Min$
   (c) $\max(c, n_i) \leq p_i.Card.Max$
2. *Speziallfall „null“*
   (a) $n_i = \text{null}$
   (b) $p_i.Card.Min = 0$

5.5 UML/P-Klassendiagramme

Als semantische Domäne zur Erklärung der Ableitung der abstrakten Syntax aus dem MontiCore-Grammatikformat wird eine Variante der UML/P-Klassendiagramme [Rum04a, Rum04b] verwendet. Die Änderungen im Vergleich zur Formalisierung in [CGR08] sind die folgenden:

- Auslassung der Modifier außer *composition* und *abstract*. Alle Klassen und Schnittstellen werden als öffentlich angesehen.
- Auslassung von Methoden, Konstruktoren, formalen Parametern, qualifizierten Assoziationen, Invarianten und Body, weil diese nicht benötigt werden.
- Erweiterung des Kardinalitätsbegriffs zur Darstellung beliebiger Bereiche.
- Umbenennung von *LeftPart* und *RightPart* in *AssocEnd*
- Umbenennung von *Type* in *CDType*
- Verwendung von Elementen der Grundlagendefinitionen aus Abbildung 5.1.
- Einführung von Enum-Klassen, die eine Menge von validen Werten zusammenfassen.
- Einführung von Kardinalitäten für Attribute.

CD	$=$	$DiagramName \times \wp(Class) \times \wp(Interface) \times \wp(Assoc) \times \wp(Enum)$
$Class$	$=$	$\wp(Modifier) \times ClassName \times \wp(SuperClassName) \times \wp(InterfaceName) \times \wp(Attr)$
$Interface$	$=$	$InterfaceName \times \wp(SuperInterfaceName) \times \wp(Attr)$
$Enum$	$=$	$EnumName \times \wp(EValue)$
$Assoc$	$=$	$\wp(Modifier) \times AssocName^{\text{opt}} \times AssocEnd \times Direction_{5.1} \times AssocEnd$
$AssocEnd$	$=$	$\wp(Modifier) \times ClassName \times Role^{\text{opt}} \times Card_{5.1}^{\text{opt}}$
$CDType$	$=$	$ClassName \cup BasicType \cup InterfaceName \cup EnumName$
$Attr$	$=$	$AttrName \times CDType \times Card_{5.1}^{\text{opt}}$
$Modifier$	$=$	$\{\text{composition}, \text{abstract}\}$
$SuperInterfaceName$, $SuperClassName$, $ClassName$, $AssocName$, $DiagramName$, $Role$, $InterfaceName$, $AttrName$, $EnumName$,		
$EValue$	$=$	$Name_{5.1}$
$BasicType$	$=$	$\ll$Basistypen$\gg$

Abbildung 5.4: Vereinfachte abstrakte Syntax von UML/P-Klassendiagrammen

5.6 Semantische Abbildung der abstrakten Syntax

Für Grammatiken, die alle Kontextbedingungen aus Abschnitt 5.3 erfüllen, kann die abstrakte Syntax der definierten Sprache abgeleitet werden. Dabei werden die einzelnen Grammatiken unabhängig von ihren Obergrammatiken in ein UML/P-Klassendiagramm überführt. Für ein vollständiges Verständnis ist dann die Semantik aller transitiven Obergrammatiken notwendig, wobei durch die Konstruktion der semantischen Abbildung sichergestellt ist, dass jede Klasse und Schnittstelle nur einmal definiert wird.

Für die Formulierung der semantische Abbildung werden zunächst zwei Hilfsfunktionen definiert, die die Datenstruktur D in die entsprechenden Attribute und Kompositionen überführt. Attribute aus D werden direkt in Attribute der entsprechenden Klassen überführt, wohingegen die Kompositionen als spezielle Assoziationen dargestellt werden. Diese Assoziationen bleiben unbenannt und nur das Assoziationsende ist entsprechend des Attributnamens aus D benannt.

Definition 5.6.1 (Abbildung der Attribute durch DToAttr)**.** Die Abbildung DToAttr bildet eine wohldefinierte Datenstruktur eines Typs t einer Grammatik g auf die entsprechenden Attribute ab. $\text{DToAttr} : D \times \mathit{Grammar} \times \mathit{JType} \to \wp(\mathit{Attr})$
$\text{DToAttr}(d, g, t) =_{\text{def}}$
$\{(\alpha.\mathit{Name}, \text{Type}_g^{\mathit{Att}}(\alpha, t), (\alpha.\mathit{Min}, \alpha.\mathit{Max})) \mid \alpha \in d, \alpha.\mathit{Kind} \in \mathit{JType} \cup \mathit{ConstantValue}\}$

Definition 5.6.2 (Abbildung der Kompositionen durch DToComp)**.** DToComp bildet eine wohldefinierte Datenstruktur eines Typs t einer Grammatik g auf die entsprechenden Kompositionen ab. $\text{DToComp} : D \times \mathit{Grammar} \times \mathit{JType} \to \wp(\mathit{Assoc})$:
$\text{DToComp}(d, g, t) =_{\text{def}} \{(\emptyset, \epsilon, e_1, \to, e_2) \mid \alpha \in d, \alpha.\mathit{Kind} \in \mathit{ProdReference} \cup \mathit{TypeReference}\}$
$e_1 =_{\text{def}} (\{\mathit{composition}\}, t, \epsilon, 0..1)$
$e_2 =_{\text{def}} (\emptyset, \text{Type}_g^{\mathit{Att}}(\alpha, t), \alpha.\mathit{Name}, (\alpha.\mathit{Min}, \alpha.\mathit{Max}))$

Definition 5.6.3 (Semantikabbildung)**.** Eine Grammatik g, die die Kontextbedingungen aus Abschnitt 5.3 erfüllt, kann wie folgt auf ein Klassendiagramm *cd* abgebildet werden: $[\![.]\!]_{as} : \mathit{Grammar} \to \mathit{CD} : g \mapsto (\text{qname}(g.\mathit{QName}, g.\mathit{Name}), C, I, A, E)$. Die einzelnen Komponenten ergeben sich wie folgt:

(1) Die Klassen ergeben sich aus den Klassenproduktionen und den abstrakten Produktionen der Grammatik. Die implementierten Schnittstellen und Oberklassen werden direkt in den Parserproduktionen oder aber in den AST-Regeln definiert. Die Attribute für die abstrakten Produktionen ergeben sich nur aus den AST-Regeln.

$$
\begin{array}{rcl}
C & =_{\text{def}} & C_P \cup C_A \\
C_P & =_{\text{def}} & \{(\emptyset, \text{DType}_g(c.\mathit{Name}), SC_c, SI_c, Att_c) \mid c \in g.\mathit{ClassProd}.\mathit{CType}_1\} \\
SC_c & =_{\text{def}} & \{\text{DType}_g(x.\mathit{IType}.\mathit{Name}) \mid x \in Pd_c.\mathit{Inheritance}, x \in \mathit{AbstractProd}\} \cup \\
 & & \{\text{DType}_g(x.\mathit{Name}) \mid x \in Pd_c.\mathit{Inheritance}, x \in \mathit{ClassProd}\} \cup \\
 & & \{\text{DType}_g(x.\mathit{Name}) \mid x \in \text{ast}_g(c.\mathit{Name}).\mathit{CType}\} \\
SI_c & =_{\text{def}} & \{\text{DType}_g(x.\mathit{IType}.\mathit{Name}) \mid x \in Pd_c.\mathit{InterfaceProd}\} \cup \\
 & & \{\text{DType}_g(x) \mid x \in \text{ast}_g(c.\mathit{Name}).\mathit{IType}\} \\
Pd_c & =_{\text{def}} & \{x \mid x \in g.\mathit{ClassProd}, x.\mathit{CType}_1 = c\} \\
Att_c & =_{\text{def}} & \mathit{DToAttr}(\text{att}_g(\text{DType}_g(c.\mathit{Name})), g, \text{DType}_g(c.\mathit{Name}))
\end{array}
$$

$$
\begin{aligned}
C_A &=_{\text{def}} \{(\{\text{abstract}\}, \text{DType}_g(a.CType), SC_a, SI_a, Att_a) \mid a \in g.AbstractProd\} \\
SC_a &=_{\text{def}} \{\text{DType}_g(x.IType.Name) \mid x \in a.Inheritance, x \in AbstractProd\} \cup \\
&\quad \{\text{DType}_g(x.Name) \mid x \in \text{ast}_g(a.CType.Name).CType\} \\
SI_a &=_{\text{def}} \{\text{DType}_g(x.IType.Name) \mid x \in a.InterfaceProd\} \cup \\
&\quad \{\text{DType}_g(x.Name) \mid x \in \text{ast}_g(a.CType.Name).IType\} \\
Att_a &=_{\text{def}} DToAttr(\text{att}_g(\text{DType}_g(c.Name)), g, \text{DType}_g(c.Name))
\end{aligned}
$$

(2) Die Schnittstellen ergeben sich aus den Schnittstellenproduktionen der Grammatik. Die Oberschnittstellen können direkt in den Parserproduktionen oder aber in den AST-Regeln festgelegt werden. Die Attribute ergeben sich aus den AST-Regeln.

$$
\begin{aligned}
I &=_{\text{def}} \{(\text{DType}_g(iName), SI_i, Att_i) \mid i \in g.InterfaceProd\} \\
SI_i &=_{\text{def}} \{\text{DType}_g(x.IType.Name) \mid x \in i.InterfaceProd\} \cup \\
&\quad \{\text{DType}_g(x.Name) \mid x \in ast_g(n).IType\} \\
Att_i &=_{\text{def}} DToAttr(\text{att}_g(\text{DType}_g(i.Name)), g, \text{DType}_g(i.Name))
\end{aligned}
$$

(3) Die Assoziationen ergeben sich aus den Kompositionen zwischen den Klassen (A_P), abstrakten Klassen (A_A), Schnittstellen (A_I) und den nicht-kompositionalen Assoziationen (A_{Ass}).

$$
\begin{aligned}
A &=_{\text{def}} A_P \cup A_A \cup A_I \cup A_{Ass} \\
A_P &=_{\text{def}} \bigcup_{c \in g.ClassProd.CType_1} \\
&\qquad DToComp(override(\text{att}_g(\text{ast}_g(c)), \text{att}_g(Prods_c)), g, \text{DType}_g(c)) \\
Prods_c &=_{\text{def}} \{x \mid x \in g.ClassProd, x.CType_1 = c\} \\
A_A &=_{\text{def}} \bigcup_{r \in g.AbstractProd} DToComp(\text{att}_g(\text{ast}_g(r.CType)), g, \text{DType}_g(r)) \\
A_I &=_{\text{def}} \bigcup_{r \in g.InterfaceProd} DToComp(\text{att}_g(ast(g, r.IType)), g, \text{DType}_g(r)) \\
A_{Ass} &=_{\text{def}} \bigcup_{a \in g.Association} (\emptyset, a.Name, E_1, a.Direction, E_2) \\
E_i &=_{\text{def}} (\emptyset, \text{DType}_g^{TN}(a.AssociationEnd_i.Type), a.AssociationEnd_i.Name, \\
&\quad a.AssociationEnd_i.Card)
\end{aligned}
$$

(4) Die Enumerationen ergeben sich aus den Enumerationsproduktionen der Grammatik und den impliziten Konstanten in den einzelnen Klassen. Die für implizite Konstanten verwendete Typabbildung Type_g^{Att} ist in Definition 5.4.20 erklärt.

$$
\begin{aligned}
E &=_{\text{def}} E' \cup E'' \\
E' &=_{\text{def}} \bigcup_{e \in g.EnumProd} \{(\text{DType}_g(e.Name), e.Constant.Name)\} \\
E'' &=_{\text{def}} \bigcup_{c \in g.ClassProd} \bigcup_{a \in \text{att}_g(\text{DType}_g(c.Name))} \\
&\qquad \{(\text{Type}_g^{Att}(a, \text{DType}_g(c.Name)), a.Kind) \mid a.Kind \subset \wp(ConstantValue)\}
\end{aligned}
$$

5.7 Zielplattform

In Abhängigkeit von der Zielplattform wird das in Abschnitt 5.6 definierte Klassendiagramm in eine Implementierung umgesetzt. Dabei kann zwischen verschiedenen voneinander unabhängigen Varianten gewählt werden, die in den Definitionen 5.4.16 und 5.4.23 erklärt wurden und hier noch einmal aufgelistet werden:

Variationspunkt 1: Mehrfachvererbung in $\succ_g^*$

(a) Mehrfachvererbung zwischen Klassentypen erlaubt

(b) Mehrfachvererbung zwischen Klassentypen nicht erlaubt

Variationspunkt 2: Subtypisierung der Attribute

(a) Subtypisierung von Attributen

(b) Subtypisierung von Attributen mit Kardinalität 1

(c) Keine Subtypisierung von Attributen

Variationspunkt 3: Kardinalität der Attribute

(a) Unbeachtet

(b) Keine Mischung

(c) Genaue Übereinstimmung

Variationspunkt 4: Subtypisierung der Konstanten

(a) Keine Typisierung von Konstanten

(b) Keine Subtypisierung von Konstanten

Variationspunkt 5: Kardinalität der Konstanten

(a) Unbeachtet

(b) Keine Mischung

(c) Genaue Übereinstimmung

Variationspunkt 6: Subtypisierung der Kompositionen

(a) Subtypisierung von Kompositionen

(b) Subtypisierung von Kompositionen mit Kardinalität 1

(c) Keine Subtypisierung von Kompositionen

Variationspunkt 7: Kardinalität der Kompositionen

(a) Unbeachtet

(b) Keine Mischung

(c) Genaue Übereinstimmung

Neben der Auswahl der verschiedenen Varianten, die die Prüfung der Kontextbedingungen beeinflussen, nicht aber die semantische Abbildung an sich, haben die einzelnen Plattformen Freiheiten bei der Umsetzung des Klassendiagramms in Quellcode:

- Die Namen der einzelnen Schnittstellen und Klassen können modifiziert werden. Beispielsweise können die Klassen mit einem Namensprefix versehen werden und in einem Unterpaket abgelegt werden.

- Die Kompositionen und Assoziationen können als Attribute der beteiligten Klassen oder als eigenständige Klassen implementiert werden.
- Der Kardinalität der Attribute und Kompositionen kann insofern Rechnung getragen werden, dass bei mehrwertigen Attributen und Kompositionen entsprechende Felder, Mengen- oder Listenimplementierungen gewählt werden.
- Enumerationen und entsprechende Attribute können auf vielfältige Art und Weise umgesetzt werden. Neben der Verwendung von Enumerationen der Implementierungssprache sind auch die generische Umsetzung in einer zentralen Konstantendatei und die Verwendung von primitiven Datentypen wie `boolean` und `int` für die Attribute möglich.

5.8 Abstrakte Syntax der Sprachdatei

Analog zum Vorgehen für das MontiCore-Grammatikformat wurde die Sprachendateien-Grammatik in Abbildung 5.5 formalisiert. Zur Definition der abstrakten Syntax wurden zunächst Vereinfachungen an der Sprachdefinition durchgeführt. Die hier dargestellte Variante der Sprachdateien ist die Essenz der Sprache, die eine kompakte Semantikdefinition ermöglicht. In Anhang D.2 werden die Sprachdateien und die verwendete Essenz als MontiCore-Grammatik dargestellt. Dabei wird in Abschnitt D.2.3 detailliert dargestellt, welche Elemente ausgelassen wurden. Die wichtigsten Änderungen sind dabei die folgenden:

- Reduktion der Sprache auf die Elemente, die sich mit der Kombination von Fragmenten befassen
- Direkte Verweise auf Fragmente und Sprachen und nicht deren Namen
- Kompaktere abstraktere Syntax

$Language$	$=$	$QName_{5.1} \times Name_{5.1} \times$ $\wp(FragmentProduction) \times \wp(SubLanguage)$
$Element$	$=$	$FragmentProduction \cup SubLanguage$
$FragmentProduction$	$=$	$Grammar_{5.3} \times ProdName \times ElementName \times Start \times$ $\wp(Emb) \times \wp(EmbParameter)$
$SubLanguage$	$=$	$Language \times ElementName \times Start \times$ $\wp(Emb) \times \wp(EmbParameter)$
$Start$	$=$	$\{true, false\}$
Emb	$=$	$ElementName \times ProdName$
$EmbParameter$	$=$	$ElementName \times ProdName \times String_{5.1}$
$ElementName$, $ProdName$	$=$	$Name_{5.1}$

Abbildung 5.5: Vereinfachte abstrakte Syntax einer Sprachdatei

5.9 Kontextbedingungen für Sprachdateien

Die folgende semantische Abbildung für Sprachdateien lässt sich nur für wohldefinierte Sprachdateien angeben. Eine Sprachdatei ist wohldefiniert, wenn folgende Kontextbedingungen erfüllt sind:

1. Die Elementnamen einer Sprachdatei und aller darin enthaltenen Subsprachen sind eindeutig (vgl. Definition 5.10.1).
2. Eine Sprache darf nur eine Startproduktion besitzen (vgl. Definition 5.10.2).
3. Die Fragment-Produktions-Paare einer Sprachdatei müssen wohldefiniert sein (vgl. Definition 5.10.4).
4. Die Einbettungen (mit Parameter) einer Sprachdatei müssen wohldefiniert sein (vgl. Definition 5.10.5).

5.10 Hilfsstrukturen für Sprachdateien

Analog zur Spezifikation von Hilfsdatenstrukturen für das MontiCore-Grammatikformat sind Hilfsdatenstrukturen für die Sprachdateien notwendig, die eine Kombination mehrerer Sprachen und Fragmente zu komplexeren Sprachen erlauben. Die innerhalb einer Sprachdatei verwendeten Namen müssen eindeutig sein.

Definition 5.10.1 (Eindeutigkeit von Elementnamen)**.** Die Namen der Elemente einer Sprachdatei l sind eindeutig, wenn $\mathrm{EN}_l \subset \wp(\mathit{Name} \times \mathit{Element})$ eine partielle Funktion ist $(\mathrm{EN}_l : \mathit{Name} \to \mathit{Element})$.

$$\mathrm{EN}_l \quad =_{\mathrm{def}} \quad \{\ (e.\mathit{ElementName}, e) \mid e \in l.\mathit{FragmentProduction} \cup l.\mathit{SubLanguage}\ \}$$

Definition 5.10.2 (Startproduktion einer Sprache)**.** Eine Sprache ist wohldefiniert, wenn sie genau eine Startproduktion hat, also s_l aus einem Element besteht, das als start(l) bezeichnet wird.
$s_l \subset \wp(\mathit{Element}) =_{\mathrm{def}} \{e \mid e \in l.\mathit{SubLanguage} \cup l.\mathit{FragmentProduction}, e.\mathit{Start} = \mathit{true}\}$

Definition 5.10.3 (Typ eines Elements Type^{El})**.** Der Typ eines Elements mit Namen n ergibt sich für Fragment-Produktions-Paare aus dem Typ der Produktion und für Sprachen aus dem Typ der Startproduktion.

$$\mathrm{Type}^{El}(n) =_{\mathrm{def}} \begin{cases} \mathrm{Type}_{e.\mathit{Grammar}}(\mathrm{EN}_l(n).\mathit{ProdName}) & : \ \mathrm{EN}_l(n) \in \mathit{FragmentProduction} \\ \mathrm{Type}^{El}(\mathrm{start}(\mathrm{EN}_l(n).\mathit{Language})) & : \ \mathrm{EN}_l(n) \in \mathit{SubLanguage} \end{cases}$$

Definition 5.10.4 (Wohldefiniertheit von Fragment-Produktions-Paaren)**.** Ein Fragment-Produktions-Paar $f \in \mathit{FragmentProduction}$ ist wohldefiniert, wenn für $g = f.\mathit{Grammar}$ gilt:

(1) Die Produktion ist in der Grammatik definiert.
$f.\mathit{PRule} \in \mathrm{def}(\mathrm{PN}_g)$

(2) Die Produktion ist eine Schnittstellen-, Klassen- oder abstrakte Produktion.
$p = \mathrm{PN}_g(f.\mathit{PRule}).\mathit{Prod} \in \mathit{InterfaceProd} \cup \mathit{AbstractProd} \cup \mathit{ClassProd}$

(3) Die Produktion darf keine Parameter haben.
$\#p.\mathit{FormalParameter} = 0$

Definition 5.10.5 (Wohldefiniertheit von Einbettungen (mit Parameter))**.** Eine Einbettung (mit Parameter) ist für eine Sprachdatei l wohldefiniert, wenn
$\forall el \in l.FragmentProduction \cup l.SubLanguage : \forall e \in Emb \cup EmbParameter :$

(1) Elementname existiert
$e.ElementName \in \text{def}(\text{EN}_l)$

(2) Elementname bezeichnet Fragment
$\text{EN}_l(e.ElementName) \in FragmentProduction$

(3) ProdName existiert
$e.ProdName \in \text{def}(\text{PN}'_{\text{EN}_l(e.ElementName).Grammar})$

(4) ProdName stellt externe Produktion dar
$\text{PN}'_{\text{EN}_l(e.ElementName).Grammar}(e.ProdName).Prod \in ExternalProd$

(5) Typ der Einbettung ist ein Subtyp der externen Produktion.
$\text{Type}_l^{El}(el.ElementName) \succ \text{Type}_{\text{EN}_l(e.ElementName).Grammar}(e.ProdName)$

5.11 Semantische Domäne für die konkrete Syntax

Im Folgenden wird erklärt, wie aus dem MontiCore-Grammatikformat die konkrete Syntax einer Sprache abgeleitet wird. Dabei wird eine im Folgenden definierte X-Grammatik als semantische Domäne verwendet. Für sie wird durch eine Ableitungsfunkion erklärt, welche Sprache sie definiert. Im Gegensatz zur üblichen Definition einer Sprache durch eine kontextfreie Grammatik, bei der die Sprache durch ein Produktionssystem definiert ist, wird hier eine Abwandlung verwendet. Die gewählte Alternative hat die folgenden Eigenschaften:

- Es wird keine explizite Startproduktion festlegt, sondern die Ableitungsfunktion ist mit der Startproduktion parametrisiert, weil eine MontiCore-Grammatik mit beliebiger Startproduktion verwendet werden kann.
- Es wird ein zweischrittiges Verfahren mit Tokenklassen verwendet, die wiederum zu einer Menge von Lexemen abgeleitet werden. Diese Formalisierung imitiert das zweischrittige Erkennungsverfahren mit Lexer und Parser in der technischen Realisierung.
- Parametrisierte Produktionen bezeichnen ein Paar aus Nichtterminalnamen und Wert, das wie ein Nichtterminal abgeleitet wird.
- Die Ableitungsfunktion verwendet zwei mathematische Strukturen, die den Ableitungsprozess beeinflussen. Die Struktur AST stellt den bereits aufgebauten AST dar, wohingegen die Struktur *Global* eine im Parser verwendete Datenstruktur ist.
- Neben Nichtterminalen und Terminalen können auch spezielle Elemente innerhalb einer Produktion verwendet werden, die gezielt die zusätzlichen Datenstrukturen der Ableitungsfunktion beeinflussen.

Definition 5.11.1 (X-Grammatik)**.** Eine X-Grammatik G_X bezeichnet ein Tupel
$(N, T, \mathcal{A}, l, P, P^p)$
N Endliche Menge der Nichtterminalsymbolen
T Endliche Menge der Tokenklassen

$\mathcal{A}$ Endliches Alphabet von Eingabezeichen
$l \subset T \times \mathcal{A}^*$ Relation zwischen den Tokenklassen und ihren Lexemen.
$P \subset N \times E$ Produktionen der Grammatik
$P^p \subset N \times \mathcal{A}^* \times E$ Parametrisierte Produktionen der Grammatik

Die Menge $E = \mathcal{T}_{\Sigma_G}(V \cup K)$ bezeichnet die Menge aller allgemeinen Terme mit Variablen $V = \mathcal{A}^* \cup N \cup T$ und $Z = \mathit{Append} \cup \mathit{Push} \cup \mathit{Pop} \cup \mathit{GS} \cup \mathit{AST} \cup \mathit{Begin} \cup \mathit{End}$ zur Signatur Σ. Die Menge Z bezeichnet die zusätzlichen Variablen, die den Aufbau des AST und die globale Datenspeicherung wiederspiegeln. Die Menge *ID* umfasst die validen Bezeichner hierfür.

Begin	$= \{\text{begin}\}$	(Einführen einer AST-Ebene)
End	$= \{\text{end}\}$	(Verlassen einer AST-Ebene)
Append	$= \{\text{append}(i,t) \mid i \in \mathit{ID}, t \in T\}$	(Anfügen an ein Attribut)
Push	$= \{\text{push}(i_1,i_2) \mid i_1, i_2 \in \mathit{ID}\}$	(Setzen eines globalen Wertes)
Pop	$= \{\text{pop}(i) \mid i \in \mathit{ID}\}$	(Entfernen eines globalen Wertes)
PNt_{AST}	$= \{[n,i]_{\text{ast}} \mid n \in N, i \in \mathit{ID}\}$	(Parametrisiertes Nichtterminal)
PNt_{GS}	$= \{[n,i]_{\text{gs}} \mid n \in N, i \in \mathit{ID}\}$	(Parametrisiertes Nichtterminal)

Definition 5.11.2 (Sprache einer X-Grammatik)**.** Die Sprache einer X-Grammatik g ist durch die Funktion $L_g : E \times \mathit{AST} \times \mathit{GS} \to \wp(\mathcal{A}^*)$ mit Hilfe zweier Datenstrukturen definiert.
$\mathit{AST} = \mathit{Stack}\,\langle \mathit{Map}\,\langle \mathit{ID}, \mathit{List}\,\langle \mathcal{A}^* \rangle \rangle \rangle$
$\mathit{GS} = \mathit{Map}\,\langle \mathit{ID}, \mathit{Stack}\,\langle \mathcal{A}^* \rangle \rangle$

Die Sprache ausgehend von einem Startsymbol $n \in g.N$ ergibt sich aus $L_g(n) = L_g(n, \mathit{Stack}.\text{new}(), \mathit{GS}.\text{new}())$. L_g ist wie folgt definiert:

(1) Identität auf Wörtern über dem Alphabet.
Für $w \in \mathcal{A}^*,\ e \in E,\ a \in \mathit{AST},\ s \in \mathit{GS}$ gilt:
$L_g(w\ e, a, s) =_{\text{def}} w \cdot L_g(e, a, s)$

(2) Ersetzen von Nichtterminalen.
Für $n \in N,\ e \in E,\ \mathit{id} \in \mathit{ID},\ a \in \mathit{AST},\ s \in \mathit{GS}$ gilt:

(a) $L_g(n\ e, a, s) =_{\text{def}} \bigcup_{(n,e_1) \in P} L_g(e_1\ e, a, s)$
(b) $L_g([n,\mathit{id}]_{\text{ast}}\ e, a, s) =_{\text{def}} \bigcup_{(n,a.\text{peek}().\text{get}(\mathit{id}).\text{head}(),e_1) \in P^p} L_g(e_1\ e, a, s)$
(c) $L_g([n,\mathit{id}]_{\text{gs}}\ e, a, s) =_{\text{def}} \bigcup_{(n,s.\mathit{get}(\mathit{id}).\mathit{peek}(),e_1) \in P^p} L_g(e_1\ e, a, s)$

(3) Ersetzen von Tokenklassen durch Lexeme.
Für $t \in T,\ e \in E,\ a \in \mathit{AST},\ s \in \mathit{GS}$ gilt:

(a) $L_g(t\ e, a, s) =_{\text{def}} \bigcup_{(t,ll) \in l} (ll \cdot L_g(e, a, s))$
(b) $L_g(\text{append}(\mathit{id}, t)\ e, a, s) =_{\text{def}} \bigcup_{(t,ll) \in l} (ll \cdot L_g(e, a', s))$ mit
$a' = a.\text{tail}().\text{push}(a.\text{peek}().\text{set}(\mathit{id}, a.\text{peek}().\text{get}(\mathit{id}).\text{append}(ll)))$

(4) Alternativen.
Für $e_1, e_2, e_3 \in E,\ a \in \mathit{AST},\ s \in \mathit{GS}$ gilt:
$L_g((e_1|e_2)\ e_3, a, s) =_{\text{def}} L_g(e_1\ e_3, a, s) \cup L_g(e_2\ e_3, a, s)$

(5) Kleenescher Stern.
Für $e_1, e_2 \in E,\ a \in \text{AST},\ s \in \text{GS}$ gilt:
$L_g({e_1}^*\ e_2, a, s) =_{\text{def}} L_g(e_1\ {e_1}^*\ e_2, a, s) \cup L_g(e_2, a, s)$

(6) Setzen und Löschen globaler Werte.
Für $id_1, id_2 \in ID,\ e \in E,\ a \in AST,\ s \in GS$ gilt:

(a) $L_g(\text{push}(id_1, id_2)\ e, a, s) =_{\text{def}} L_g(e, a, s')$ mit
$s' = s.\text{set}(id_1, s.\text{get}(id_1).\text{push}(a.\text{peek}().\text{get}(id_2).\text{head}()))$

(b) $L_g(\text{tail}(id_1)\ e, a, s) =_{\text{def}} L_g(e, a, s')$ mit
$s' = s.\text{set}(id_1, s.\text{get}(id_1).\text{tail}())$

(7) AST-Aufbau.
Für $e \in E,\ a \in AST,\ s \in GS$ gilt:

(a) $L_g(\text{begin}\ e, a, s) =_{\text{def}} L_g(e, a', s)$ mit
$a' = a.\text{push}(AST.\text{new}())$

(b) $L_g(\text{end}\ e, a, s) =_{\text{def}} L_g(e, a', s)$ mit
$a' = a.\text{tail}()$

5.12 Semantische Abbildung der konkreten Syntax

Die folgende semantische Abbildung erklärt, wie eine MontiCore-Grammatik auf eine X-Grammatik abgebildet wird. Dabei wird nicht der volle Sprachumfang des MontiCore-Grammatikformats betrachtet, um die Abbildung übersichtlich gestalten zu können und die wesentlichen Aspekte darzustellen. Dabei steht insbesondere die Realisierung der parametrischen Einbettung im Vordergrund, da dieser Mechanismus dafür verantwortlich ist, dass der Erkennungsprozess nicht rein kontextfrei ist, sondern gezielt Kontextinformationen genutzt werden können. Bei der Betrachtung wurden folgende Einschränkungen vorgenommen:

1. Die lexikalische Relation wird nicht detaillierter betrachtet, obwohl sie wie die Parserproduktionen durch ein Produktionssystem definiert ist, weil es sich um die übliche Interpretation regulärer Ausdrücke handelt.

2. Konkrete Einschränkungen, die aufgrund der Berechnung des Lookaheads im Parsergenerator und des Einsatzes von Prädikaten entstehen, die Auswirkungen von Mehrdeutigkeiten und die Nutzung von Greedy/Nongreedy-Parseverhalten werden nicht betrachtet. Eine Formalisierung würde sonst kein allgemeingültiges Verständnis herstellen, sondern die technische Lösung innerhalb des verwendeten Parsergenerators erklären.

3. Variablen und Regelparameter werden nicht betrachtet, weil diese die zusätzlich notwendigen Datenstrukturen verkomplizieren. Innerhalb der folgenden Darstellung wird ein reduzierter AST als Stack dargestellt, der nur die Attribute und nicht die Kompositionen enthält. Dieses reicht aus, um die Auswahl verschiedener parametrisierter Nichtterminale aufgrund der Attribute des aktuellen AST-Knotens darzustellen. MontiCore selbst erlaubt zusätzlich die Auswahl aufgrund der Variablen und Wertte der Kompositionen, was als zusätzliche Datenstruktur ein Objektdiagramm und eine Menge der derzeit gültigen Variablenbelegungen erfordern würde. Da dieses die Darstellung verkomplizieren würde, ohne am grundsätzlichen Mechanismus der Einbettung etwas zu ändern, wird auf eine Darstellung verzichtet.

Definition 5.12.1 (Semantik einer MontiCore-Grammatik bezüglich der konkreten Syntax)**.** Eine MontiCore-Grammatik kann wie folgt auf eine X-Grammatik abgebildet werden:
$[\![.]\!]_{cs} : \mathit{Grammar} \to G_X : g \mapsto (N, T, \mathcal{A}, l, P, \emptyset)$, wobei gilt
$N = \{n \mid n \in \mathrm{def}(\mathrm{PN}'_g),\ (\gamma, \pi) = \mathrm{PN}'_g(n), \pi \notin \gamma.\mathit{LexProd}\}$
$T = \{n \mid n \in \mathrm{def}(\mathrm{PN}'_g),\ (\gamma, \pi) = \mathrm{PN}'_g(n), \pi \in \gamma.\mathit{LexProd}\}$
$\mathcal{A} = g.\mathit{Alphabet}$
$l = \bigcup_{n \in T} \{\ (n, \pi.\mathit{RegularExpression})\ \mid (\gamma, \pi) \in \mathrm{PN}'_g(n)\}$
$P = \bigcup_{i \in \{1, \ldots, 7\}} P_i$, wobei

(1) Schnittstellen in abstrakten und Klassenproduktionen

$$P_1 = \{\ (i, n) \mid\ n \in \mathrm{def}(\mathrm{PN}'_g),\ (\gamma, \pi) = \mathrm{PN}'_g(n), \pi \in \gamma.\mathit{ClassProd} \cup \gamma.\mathit{AbstractProd},\ i \in \pi.\mathit{Implementation.Name}\ \}$$

(2) Vererbung in Parserproduktionen

$$P_2 = \{\ (c, n) \mid\ n \in \mathrm{def}(\mathrm{PN}'_g),\ (\gamma, \pi) = \mathrm{PN}'_g(n), \pi \in \gamma.\mathit{ParserProd},\ c \in \pi.\mathit{Inheritance.Name}\ \}$$

(3) Produktionskörper mit identischem Rückgabewert und definiertem Typ

$$P_3 = \{\ (n, e) \mid\ n \in \mathrm{def}(\mathrm{PN}'_g),\ (\gamma, \pi) = \mathrm{PN}'_g(n), \pi \in \gamma.\mathit{ClassProd},\ \pi.\mathit{CType}_1 = \pi.\mathit{CType}_2,\ e = \text{begin } \mathrm{h}(p.\mathit{ProdElement})\ \mathrm{h}(\pi.\mathit{Body}) \text{ end }\}$$

(4) Produktionskörper mit unterschiedlichem Rückgabewert und definiertem Typ

$$P_4 = \{\ (n, e) \mid\ n \in \mathrm{def}(\mathrm{PN}'_g),\ (\gamma, \pi) = \mathrm{PN}'_g(n), \pi \in \gamma.\mathit{ClassProd},\ \pi.\mathit{CType}_1 \neq \pi.\mathit{CType}_2,\ e = \mathrm{h}(\pi.\mathit{Body})\ \}$$

Dabei wird die Funktion h, die die Elemente von $p.\mathit{Body}$ auf $x.E_x$ abbildet, verwendet. Da beide Algebren dieselbe Signatur haben, wird hier nur erklärt, wie die Variablen überführt werden:

(a) $x \in \mathit{ProdElement}, r = x.\mathit{ProdElementRef} \in \mathit{NonTerminal}, n = r.\mathit{Name}$:

$$h(x) =_{\mathrm{def}} \begin{cases} [n, \mathit{name}]_{\mathrm{ast}} & : \ r.\mathit{ParserParameter} = (\mathrm{ast}, \mathit{name}) \\ [n, \mathit{name}]_{\mathrm{gs}} & : \ r.\mathit{ParserParameter} = (\mathrm{global}, \mathit{name}) \\ \mathrm{append}(x.\mathit{Name}, n) & : \ \mathrm{PN}'_g(x.\mathit{Name}).\mathit{Prod} \in \mathit{LexProd}, \\ & \quad x.\mathit{ProdElementType} = \mathrm{AST} \\ n & : \ \mathrm{sonst} \end{cases}$$

(b) $x \in \mathit{ProdElement}, t = x.\mathit{ProdElementRef} \in \mathit{Terminal}$:

$$h(x) =_{\mathrm{def}} \begin{cases} \mathit{append}(x.\mathit{Name}, t.\mathit{String}) & : \ x.\mathit{ProdElementType} = \mathrm{AST} \\ t.\mathit{String} & : \ \mathrm{sonst} \end{cases}$$

(c) $x \in \mathit{ProdElement}, c = x.\mathit{ProdElementRef} \in \mathit{Constant} : \mathrm{h}(x) =_{\mathrm{def}} c.\mathit{String}$

(d) $x \in \mathit{Constr} : \mathrm{h}(x) =_{\mathrm{def}} \text{begin}$

(e) $x \in \mathit{Push} : \mathrm{h}(x) =_{\mathrm{def}} \mathrm{push}(x.\mathit{StackName}, x.\mathit{VarName})$

(f) $x \in \mathit{Pop} : \mathrm{h}(x) =_{\mathrm{def}} \mathrm{pop}(x.\mathit{StackName})$

Die Komposition verschiedener Fragmente und Sprachen zu einer neuen Sprache macht es notwendig, die einzelnen Nichtterminale und Terminale einer Sprache einheitlich umzubenennen, um diese dann disjunkt vereinen zu können.

Definition 5.12.2 (Umbenennung von X-Grammatiken durch die Funktionen dis und ren)**.** Die Funktion dis : $G_X \times \mathit{Name} \to G_X$ bildet eine X-Grammatik auf eine andere X-Grammatik ab. Dabei werden alle Nichtterminale und Tokenklassen (in Ausdrücken der Grammatik) durch eine hier nicht näher definierte Funktion ren : $E \times \mathit{Name} \to E$ umbenannt und so die disjunkte Vereinigung von Grammatiken vorbereitet. Die Funktionen dis und ren müssen dabei die folgenden Eigenschaften erfüllen:

(1) Tokenklassen und Nichtterminale sind disjunkt
Für $g_1, g_2 \in G_X, n_1 \neq n_2 \in \mathit{Name}$ gilt
$g_1.T \cap \mathrm{dis}(g_1, n_1).T = \mathrm{dis}(g_1, n_1).T \cap \mathrm{dis}(g_2, n_2).T = \emptyset$
$g_1.N \cap \mathrm{dis}(g_1, n_1).N = \mathrm{dis}(g_1, n_1).N \cap \mathrm{dis}(g_2, n_2).N = \emptyset$

(2) Sprache bleibt unverändert
Für $g \in G_X, n \in \mathit{Name}, a \in \mathit{AST}, s \in \mathit{GS}$ gilt:
$L_g(e, a, s) = L_{\mathrm{dis}(g,n)}(\mathrm{ren}(e, n), a, s)$

Definition 5.12.3 (Semantik einer Sprachdatei bezüglich der konkreten Syntax)**.** Eine Sprachdatei l kann wie folgt auf eine X-Grammatik abgebildet werden:
$[\![.]\!]_{cs} : \mathit{Language} \to G_X : g \mapsto (N, T, \mathcal{A}, l, P, P^p)$, wobei gilt

$$E = \{\mathrm{dis}([\![x.\mathit{Grammar}]\!]_{cs}, x.\mathit{ElementName}) \mid x \in l.\mathit{FragmentProduction}\} \cup \{\mathrm{dis}([\![x]\!]_{cs}, x.\mathit{ElementName}) \mid x \in l.\mathit{SubLanguage}\}$$

(1) Disjunkte Vereinigung aller Nichtterminale
$N = E.N$

(2) Disjunkte Vereinigung aller Terminale
$T = E.T$

(3) Vereinigung der Alphabete
$\mathcal{A} = E.\mathcal{A}$

(4) Vereinigung aller regulären Abbildungen
$l = E.l$

(5) Produktionen
$P = P_1 \cup P_2$

(a) *Alle Produktionen aller Fragmente und Subsprachen*
$P_1 = E.P$

(b) *Zusätzliche Produktionen aufgrund der Einbettung*
$P_2 = \bigcup_{el \in l.\mathit{SubLanguage} \cup l.\mathit{FragmentProduction}} \bigcup_{e \in e.\mathit{Emb}} (\mathrm{ren}(e.\mathit{ProdName}, e.\mathit{ElementName}), h(el.\mathit{ElementName}))$

(6) Parametrisierte Produktionen
$P = P_1^p \cup P_2^p$

(a) *Alle Produktionen aller Fragmente und Subsprachen*
$P_1^p = E.P^p$

(b) *Zusätzliche Produktionen aufgrund der parametrisierten Einbettung*
$P_2^p = \bigcup_{el \in l.SubLanguage \cup l.FragmentProduction} \bigcup_{e \in e.EmbParameter}$
$(\text{ren}(e.ProdName, e.ElementName)e.String, h(el.ElementName)$

Die Funktion h bildet einen Elementnamen, der eine Sprache bzw. ein Fragment repräsentiert, auf das entsprechende Start-Nichtterminal ab.

$$h_l(n) =_{\text{def}} \begin{cases} \text{ren}(h_x.Language(x.Language), n) & : \quad x = \text{EN}_l(n) \in SubLanguage \\ \text{ren}(x.Language.PRule, n) & : \quad x = \text{EN}_l(n) \in FragmentProduction \end{cases}$$

5.13 Zusammenfassung

Dieses Kapitel formalisiert das MontiCore-Grammatikformat aus den Kapiteln 3 und 4. Die Darstellung beschränkt sich dabei auf die Grundform, die nicht den „syntaktischen Zucker“ beinhaltet, der die konkrete Verwendung deutlich vereinfacht, eine Formalisierung aber nur unnötig verkompliziert. Dabei wurden zunächst die Kontextbedingungen dargestellt, die eine wohlgeformte Grammatik erfüllen muss. Diese werden teilweise durch die Zielplattform beeinflusst, um sicherzustellen, dass über allgemeine Anforderungen hinaus die abstrakte Syntax erfolgreich in eine Implementierung umgesetzt werden kann. Dabei ist zunächst wichtig, dass sowohl Produktionsnamen als auch Typnamen jeweils eindeutig sind, wenn Sprachvererbung eingesetzt wird und sich so eindeutig aus den Produktionen Typen ableiten lassen. Diese Typen können dann an verschiedenen Stellen der Grammatik eingesetzt werden, wobei bestimmte Subtypbeziehungen beachtet werden müssen. Ausgehend von den Typbeziehungen müssen sich die Typen der Attribute, Kompositionen, Assoziationsenden und Variablen eindeutig aus der Struktur der Grammatik und den AST-Regeln ableiten lassen. Bei den Variablen wird zusätzlich statisch geprüft werden, ob sie die Kardinalitäten der Parameter erfüllen.

Die Ableitung der abstrakten Syntax einer DSL aus der Grammatik wurde durch die Umsetzung in UML/P-Klassendiagramme erklärt, wobei sich die Komposition verschiedener Grammatiken durch Sprachvererbung und Einbettung durch die Komposition der einzelnen Klassendiagramme ergibt, die aus den Grammatiken abgeleitet werden. Dadurch kann auf der Ebene der abstrakten Syntax nachvollzogen werden, welche Klassen und Schnittstellen aus einer Grammatik erzeugt werden, wobei strukturell sichergestellt ist, dass jede Klasse und Schnittstelle nur einmal erzeugt werden kann.

Die durch eine MontiCore-Grammatik definierte konkrete Syntax einer DSL wurde durch die Abbildung auf X-Grammatiken erklärt. Für die X-Grammatiken wird in einem zweiten Schritt die durch sie definierte Sprache erklärt, wobei damit die Menge der zur Grammatik konformen Wörter gemeint ist. Somit lässt sich durch die transitive Anwendung erkennen, welche Wörter zu einer MontiCore-Grammatik konform sind. Durch die Abbildung von Sprachdateien und Grammatiken auf diese einfache Grammatikform wurden die Auswirkungen der Sprachvererbung erklärt, weil die X-Grammatiken diesen Mechanismus nicht bieten. Die X-Grammatiken verwenden jedoch spezielle parametrisierte Nichtterminale, die wie die Einbettung realisiert sind und zeigen, wie sich Teile der Eingabe auf die Auswahl der Nichtterminale auswirken.

Kapitel 6

MontiCore

Die zentrale Aufgabe des MontiCore-Frameworks besteht darin, Grammatiken und Sprachdateien einzulesen, um dazu passende Parser und AST-Klassen zu erzeugen und so die agile Modellierung auf DSL-Basis durch die Realisierung von automatisierten Werkzeuge zu unterstützen. Dabei unterscheidet sich die Struktur des Frameworks vor allem durch seine Erweiterungsfähigkeit von anderen Ansätzen. Es wurden gezielt Erweiterungspunkte identifiziert, die eine Anpassung und vor allem Erweiterung des Werkzeugs für spezifische Bedürfnisse ermöglicht. Somit können weitere Softwarekomponenten erzeugt werden, um DSLs leichter für die generative Softwareentwicklung einsetzen zu können. Die Möglichkeiten und Einsatzszenarien der Erweiterungspunkte werden in diesem Kapitel durch eine Auswahl der existierenden Erweiterungen illustriert.

Das Kapitel ist dabei wie folgt gegliedert: In Abschnitt 6.1 wird die Struktur des Projekts und die Aufteilung in Teilprojekte beschrieben. Der Abschnitt 6.2 erläutert den Funktionsumfang des MontiCore-Generators, indem die wesentlichen aus den Eingabesprachen erzeugten Dateitypen aufgelistet werden. Der Abschnitt 6.3 zeigt die Software-Architektur des MontiCore-Generators. In Abschnitt 6.4 werden die Erweiterungspunkte von MontiCore erklärt und einige der vorhandenen Erweiterungen vorgestellt. In Abschnitt 6.5 wird der Einsatz von MontiCore zur Erstellung generativer Werkzeuge erläutert. Der Abschnitt 6.6 fasst das Kapitel kurz zusammen.

6.1 Projektstruktur

Der MontiCore-Generator ist in Teilprojekte unterteilt, die jeweils ein eigenes OSGi-Bundle (Open Services Gateway initiative) [OSG07] bilden. Die Abbildung 6.1 zeigt die einzelnen Teilprojekte und deren Abhängigkeiten untereinander, wenn sie mit installierter Generator-Komponente betrieben werden. Dabei kann MontiCore entweder über die Konsole (linker Teil der Abbildung) oder über die Eclipse-Integration genutzt werden (rechter Teil der Abbildung). Innerhalb einer OSGi-Laufzeitumgebung lassen sich unterschiedliche Versionen parallel betreiben, ohne dass sie sich gegenseitig beeinflussen. Für die Ausführung auf der Kommandozeile ist keine solche Laufzeitumgebung notwendig, sondern es kann eine Standard-Java-VM genutzt werden, wobei für verschiedene Versionen von MontiCore dann auch unterschiedliche Instanzen der VM benutzt werden müssen.

In Abbildung 6.2 wird die Verteilung der Komponenten als UML-Verteilungsdiagramm gezeigt, wenn der Generator über die *Online Software Transformation Platform* (OSTP) [Her06] genutzt wird. Die OSTP ist dabei eine Plattform zur Realisierung von Transformationen, die auf einem zentralen Server ausgeführt werden. Dabei werden die Arbeitsbereiche

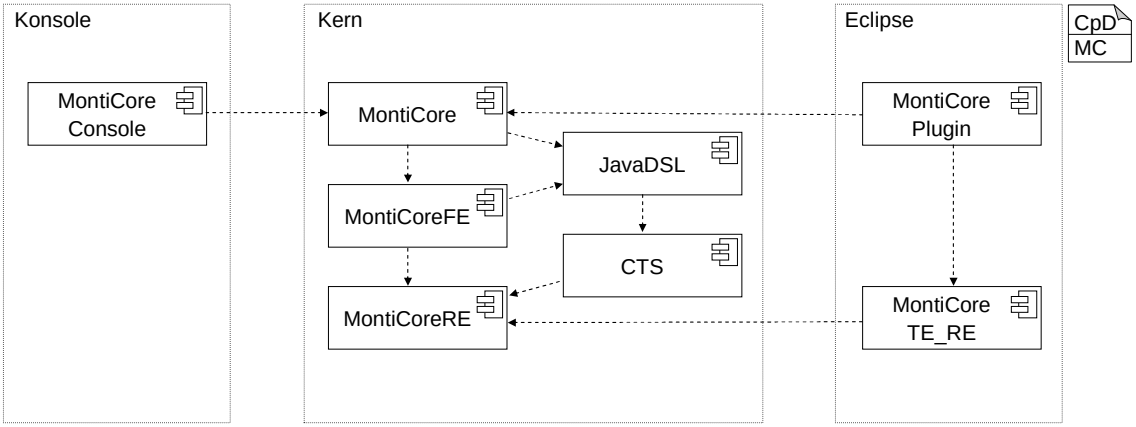


Abbildung 6.1: Komponentendiagramm für den installierten Generator in Eclipse und auf der Konsole mit den Abhängigkeiten zwischen den Modulen.

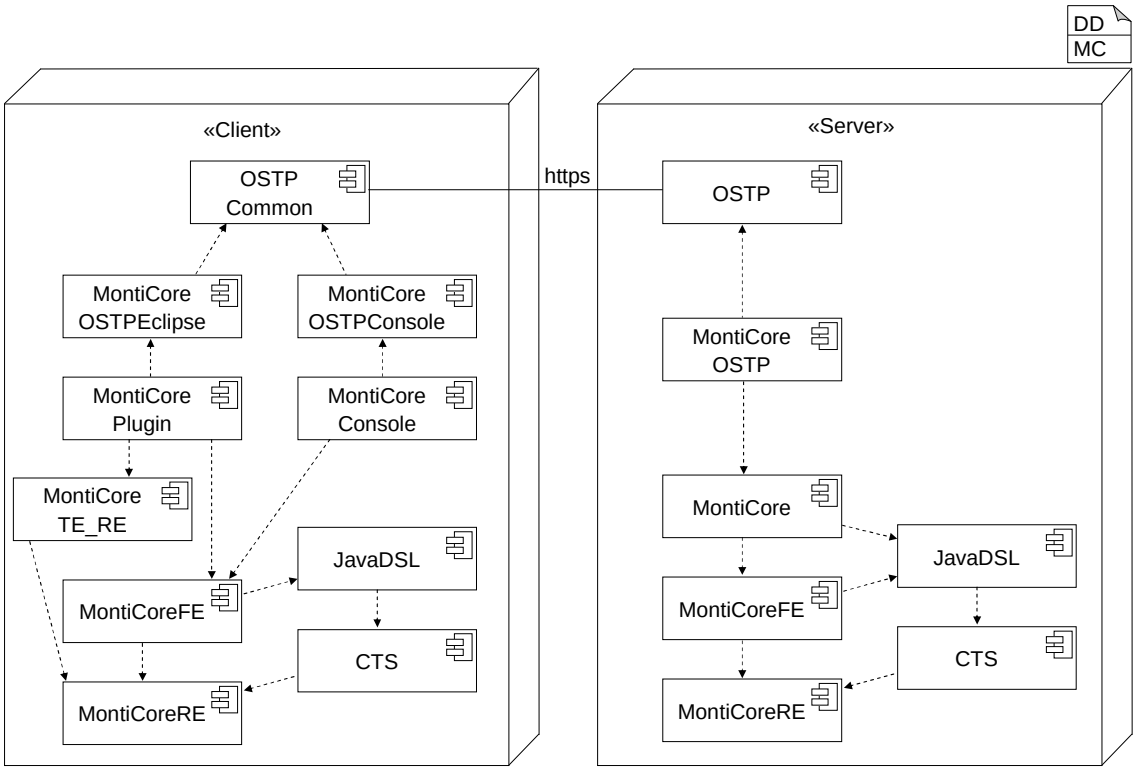


Abbildung 6.2: Verteilungsdiagramm für den Betrieb über den OSTP-Service.

auf Client und Server vor und nach den Transformationen synchronisiert, so dass der Entwickler die Transformationsdienste ohne Installation nutzen kann. Neben einer möglichen generischen Integration nutzt MontiCore die Möglichkeit, den Zugriff über OSTP in seiner eigenen Anwendung zu kapseln. Daher ist die Nutzung für den Entwickler transparent, weil unabhängig von der gewählten Betriebsart dasselbe Kommandozeilen- oder Eclipsefrontend genutzt wird. Die Erkennung der Betriebsart erfolgt automatisiert aufgrund der Installation auf dem Client.

Die einzelnen in den Abbildungen dargestellten Komponenten erfüllen klar definierte Aufgaben innerhalb des MontiCore-Generators. Diese Dissertation umfasst die Komponenten MontiCore, MontiCoreFE und MontiCoreRE sowie Teile der JavaDSL. Die Darstellungen innerhalb dieses Kapitels beschränken sich daher im Wesentlichen auf diese Teile. Unabhängig davon sind die wesentlichen Aufgabenbereiche der einzelnen Komponenten die folgenden:

- Das *Common Type System* (CTS) beinhaltet die Grundinfrastruktur für Symboltabellen.
- Die *JavaDSL* beinhaltet eine Java 6.0 Grammatik mit einer auf dem CTS basierenden Symboltabelle.
- *MontiCore* beinhaltet den MontiCore-Generator.
- *MontiCoreConsole* ist die Konsolenanwendung zur Verwendung des MontiCore-Generators.
- *MontiCoreFE* umfasst die Symboltabelleninfrastruktur für den MontiCore-Generator. Diese Komponente kann Grammatiken vollständig auf Korrektheit überprüfen und ist ausreichend für eine Nutzerunterstützung in Eclipse.
- *MontiCoreRE* ist die Laufzeitumgebung für MontiCore-Anwendungen. Sie beinhaltet die Grundinfrastruktur für die Parsererzeugung und -ausführung sowie das DSLTool-Framework und wird daher zur Laufzeit der DSLs benötigt.
- *MontiCoreOSTP* umfasst die MontiCore-OSTP-Integration.
- *MontiCoreOSTPConsole* beinhaltet die Funktionalität für die Anbindung der MontiCore-Konsolenanwendung an OSTP.
- *MontiCoreOSTPEclipse* beinhaltet die Funktionalität für die Anbindung der MontiCore-Eclipseanwendung an OSTP.
- *MontiCorePlugin* bezeichnet die Eclipse-Integration von MontiCore.
- *MontiCoreTE_RE* ist die Laufzeitumgebung für die generierten Editoren.
- *OSTP* bezeichnet die Online-Software-Transformation-Platform.
- *OSTPCommon* beinhaltet die Basisfunktionalität zur Kommunikation mit dem OSTP-Server.

6.2 Funktionsumfang des MontiCore-Generators

Der MontiCore-Generator verarbeitet Modelle, die in fünf Sprachen formuliert sein können, und erzeugt daraus verschiedene Arten von Dateien.

- Grammatiken mit der Dateiendung „mc“ definieren ein Sprachfragment und bilden die Grundlage für die Parser- und AST-Klassenerzeugung.
- Sprachdateien verwenden die Dateiendung „lng“ und bezeichnen die Kombination mehrerer Fragmente zu einer Sprache. Aus einer Sprachdatei werden zusätzlich Klassen erzeugt, die eine Integration in das DSLTool-Framework ermöglichen.

- Die Bundledatei „mc.bdl“ enthält den Bundlenamen und weitere Angaben, die eine Integration in OSGi ermöglichen und es erlauben, Abhängigkeiten zwischen Komponenten zu spezifizieren.
- Antlr-Grammatiken verwenden die Dateiendung „g“ und können normal mit MontiCore verarbeitet werden.
- Datentypbeschreibungen mit der Dateiendung „ast“ beschreiben dieselben Datenstrukturen, die aus Grammatiken als Repräsentation der abstrakten Syntax erzeugt werden.

Für jede einzelne Dateiart existiert eine Anzahl von Workflows. Ein Workflow stellt innerhalb des DSLTool-Frameworks (vgl. Kapitel 9), das als Referenzarchitektur für MontiCore verwendet wird, zustandslose Algorithmen dar, die auf Eingabedateien ausgeführt werden. Der Inhalt der Eingabedateien wird dabei durch so genannte Root-Objekte im Arbeitsspeicher verwaltet, wobei die Objekte von Root-Factories erzeugt werden. Standardmäßig werden für alle Dateiarten die beiden Workflows *parse* und *generate*, sofern diese für die Dateiart existieren, und alle Partmerger ausgeführt. PartMerger sind ein weiterer Baustein des DSLTool-Frameworks, der der Erstellung von Dateien dient, deren Inhalt sich aus mehreren Eingabedateien ergibt. Dazu wird zunächst für die einzelnen Eingabedateien ein Part erstellt, der dann vom PartMerger zusammengefügt wird. Während der Workflow *parse* verpflichtend ist, weil er den Erkennungsprozess und den Symboltabellenaufbau steuert, kann der Workflow *generate* ersetzt werden. Dieses geschieht zum Beispiel durch die Kommandozeilenparameter `-workflow grammar isabelle`, um für die Grammatiken nur den Workflow `isabelle` auszuführen. Analog können Partmerger mit `-partmerger name` ausgewählt werden. Die Aufrufe verwenden dabei die Standards des DSLTool-Frameworks, die in Kapitel 9 detailliert beschrieben werden. Die folgende Liste stellt die registrierten Workflows für die einzelnen Dateitypen im MontiCore-Generator dar.

Grammatiken (grammar)

parse baut den AST und die Symboltabelle auf und prüft die Kontextbedingungen.

generate erzeugt Lexer, Parser und Datenstrukturen für die ClassicPlatform.

EMFGenNew (in der Erweiterung MontiCoreEMF [Hof08]) erzeugt Transformationen und Datenstrukturen für die Integration von MontiCore und EMF [BSM+03].

pp erzeugt eine wohlformatierte Grammatik aus der Eingabe mittels eines Pretty-Printers.

showsymtab gibt den Inhalt der Symboltabelle auf die Konsole aus.

quality überprüft Qualitätskriterien der Grammatik und weist auf problematische Konstrukte hin.

isabelle im Zuge der semantischen Fundierung der UML [CGR08] erstellter Workflow, der aus einer Grammatik Datenstrukturen für Isabelle [NPW02] erzeugt.

parsenosymtab führt nur einen Parseprozess aber ohne Symboltabellenaufbau und Kontextbedingungsprüfung durch.

createinstances (in der Erweiterung MCSupport) erzeugt exemplarische Instanzen, die konform zu einer Grammatik sind [Sah06, Ren07].

pptest (in der Erweiterung MCSupport) überprüft den Prettyprinter aufgrund systematisch erzeugter exemplarischer Instanzen [Ren07].

Sprachdateien (language)

parse baut den AST auf und prüft die Kontextbedingungen.

generate erzeugt die spezifizierten Root-Klassen, Root-Factories und Parsing-Workflows.

Bundledatei (bundle)

parse baut den AST auf und prüft die Kontextbedingungen.

generate fügt die notwendigen Informationen zur Partdatei der Manifst.mf hinzu.

Antlr-Grammatiken (antlr)

parse baut den AST auf und prüft die Kontextbedingungen.

generate erzeugt einen Antlr-Parser und einen Antlr-Lexer.

Datentypbeschreibungen (ast)

parse baut den AST auf und prüft die Kontextbedingungen.

generate erzeugt die Datenstrukturen.

Die folgenden PartMerger sind registriert:

manifest erzeugt die manifest.mf aus der Bundledatei und weiteren automatisch aus Grammatiken erzeugten Abhängigkeiten.

pluginxml erzeugt die Plugin.xml (für die Eclipse-Integration), wenn entsprechende Konzepte in Grammatiken oder Sprachdateien verwendet wurden.

pluginClass erzeugt die Plugin.java (für die Eclipse-Integration), wenn entsprechende Konzepte in Grammatiken oder Sprachdateien verwendet wurden.

pluginAccessor erzeugt die PluginAccessor.java (für die Eclipse-Integration), wenn entsprechende Konzepte in Grammatiken oder Sprachdateien verwendet wurden.

MontiCore verwendet in Zusammenhang mit den verarbeiteten DSLs, wie in [Wil03] empfohlen, einen 80%-Ansatz. Darunter wird verstanden, dass nur die Elemente durch eine DSL beschrieben werden, für die ein generativer Einsatz eine große Produktivitätssteigerung verspricht. Die Details eines Projekts, die weniger regulären Gesetzen folgen, werden weiterhin durch Programmierung in einer GPL erfasst. Ein solches Vorgehen macht eine Integration von generierten und handcodierten Code notwendig. Dabei ist es nicht wünschenswert, dass der Entwickler die Interna der Codegenerierungen kennen muss, um die Interaktion programmieren zu können. Dieses würde die Gefahr mit sich bringen, dass zufällige Eigenschaften der Generierung genutzt werden, die bei einer Veränderung der Generierung verloren gehen. Dabei ist es für den Entwickler schwierig, die richtigen Ansatzpunkte zu erkennen und so eine einfache Integration zu schaffen. Die von MontiCore erzeugten Klassen und Schnittstellen besitzen die im Folgenden dargestellten Eigenschaften, auf die sich ein Entwickler verlassen kann.

6.2.1 AST-Klassen

Aus einer Grammatik wird, wie in Kapitel 3 beschrieben und in Kapitel 5 formalisiert, die abstrakte Syntax einer DSL abgeleitet. Neben der Ableitung aus einer Grammatik bietet die Datentypbeschreibungssprache eine Möglichkeit zur expliziten Erstellung solcher Datentypen. Diese abstrakte Syntax in Form eines UML/P-Klassendiagramms wird dann spezifisch für eine spezielle Plattform umgesetzt. Eine Plattform ist dabei eine Bezeichnung für eine Ausführungsumgebung, auf der die Klassen der abstrakten Syntax vom Parser instanziert werden. Die folgenden Darstellungen beziehen sich auf die so genannte *ClassicPlatform*, die die Standardplattform innerhalb des MontiCore-Generators darstellt. Die *ClassicPlatform* realisiert die abstrakte Syntax einer DSL auf die folgende Weise:

- Die Klassen und Schnittstellen werden als Java-Klassen und -Schnittstellen realisiert, wobei der Name mit dem Prefix AST versehen wird und die Klassen im Unterpaket `_ast` abgelegt werden. Dabei wird als gemeinsame Oberschnittstelle `mc.ast.ASTNode` ($\top$, vgl. Definition 5.4.8, Seite 91) und als Klasse `ASTCNode` verwendet.

- Die Attribute der Klassen und Schnittstellen werden durch get/set-Methoden realisiert. Bei einer Kardinalität von mehr als eins, wird eine `java.util.List` vom selben Typ verwendet.

- Die Kompositionen werden durch get/set-Methoden innerhalb der jeweiligen Klasse realisiert. Bei einer Kardinalität von mehr als eins wird eine speziell generierte Liste vom selben Typ verwendet.

- Die Enumerationen werden durch Integer-Konstanten realisiert, die zentral in einer Klasse mit dem Prefix „ASTConstants“ gefolgt vom Namen der Grammatik definiert werden.

- Die Kommentare innerhalb der Grammatik werden in Javadoc-Kommentare in den Klassen und Schnittstellen umgewandelt.

Durch diese Realisierung der AST-Klassen wurden folgende Variationspunkte gewählt (vgl. Abschnitt 5.7):

- Variationspunkt 1: Mehrfachvererbung zwischen Klassentypen ist nicht erlaubt.

- Variationspunkt 2: Subtypisierung von Attributen mit Kardinalität 1.

- Variationspunkt 3: Keine Mischung bei Kardinalität der Attribute.

- Variationspunkt 4: Keine Typisierung von Konstanten.

- Variationspunkt 5: Keine Mischung bei Kardinalität der Konstanten.

- Variationspunkt 6: Subtypisierung von Kompositionen mit Kardinalität 1.

- Variationspunkt 7: Keine Mischung bei Kardinalität der Kompositionen.

6.2.2 Antlr-Parser

Aus einer MontiCore- oder einer Antlr-Grammatik wird mittels der eingebetteten Antlr-Version eine Lexer- und eine Parser-Klasse erzeugt. Die Namensgebung ergibt sich dabei aus dem Grammatiknamen gefolgt von `Lexer` bzw. `Parser`. Wird der Parser aus einer Antlr-Grammatik erzeugt, kann er wie in [Par07] beschrieben genutzt werden. Sind die Klassen jedoch aus einer MontiCore-Grammatik entstanden, werden sie innerhalb der MontiCore-Parser intern genutzt und sollten daher nicht direkt verwendet werden. Der folgende Abschnitt beschreibt die Nutzung über die dokumentierte Schnittstelle `MCConcreteParser`.

6.2.3 Parser

Das Klassendiagramm in Abbbildung 6.3 zeigt die wichtigsten Attribute und Methoden der Klasse `MCConcreteParser`, die einen Parser im MontiCore-Kontext kapselt. Das Vorkommen von Parsefehlern kann am Attribut `error` erkannt werden, das Attribut `name` kennzeichnet das Objekt und das Attribut `parserTarget` entscheidet darüber, ob am Ende des Parsevorgangs ein Dateiendezeichen erwartet wird oder nicht. Die Attributzugriffe erfolgen wie überall im MontiCore-Framework über Get- und Set-Methoden.

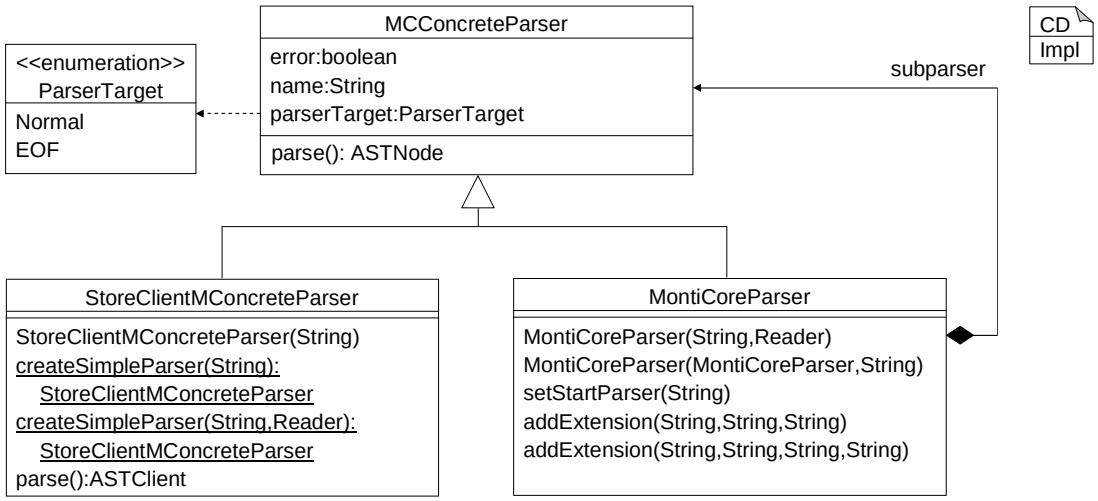


Abbildung 6.3: Grundstruktur der Parser im MontiCore-Framework

Aus einer MontiCore-Grammatik wird für jede Parserproduktion ohne Regelparameter eine Unterklasse von `MCConcreteParser` erzeugt. Der Name setzt sich dabei aus dem Grammatiknamen, dem Namen der Produktion und dem Zusatz `MCConcreteParser` zusammen. Im Klassendiagramm ist exemplarisch die Klasse `StoreClientMCConcreteParser` dargestellt, die aus der Produktion `Client` einer Grammatik mit Namen `Store` entstanden ist. Die Methode `parse()` ist dabei so überschrieben, dass sie die entsprechende Datenstruktur typsicher zurückliefert. Für diese Klassen gibt es zwei Verwendungsmöglichkeiten: Erstens können die statischen Methoden `createSimpleParser(...)` verwendet werden, um auf kompakte Art und Weise einen Parser für eine Datei oder einen Reader zu erzeugen. Dieses ist vor allem zum Schreiben von Unit-Tests und zur einfachen Erzeugung von komplexen AST-Datenstrukturen aus kurzen Textfragmenten sinnvoll. Zweitens können die `MCConcreteParser` zusammen mit der Klasse `mc.grammar.MontiCoreParser` eingesetzt werden, die ihrerseits ein spezieller `MCConcreteParser` ist. Diese Kombination erlaubt insbesondere auch die Verwendung von Spracheinbettung, weil mehrere `MCConcreteParser` mit einem `MontiCoreParser` kombiniert werden können.

Ein `MontiCoreParser` verwendet daher eine Menge an Subparsern, die für den Parsevorgang verantwortlich sind. Diese werden eindeutig über ihren Namen identifiziert. Die Methode `setStartParser(String)` erlaubt den StartParser festzulegen, wohingegen die Methoden `addExtension(...)` unter Angabe des äußeren und inneren Fragments, des externen Produktionsnamens und gegebenenfalls des Parameters die Einbettung beschreiben. Die Konstruktoren dienen zur eigenständigen Instanzierung als Parser auf einer Eingabedatei oder zur Verwendung als Subparser eines anderen `MontiCoreParser`s.

6.2.4 Root-Klassen

Die Abbildung 6.4 zeigt eine Sprachdatei, die zur Kombination der Fragmente zu einer Sprache dient. Damit wird das Beispiel aus Kapitel 3 wieder aufgenommen. In Zeile 3 wird zunächst eine Root-Klasse definiert, die die Repräsentation der Eingabedatei im Speicher darstellt. Dabei wird ein Name frei vergeben und auf eine AST-Klasse im selben Paket verwiesen. Aus diesen Angaben wird dann eine Root-Klasse erzeugt.

MontiCore-Grammatik

```
1  language ShopSystem {
2
3    root ShopRoot<ShopSystem>;
4
5    rootfactory ShopRootFactory for ShopRoot<ShopSystem> {
6
7      Shop11.ShopSystem shop <<start>>;
8
9      general.Statement.Stmt s in shop.StatementCash;
10
11     brand.Visa.Statement s1 in shop.StatementCredit(visa);
12     brand.Master.Stmt s2 in shop.StatementCredit(master);
13   }
14
15   parsingworkflow ShopRootParsingWorkflow for ShopRoot<ShopSystem>;
16
17 }
```

Abbildung 6.4: Kombination mittels einer Sprachdatei

Weitere Details zur Bedeutung von Root-Klassen und der im Folgenden dargestellten Parsing-Workflows und Root-Factories innerhalb des DSLTool-Frameworks finden sich in Abschnitt 9.2.

6.2.5 Root-Factory

In den Zeilen 5 bis 13 der Abbildung 6.4 wird eine Root-Factory definiert. Dabei werden, wie bereits in Abschnitt 4.2 erklärt, Fragmente miteinander zu einer Sprache kombiniert. Aus dieser Angabe wird eine Klasse `ShopRootFactory` erzeugt, die die Erzeugung eines Root-Objekts einschließlich der Erzeugung des Parsers als Aufgabe hat. Individuelle Anpassungen können am besten durch Unterklassenbildung realisiert werden. Die Root-Factory kann wie in Abschnitt 9.2 erklärt in das DSLTool-Framework eingebunden werden, um ein generatives Werkzeug zu realisieren.

6.2.6 Parsing-Workflow

In Zeile 15 der Abbildung 6.4 wird ein Workflow mit Namen `ShopRootParsingWorkflow` definiert, der den Parser eines Root-Objekts aufruft, Fehlermeldungen geeignet auswertet und den AST in das Root-Objekt überführt. Der Parsing-Workflow kann wie in Abschnitt 9.2 erklärt in das DSLTool-Framework eingebunden werden, um ein generatives Werkzeug zu realisieren. Dabei kann er durch weitere Workflows ergänzt werden, die die gewünschte Funktionalität realisieren.

6.2.7 Manifest.mf

Die Datei `Manifest.mf` beschreibt ein Java- bzw. OSGi-Bundle mit seinen Eigenschaften und Abhängigkeiten zu anderen Bundles. Unter anderem werden hier auch der symbolische Name und die Version des Bundles festgelegt, die dieses eindeutig identifizieren. Die Inhalte der Datei entstehen aus der Datei `mc.bdl`, die dieselbe Syntax verwendet, und weiteren automatisch ermittelten Abhängigkeiten zu MontiCore-Laufzeitumgebungen.

6.3 Software-Architektur

Die folgende Darstellung der Software-Architektur bezieht sich auf die Kernelemente des Generators, d.h. die Komponenten `MontiCore` und `MontiCoreFE` des MontiCore-Frameworks. Zusätzlich werden spezifische Elemente der Laufzeitumgebung `MontiCoreRE` erklärt.

Die Software-Architektur des MontiCore-Generators basiert auf der Referenzarchitektur des DSLTool-Frameworks, welche detailliert in Kapitel 9 beschrieben wird. Dabei wird die Struktur der Referenzarchitektur übernommen, einzelne Systemteile jedoch durch spezifische Varianten ersetzt. Zusätzlich werden Erweiterungspunkte definiert, die einer Anpassung des Werkzeugs durch den Sprachentwickler dienen. Die Architektur ist in zwei Varianten aufgeteilt: Erstens das `MontiCoreParsingTool`, das die syntaktische Prüfung und die Kontextbedingungsanalyse durchführen kann. Dieses Werkzeug wird innerhalb von MontiCore auf dem Client verwendet, wenn es über die OSTP-Plattform ausgeführt wird. Zweitens die Klasse `MontiCore`, welche verwendet wird, wenn MontiCore mit installiertem Generator genutzt wird. Dabei liegen die Unterschiede neben den strukturellen Erweiterungen vor allem in der größeren Anzahl verfügbarer Workflows. Die Abbildung 6.5 stellt den Zusammenhang der beiden Klassen `MontiCoreParsingTool` und `MontiCore` mit der Referenzarchitektur dar.

Im Gegensatz zur Referenzarchitektur wird die Konfiguration des Werkzeugs durch die spezifische Klasse `MontiCoreConfiguration` ersetzt, um zusätzlich die Erweiterungspunkte zu verwalten. Die Basisfunktionalität, die Verwaltung der vorhandenen Workflows, bleibt dabei erhalten. Die Root-Factory `DSLRootFactory` wird mit den fünf verarbeiteten Sprachen konfiguriert, wobei auch die Konzeptfabriken aus der Konfiguration benötigt werden und daher ein zusätzlicher Konnektor notwendig ist. Gegenüber den Workflows und Root-Objekten wird die Schnittstelle im `MCInfrastructureProvider` erweitert, weil die Konzeptfabriken teilweise für die Überprüfung der Kontextbedingungen in Grammatiken, Sprachdateien und der Bundledatei verantwortlich sind. Die Grundfunktionalität, die Bereitstellung der notwendigen Methoden für die Workflows für die Verarbeitung der Root-Objekte, bleibt erhalten. Der Systemteil `IDSLToolParameters` verwaltet die Parameter des aktuellen DSLTools und bestimmt so die auszuführenden Workflows und deren Reihenfolge. Die Klasse `ADSLToolExecuter` führt entsprechend der Konfiguration und der Parameter die Workflows aus. Der `Modelloader` dient zum Nachladen von Modellen, die nicht direkte Eingabemodelle des Werkzeugs sind. Der `IDSLToolErrorDelegator` leitet die

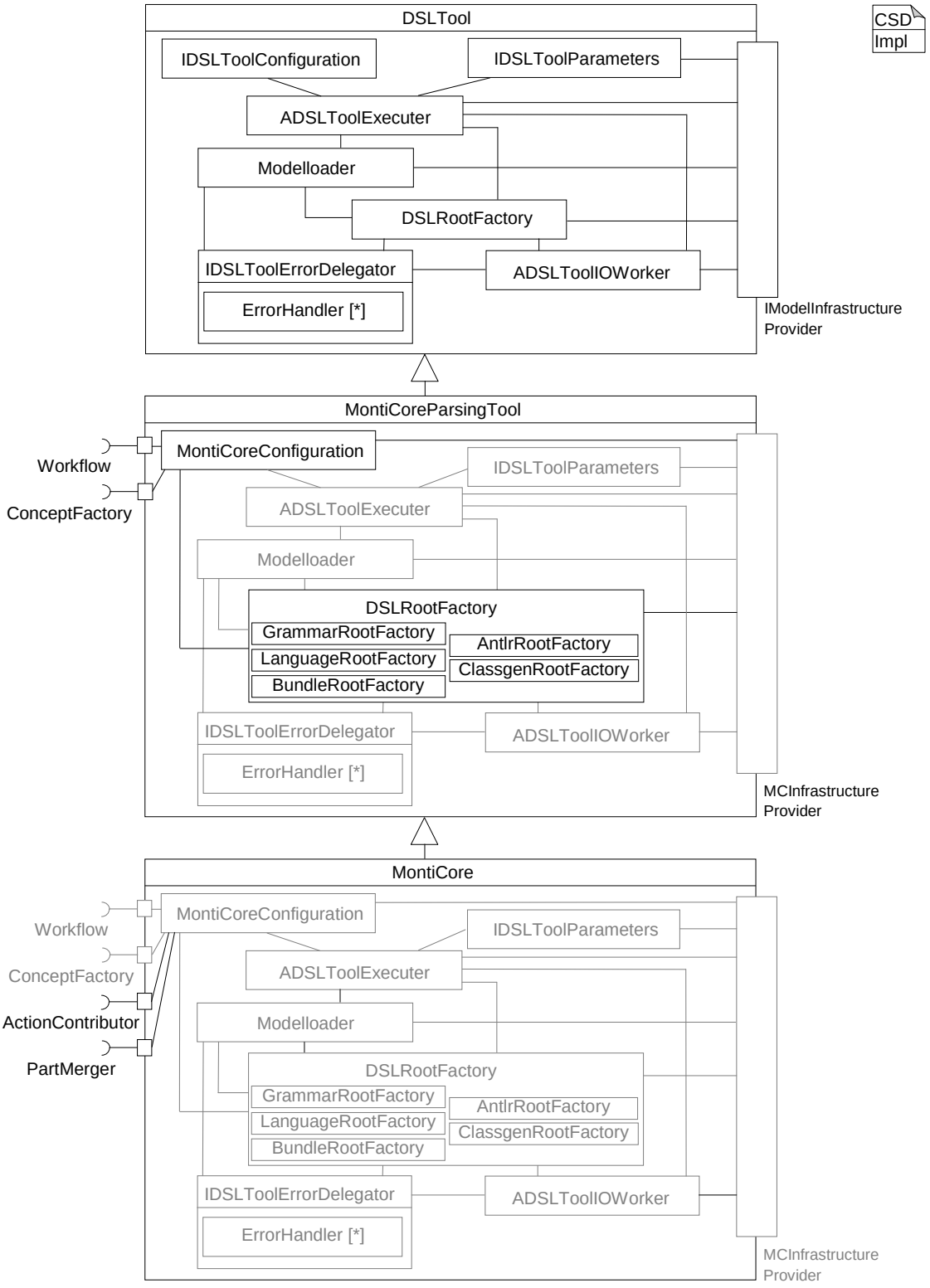


Abbildung 6.5: Zusammenhang der beiden Klassen `MontiCoreParsingTool` und `MontiCore` mit der Referenzarchitektur

Fehlermeldungen an die registrierten `ErrorHandler` weiter. Der `ADSLToolIOWorker` stellt eine Abstraktion des Dateisystems bereit, um dieses für Testfälle vom Werkzeug entkoppeln zu können.

6.3.1 Symboltabelle der Grammatik

Die Symboltabelle einer Grammatik hat innerhalb von MontiCore zwei Aufgaben. Erstens dient sie zur Überprüfung der Kontextbedingungen der Grammatiken, so dass aussagekräftige Fehlermeldungen bei fehlerhaften Eingaben für Sprachentwickler erzeugt werden können. Zweitens ermöglicht sie weitergehenden Verarbeitungsschritten und Konzepten, auf zentrale Analyseergebnisse zurückzugreifen. Insbesondere können die Auswirkungen der Sprachvererbung so zentral aufgelöst werden und transparent für den Entwickler von Erweiterungen zur Verfügung gestellt werden.

Die Überprüfung der Kontextbedingungen erfolgt schrittweise in sieben Phasen. Nach jeder Phase gilt eine bestimmte Klasse von Kontextbedingungen für die Grammatik als geprüft. Dieses erleichtert teilweise die Programmierung späterer Überprüfungen, weil dabei bereits einige Funktionalitäten der Symboltabelle genutzt werden können. Treten in einer Phase fatale Fehler auf, wird die Kontextbedingungsprüfung abgebrochen und keine weitere Analyse durchgeführt.

1. Laden der Symboltabellen der Obergrammatiken.
2. Hinzufügen von neuen Produktionen durch Konzepte.
3. Aufbau der Produktionen, Typen und Assoziationen in der Symboltabelle und Auswertung der Optionen.
   (a) Auswertung der Optionen der Grammatik.
   (b) Aufbau der Produktionen und Typen in der Symboltabelle für alle Arten von Produktionen der Grammatik.
   (c) Aufbau der Assoziationen.
   (d) Erzeugen der zusätzlichen Vererbungsbeziehungen aufgrund der Sprachvererbung und der Verschattung von Produktionen.
   (e) Modifikation der Elemente der Symboltabelle oder Einführung von neuen Elementen durch Konzepte.
4. Überprüfung der Produktionen, Typen und Assoziationen in der Symboltabelle.
   (a) Überprüfung aller Typ- und Produktionsreferenzen in der Grammatik.
   (b) Überprüfung aller Typ- und Produktionsreferenzen in den Konzepten.
5. Aufbau der Attribute innerhalb der Symboltabelle.
   (a) Aufbau der Attribute aus den Produktionen.
   (b) Aufbau der Attribute aus den Konzepten.
6. Überprüfung der Attribute in der Symboltabelle.
   (a) Überprüfung der Attribute auf Korrektheit.
   (b) Überprüfung der Attribute innerhalb der Konzepte auf Korrektheit.
7. Prüfung der Symboltabelle auf innere Konsistenz.

Nach erfolgreicher Durchführung der Phase 1 ist gesichert, dass alle Obergrammatiken wohldefiniert sind und deren Symboltabelle vollständig zur Verfügung steht. In Phase 2 können Konzepte weitere Produktionen zur Grammatik hinzufügen, als wären diese in der Grammatik ursprünglich notiert worden. Nach dieser Phase dürfen keine Modifikationen mehr am AST durchgeführt werden. Nach Phase 3 sind alle Typen, Produktionen und Assoziationen der Symboltabelle analysiert und dürfen nicht mehr verändert werden. In der Phase 4 werden diese auf Konsistenz geprüft und alle Vorkommen in der Grammatik und den Konzepten überprüft. In Phase 5 werden die Attribute aus den Produktionen berechnet und eventuell durch die Konzepte ergänzt oder modifiziert. Nach dieser Phase ist sichergestellt, dass die Attribute wohldefiniert sind, und dürfen nicht mehr verändert werden. Daraufhin werden in Phase 6 alle Vorkommen von Attributen in der Grammatik oder den Konzepten geprüft. Zum Schluss wird in Phase 7 die innere Konsistenz der Symboltabelle geprüft, so dass nach dieser Phase alle Kontextbedingungen geprüft werden.

Am Ende der Prüfung entsteht eine Datenstruktur, die die Typstruktur der Grammatik darstellt. Die Abbildung 6.6 stellt die wichtigsten Klassen als Klassendiagramm mit einer Auswahl der Methoden dar.

Eine Grammatik wird dabei durch eine Instanz vom Typ `GrammarRoot` repräsentiert. In dieser Datenstruktur wird der AST und die Symboltabelle abgelegt. Der AST ist hierbei durch die Klassen `MCCompilationUnit` und `ASTGrammar` dargestellt, wobei die weiteren Klassen ausgelassen sind. Die zentrale Datenstruktur der Symboltabelle ist die Klasse `STGrammar`, die eine Grammatik repräsentiert und mit einer Menge an Obergrammatiken bzw. deren Symboltabelle über die Assoziation `supergrammars` verbunden ist. Die Symboltabelle enthält zusätzlich einen angereicherten AST, der um zusätzliche Elemente zum Beispiel durch Konzepte und Optionen in den Schritten 2 und 3a des Symboltabellenaufbaus ergänzt wird. Eine Symboltabelle besteht aus einer Menge von Objekten vom Typ `STProd`, die eine Produktion einer Grammatik repräsentieren. Dabei existieren geeignete Untertypen für die verschiedenen Arten von Produktionen, also `STEnumProd` für Enumerationsproduktion und `STExternalProd` für externe Produktionen. Schnittstellen- und abstrakte Produktionen werden beide durch die Klasse `STInterfaceOrAbstractProd` behandelt, wobei beide über einen Produktionskörper verfügen können, der die standardmäßige Ableitung aus den existierenden Vererbungsbeziehungen überschreiben kann. Lexikalische Produktionen werden durch `STLexProd` repräsentiert, wobei diese implizit sein können, also nur in einer anderen Produktion durch Setzen des Tokentyps entstehen. Des Weiteren können sie `protected` sein, also nicht vom Lexer zurückgeliefert werden, sondern nur in anderen lexikalischen Produktionen genutzt werden. Klassenproduktionen schließlich werden durch Objekte vom Typ `STClassProd` repräsentiert.

Eng verwandt mit den Produktionen sind die daraus analysierten Typen, die durch Objekte vom Typ `STType` repräsentiert werden, wobei aufgrund der mehrteiligen Parserproduktionen die Zuordnung nicht eins-zu-eins ist und sich der definierte vom returnierten Typ unterscheiden kann. Typen können dabei als `external` gekennzeichnet werden, wenn die entsprechenden Typen nicht erzeugt werden sollen, sondern bereits existieren. Für Assoziationen und die Einbindung von Javatypen existieren mit `STAssociation` und `STExternalType` spezielle Unterklassen. Zusätzlich existiert für einen Typfehler eine spezielle Klasse `STUndefinedType`, die einen Symboltabellenaufbau mit nachträglicher Fehleranalyse erlaubt.

Typen können dabei Attribute besitzen, die durch `STAttribute` abgebildet werden und ihrerseits wiederum einen Typ besitzen. Dabei hat jedes Attribut eine Kardinalität, die durch Minimal- und Maximalwerte dargestellt wird. Zusätzlich kann zwischen komponierten Attributen wie Kompositionen und primitiven Attributtypen (Composition) und Assoziationenden (Association) unterschieden werden.

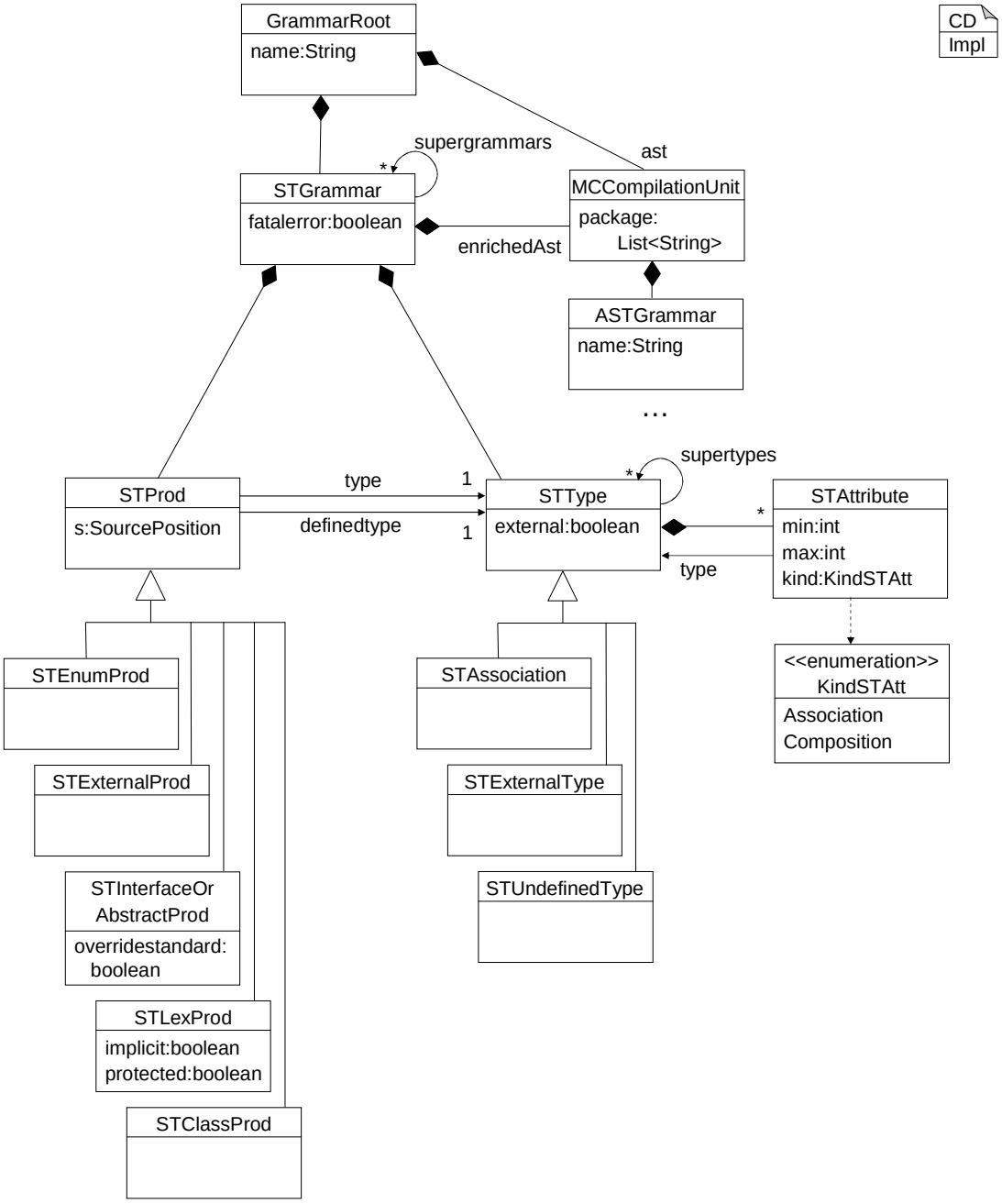


Abbildung 6.6: Klassendiagramm der Symboltabelle des MontiCore-Grammatikformats

6.4 Erweiterungsfähigkeit

Der MontiCore-Generator ist kein in sich geschlossenes Werkzeug, sondern kann gezielt erweitert werden.

6.4.1 Erweiterungspunkte von MontiCore

Das MontiCore-Werkzeug bietet fünf Erweiterungspunkte, an denen die jeweiligen Erweiterungen angefügt werden können. Eine Ausnahme stellt die Plattform dar, weil sie kein

eigenständiger Erweiterungspunkt ist, sondern bestimmte existierende Workflows parametrisieren kann. Diese werden über den Erweiterungspunkt für Workflows zum Werkzeug hinzugefügt.

Workflows Workflows stellen innerhalb des DSLTool-Frameworks (vergleiche Kapitel 9) zustandslose Algorithmen dar, die auf Eingabedateien ausgeführt werden. Die Parsergenerierung, Datenstrukturerzeugung, die Erzeugung von Hilfsdatenstrukturen und alle weiteren Generierungen innerhalb MontiCores sind in Workflows gekapselt. Der MontiCore-Generator führt standardmäßig für die von ihm verarbeiteten Sprachen die beiden Workflows *Parse* und *Generate* aus. *Parse* führt den eigentlichen Erkennungsprozess und den Aufbau geeigneter Symboltabellen durch. *Generate* hingegen erzeugt für die einzelnen Datentypen die daraus entstehenden Klassen und Dateien. Die Workflows sind dabei intern weiter in Teilschritte unterteilt.

Für die vom MontiCore-Generator verarbeiteten Sprachen gibt es die Möglichkeit, weitere Workflows zu programmieren und diese als Alternative zur normalen Verarbeitung oder zusätzlich auszuführen. Dadurch lassen sich weitere Dateien aus einer Sprachdefinition erzeugen, die den Einsatz innerhalb einer generativen Entwicklung vereinfachen.

Konzepte Das Grammatikformat, die Sprachdateien und die Bundledatei können gezielt erweitert werden, so dass neben der bereits beschriebenen Infrastruktur weitere Elemente generiert werden. Die Erweiterung der vorhandenen Notationen hat den Vorteil, dass spezifische Ergänzungen direkt in der Eingabedatei gemacht werden können und keine zusätzlichen Dateitypen notwendig sind. Ein Beispiel für diese Idee ist die Editorgenerierung aus der Sprachdefinition heraus. Hierzu sind neben der Sprachdefinition einige Informationen notwendig. Werden diese direkt in der Grammatik notiert, können sie in allen Sprachdateien verwendet werden, die dieses Fragment nutzen. Dadurch können mit der Sprachdefinition auch weitere Informationen zur Verfügung gestellt werden, die dann in einem anderen Kontext wieder verwendet werden können.

Unter einem Konzept wird also eine Spracherweiterung verstanden, die für Grammatiken, Sprach- und Bundledateien zur Verfügung steht. Diese kann jeweils auch nur für eine einzelne Dateiart oder auch mehrere definiert sein. Zentral für die Integration ist die Klasse `ConceptFactory`, die alle zu einem Konzept gehörigen Elemente in einer Klasse kapselt. Die Konzeptfabrik fasst jeweils die notwendigen Parser und die Überprüfung von Kontextbedingungen zusammen. Somit sind die Konzepte zum Beispiel in die Typanalyse der Grammatik mit eingebunden und können daher gezielt in den Schritten 2, 3e und 5b Elemente zur Symboltabelle hinzufügen und in den Schritten 4b und 6b eigene Überprüfungen durchführen.

Plattformen Die innerhalb von MontiCore erzeugten Parser und die verwendeten Datenstrukturen sind aufeinander abgestimmt. Die Definition des Grammatikformats erlaubt jedoch eine variable Umsetzung der Datenstrukturen. Dazu müssen in einer Subklasse von `PlatformContext` bestimmte Konstanten definiert werden, die die Codegenerierung innerhalb des Frameworks beschreiben. Mit dieser Klasse kann dann der `GrammarGenerateGenericWorkflow` parametrisiert und gegebenenfalls geeignet erweitert werden. Dadurch kann die Parsererzeugung weiter genutzt werden, auch wenn sich beispielsweise die Struktur der AST-Klassen ändert, wobei dann die verwendeten `ActionContributor` verändert werden müssen.

ActionContributor Die Erzeugung eines Parsers innerhalb des MontiCore-Frameworks ist soweit modularisiert, dass der Generator nur den Teil der Antlr-Grammatik erzeugt, der vom Parsergenerator zum Erkennen der Sprache benötigt wird. Weitere Teile, die zum Beispiel zum Aufbau des AST und Erkennen von Zeileninformationen notwendig sind, sind davon getrennt in `ActionContributor` spezifiziert.

PartMerger PartMerger dienen innerhalb der inkrementellen Codeerzeugung des DSLTool-Frameworks (vergleiche Abschnitt 9.3.5) dazu, Daten aus mehreren Quellen in einer Datei zusammenzufügen. Die Parts, die die Daten enthalten, müssen dabei von ebenfalls hinzuzufügenden Workflows erzeugt werden.

Die dargestellten Erweiterungspunkte von MontiCore lassen sich wie folgt nutzen:

1. Durch Ableiten der Klasse `mc.MontiCore` und überschreiben der Konfigurationsmethoden.
2. Durch Instanzierung der Klasse `mc.MontiCore` und Verwendung der Erweiterungspunkte.
3. Durch die Verwendung von Kommandozeilenparametern (nur für Workflows und ActionContributor).
4. Durch Verwendung eines Konfigurationsskripts in Zusammenhang mit dem Parameter `-conf` (nur für Workflows).

6.4.2 Vorhandene Erweiterungen

Dieser Abschnitt stellt einige der vorhandenen Erweiterungen des MontiCore-Generators dar. Bei der Auswahl wurde darauf geachtet, dass jeder Erweiterungspunkt einmal betrachtet wird, so dass die dargestellten Erweiterungen vor allem illustrieren sollen, welche Möglichkeiten einem Entwickler geboten werden. Die Konzeption und Entwicklung der Erweiterungspunkte von MontiCore wurden vom Autor selbst gestaltet, wohingegen dieses für Erweiterungen nur teilweise zutrifft.

EMF Innerhalb von [Hof08] wurde ein Workflow erarbeitet, der eine MontiCore-Grammatik auf ein EMF-Metamodell abbildet, indem eine passende Ecore-Datei und ein Generierungsmodell erzeugt werden. Die dabei entstehende Struktur des Metamodells basiert auf der Symboltabelle und entspricht daher den Darstellungen aus Kapitel 5, in dem die Ableitung der abstrakten Syntax formalisiert wurde. Zusätzlich wird ein Workflow generiert, der einen MontiCore-AST in das entsprechende EMF-Modell überführt.

Instanzerzeugung Die Instanzerzeugung für Grammatiken [Sah06] generiert aus einer Grammatik Eingabedateien, die zur Grammatik konform sind, also beim Parsen zu wohlgeformten ASTs führen. Dabei wurden zwei Workflows implementiert, die garantieren, dass die Instanzmenge verschiedene Abdeckungskriterien [Pur72, Läm01b] für Grammatiken erfüllt.

SReference Das Konzept *sreference* dient zur Implementierung der Schnittstellen, die aus den Assoziationen einer MontiCore-Grammatik erzeugt werden. Die grundlegende Idee ist dabei, dass sich Objekte innerhalb des AST teilweise eindeutig durch ein Attribut vom Typ `String` identifizieren lassen. Durch das Konzept wird ein Link passend zu einer angegebenen Assoziation zwischen zwei Objekten hergestellt, wenn

der Wert eines angegebenen Attributs der ersten Klasse und der Wert eines Attributs der zweiten Klasse der Assoziation übereinstimmen.

Attribute Attributgrammatiken [Knu68] bezeichnen einen Formalismus, der Datenabhängigkeiten innerhalb einer Grammatik spezifizieren kann. Dabei werden Attribute von Nichtterminalen entweder durch Attribute der entsprechenden Knoten oberhalb im AST (vererbt) oder durch abhängige Knoten (synthetisiert) bestimmt. MontiCore stellt einen einfachen Attributgrammatikformalismus [Ste08] zur Verfügung, der sich an der Realisierung in JastAdd [EH07] orientiert, da die entsprechenden Attributberechnungen mit Java auf der abstrakten Syntax basieren. Die Berechnung ist dabei *lazy*, das heißt, die Werte werden erst berechnet, wenn auf ein Attribut zugegriffen wird. Daher muss keine spezielle Auswertungsreihenfolge eingehalten werden, da die Berechnung für nicht-zyklische Grammatiken stets terminiert und für zyklische Grammatiken die Terminierung durch eine Markierung der Objekte garantiert werden kann [Hed89].

Ein besonderes Augenmerk bei der Realisierung lag auf der Unterstützung der Modularität des MontiCore-Frameworks. Es können Attribute und Attributberechnungen für eine Grammatik spezifiziert werden und dann innerhalb einer Sprachkombination übernommen werden. Dabei sind insbesondere Namensänderungen möglich, um unerwünschte Kollisionen zu vermeiden oder semantisch gleichbedeutende Attribute zusammenzuführen. Der Mechanismus wurde dabei so umgesetzt, dass eine direkte Implementierung der Attributberechnungen in Java möglich ist und keine Beschreibung in einer DSL notwendig wird, um die Unterstützung durch die Entwicklungsumgebung bei der Implementierung der Attribute nutzen zu können.

POJO-Plattform Die POJO-Plattform (Plain Old Java Objects) erlaubt die Erzeugung von AST-Klassen und eines dazugehörigen Parsers, die nicht von zusätzlicher MontiCore-Infrastruktur abhängen, sondern aus einfachen Java-Klassen bestehen. Die Plattform verwendet dabei den Plattformkontext `POJOPlatformContext` zur Spezifikation der Konstanten und den Workflow `GrammarGeneratePOJOWorkflow` zur Erzeugung der notwendigen Artefakte.

Abdeckungskriterien Innerhalb des Workflows `CoverageWorkflow` werden ActionContributors verwendet, um den Parser geeignet zu instrumentalisieren. Dabei entsteht ein alternativer Parser, der eingesetzt werden kann, um Eingabedateien zu verarbeiten und den üblichen AST zu erzeugen. Zusätzlich wird eine Datenstruktur erzeugt, um die Erfüllung verschiedener Abdeckungskriterien [Pur72, Läm01b] für Grammatiken zu berechnen.

Editorgenerierung Die Editorgenerierung innerhalb des MontiCore-Frameworks erlaubt die Erstellung von Editoren, die eine komfortable Eingabe textueller Modelle innerhalb des Eclipse-Frameworks ermöglichen [KRV07a]. Dabei werden die notwendigen Zusatzinformationen durch ein Konzept gekapselt, das innerhalb von Sprachdateien und Grammatiken eingesetzt werden kann. Die Spezifikation innerhalb der Grammatik erlaubt die Nutzung der Informationen, wenn die Fragmente mittels einer Sprachdatei kombiniert werden, ohne dass eine erneute Angabe der Informationen notwendig ist.

Innerhalb der Editorgenerierung wird neben den Konzepten und den Workflows zur Erzeugung der Editoren auch ein `PartMerger` eingesetzt. Die Verwendung ist notwendig, weil die Editoren in einer Datei `plugin.xml` aufgeführt werden müssen, damit

diese in das Eclipse-Framework integriert werden können. Daher erzeugt die Editorgenerierung, sofern das Konzept verwendet wird, einzelne Part-Objekte, die dann im `PartMerger` zu einer Datei `plugin.xml` zusammengefasst werden.

6.5 Einsatz von MontiCore

Der Build-Prozess für MontiCore-Projekte ist vollständig über Ant-Skripte [Mat05] automatisierbar. Eine genaue Projektorganisation ist dabei nicht vorgeschrieben, vereinfacht jedoch die Entwicklung und erlaubt eine hohe Kompatibilität und Unterstützung durch MontiCore. Insbesondere wird die Einarbeitungszeit der Entwickler in ein neues Projekt verkürzt.

Eine besondere Schwierigkeit innerhalb der Entwicklung mit MontiCore ist, dass Module andere Module als Generator benutzen. In der normalen Softwareentwicklung müssen Module einer Software nur dann neu kompiliert werden, wenn Veränderungen an den Modulen selbst auftreten. Bei der generativen Entwicklung ist eine Neukompilierung aber auch dann notwendig, wenn sich Generatoren verändern, die in einem Modul eingesetzt werden. Die Ant-Skripte analysieren daher, ob sich eingesetzte Generatoren verändert haben und erzeugen nötigenfalls die generierten Klassen neu aus den Modellen.

Ein Projekt besteht dabei grundsätzlich aus mehreren Ordnern, in denen sich unterschiedliche Arten von Dateien befinden. Die Abbildung 6.7 zeigt die einzelnen Projekte und Unterorder als Pakete in einem UML-Komponentendiagramm. Im Ordner `src` befinden sich handcodierte Java-Dateien, wohingegen im Ordner `def` Modelle abgelegt werden, die von

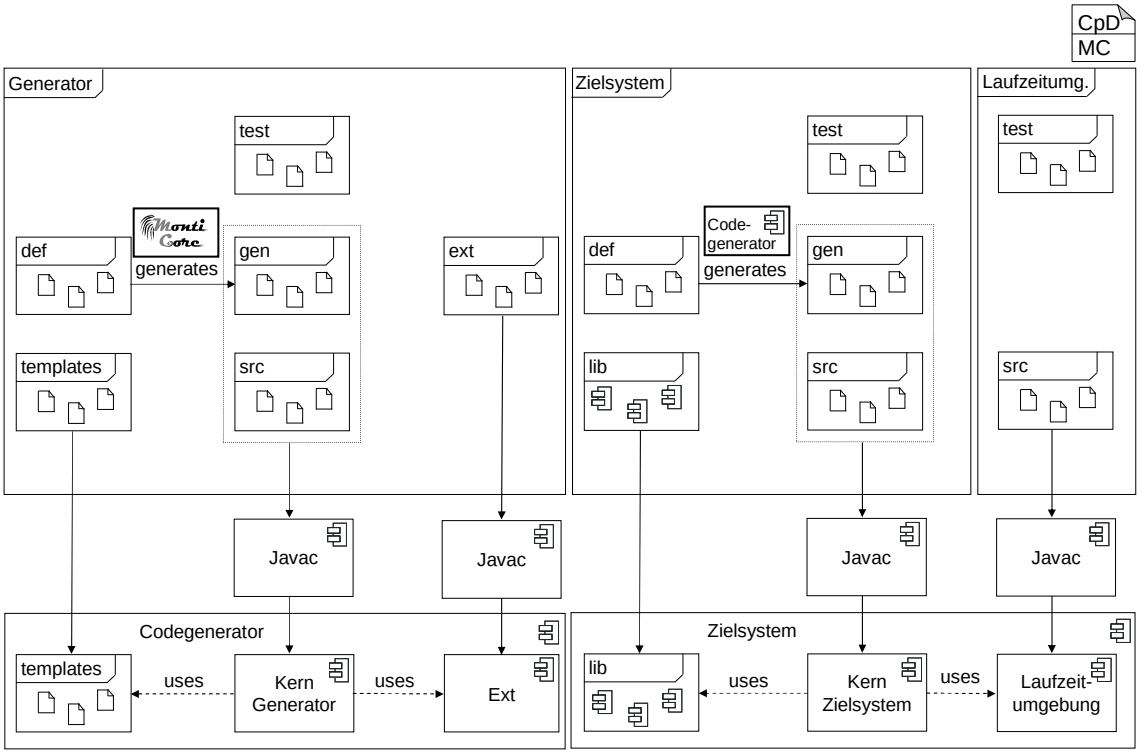


Abbildung 6.7: Komponentendiagramm eines Generators, der dazugehörigen Laufzeitumgebung und des Zielsystem.

Codegeneratoren in den Java-Dateien umgewandelt und im Ordner `gen` abgelegt werden. Die Testfälle zur Qualitätssicherung eines Projekts befinden sich im Ordner `test`. Sind für ein Projekt Bibliotheken notwendig, sollen diese im Order `ext` als Quelldateien oder im Order `lib` als Archive abgelegt werden. Handelt es sich bei diesem Projekt um einen Generator können sich im Ordner `templates` Templates für die Codegenerierung befinden. Bei der Organisation ist es wichtig, Generatoren, Laufzeitumgebungen, die den generierten Code ergänzen, und das Zielsystem, für das der Generator bei der Entwicklung eingesetzt wird, voneinander zu trennen. Die Abbildung zeigt die empfohlene Verzeichnisorganisation eines fiktiven Generators, eines Zielsystems und einer Laufzeitumgebung. Dabei könnten alle drei Projekte über `ext`- und `lib`-Ordner verfügen, was jedoch aus Übersichtsgründen vermieden wurde. Der Einsatz von mehreren Generatoren in einem Zielsystem und die Verwendung eines Generators für verschiedene Zielsysteme sind erwünscht und realisierbar.

6.6 Zusammenfassung

Dieses Kapitel hat die wesentlichen Funktionen des Frameworks MontiCore aufgezeigt, indem die Eingabesprachen und die darauf basierenden Analyse- und Generierungsschritte beschrieben wurden. Für die generierten Datentypen wurden die Eigenschaften beschrieben, auf die sich ein Entwickler verlassen kann und die auch in zukünftigen MontiCore-Versionen gepflegt werden.

Die Aufteilung des Gesamtsystems in Komponenten wurde gewählt, damit die Software mit unterschiedlichen Konfigurationen betrieben werden kann. Dabei wurden die unterschiedlichen Konfigurationen für die zwei Ausführungsmodi, also mit installiertem Generator oder über den OSTP-Server, und die nutzbaren Plattformen, also Eclipse und die Kommandozeile, erläutert. Die einzelnen Komponenten werden dabei in einem partiellen Bootstrappingprozess entwickelt.

Die Architektur des Werkzeugs basiert auf der Referenzarchitektur des DSLTool-Frameworks, das sich zur Realisierung von generativen Werkzeugen eignet und in Kapitel 9 noch näher beschrieben wird. Dabei wurde ein besonderer Schwerpunkt auf die Erweiterungs- und Anpassungsfähigkeit gelegt. Die Erweiterungspunkte wurden in die existierenden Analysen und Codegenerierungen integriert, so dass eine gemeinsame Infrastruktur genutzt werden kann.

Kapitel 7

Verwandte Arbeiten zu MontiCore

In diesem Kapitel sind verwandte Arbeiten zuammengestellt, die eine große Ähnlichkeit zum Grammatikformat und MontiCore aufweisen. Zunächst werden Ansätze betrachtet, die die abstrakte Syntax aus der konkreten ableiten. Daraufhin werden modulare Definitionsformen für Modellierungsformen betrachtet und Ansätze dargestellt, die für eine solche modellbasierte abstrakte Syntax die konkrete Syntax modellieren. Es werden mehrere Formen der Modularität von textuellen Sprachen und Grammatiken betrachtet und der Einsatz von Grammatiken zur Beschreibung von Komponentenschnittstellen, da diese Form der Schnittstellenbeschreibung auch in MontiCore häufig eingesetzt wird. Abschließend werden so genannte Sprachbaukästen und Metamodellierungs-Frameworks beschrieben, die die Entwicklung eigenständiger DSLs ermöglichen.

7.1 Ableitung der abstrakten aus der konkreten Syntax

MontiCore verwendet ein erweitertes Grammatikformat, das ausreichend ist, um die abstrakte Syntax einer Sprache zu definieren. Da die zugrunde liegende ENBF-Notation oft zur Modellierung der konkreten Syntax verwendet wird, werden im Folgenden Ansätze betrachtet, die die abstrakte Syntax aus der konkreten ableiten.

In [WW93] wird die Object-Oriented Normal Form (ONF), eine Submenge der EBNF, dargestellt. Die ONF verwendet ausschließlich Produktionen, die nur aus alternativen Nichtterminalen bestehen, oder aber Produktionen, die keine Alternativen enthalten. Mischformen sind nicht erlaubt. Aus den alternativen Produktionen werden Vererbungsbeziehungen in der abstrakten Syntax abgeleitet.

Die Ableitung der abstrakten aus der konkreten Syntax wird auch in [Wil97a, Wil97b] dargestellt, wobei eine Realisierung in der funktionalen Programmiersprache LISP im Popart-System [Wil94] existiert und eine Übersetzung nach C++ beschrieben wird. Eine Benennung von Nichtterminalen innerhalb einer Produktion ist möglich, um gezielt die Namen der Attribute zu beeinflussen. Spezialkonstrukte erlauben die Realisierung flacher Bäume für Ausdrücke einer Programmiersprache.

Ein neuerer Ansatz [HN05] beschreibt die Ableitung der abstrakten Syntax aus einer Submenge der EBNF, die *Generalized Object Normal Form* (GONF) genannt wird. Die GONF erlaubt nicht den gemischten Einsatz von Alternativen und Sequenzen in einer Produktion, verwendet jedoch optionale Elemente und den kleeneschen Stern. Es gibt keine Möglichkeiten, die Attribute gezielt zu benennen.

Der Parser-Generator SableCC [GH98] verwendet ein sehr ähnliches Format zur GONF. Im Gegensatz zu den anderen Ansätzen entstehen für die alternativen Produktionen im AST wirkliche Zwischenknoten, was insgesamt zu einer komplexen Datenstruktur führt.

In [KW96] wird ein Verfahren beschrieben, zwei Arten von Grammatiken für konkrete und abstrakte Syntax zu verwenden, wobei sich die Informationen jedoch ergänzen und zusätzlich Heuristiken zur Ergänzung fehlender Informationen verwendet werden.

In [AP03] werden zwei Abbildungen vorgestellt, die Grammatiken und Metamodelle aufeinander abbilden. Dabei wird eine reine EBNF-Notation verwendet, so dass die entstehenden Metamodelle ohne sprechende Bezeichner für Attribute und Assoziationen ohne weitere manuelle Bearbeitung kaum nutzbar sind.

In [WK05] wird ein Verfahren beschrieben, das aus einer EBNF-Grammatik ein Metamodell extrahiert. Die Ableitung erfolgt sehr technisch, so dass zunächst ein sehr komplexes Metamodell entsteht, das dann aufgrund von Regeln wiederum vereinfacht wird.

Das MontiCore-Grammatikformat orientiert sich an der in den Ansätzen dargestellten Ableitung der abstrakten Syntax. Dabei wird insbesondere von [Wil97a] die Realisierung der alternativen Produktionen durch Schnittstellen übernommen. Es existieren andersartige Konstrukte zur effektiven Gestaltung von flachen Ausdrucksbäumen, die jedoch zum selben Ergebnis führen. Im Gegensatz zu anderen Ansätzen werden in MontiCore Vererbungsbeziehungen nicht automatisch erstellt, sondern sind durch den Benutzer gezielt zu beeinflussen. Zusätzlich kann er zwischen Schnittstellen und Klassen unterscheiden. Die Benennung von Nichtterminalen innerhalb einer Produktion ist möglich, um gezielt die Attributwerte zu beeinflussen, wobei nur MontiCore eine doppelte Verwendung von Bezeichnern erlaubt, um komplexere Listenstrukturen zu erzeugen. Hierfür ist kein Spezialkonstrukt wie zum Beispiel in [Wil97a] notwendig, welches insgesamt weniger flexibel als die doppelte Verwendung von Namen wie im MontiCore-Grammatikformat ist. Alle dargestellten Ansätze zur automatischen Ableitung der abstrakten aus der konkreten Syntax umfassen immer nur Baumstrukturen und nutzen kein zu Assoziationen äquivalentes Konzept, wie es in der Metamodellierung üblich ist. Das MontiCore-Grammatikformat hingegen wurde als integriertes Format entwickelt, um solche Beziehungen direkt spezifizieren zu können.

7.2 Modulare Sprachdefinitionen von Modellierungssprachen

Modellbasierte Ansätze beschreiben die abstrakte Syntax einer Sprache mittels eines so genannten Metamodells [JB06]. Dabei handelt es sich um eine Variante von Klassendiagrammen, also eine Strukturbeschreibung, die zusätzlich zu den Vererbungs- und Kompositionsbeziehungen auch noch Assoziationen enthält und somit einen Graph aufspannt. In der Metamodellierung mit der Meta Object Facilty [OMG06b] ist es möglich, gezielt in Metamodelle weitere andere Metamodelle durch so genannte *Imports* einzubinden. Des Weiteren können Metamodelle gezielt durch *Package Merges* erweitert werden. Dieses Vorgehen wird auch von auf MOF basierenden Werkzeugen unterstützt [AKRS06]. In [WS08] wird ein Modulkonzept beschrieben, das die Definition von Sprachkomponenten erlaubt und so die modulare Sprachdefinition mit MOF ermöglicht. Dabei können explizit Internata der einzelnen Komponenten nach außen versteckt werden und klare Schnittstellen der Metamodelle verwendet werden.

Das EMF bietet mit seiner Metamodellierungssprache Ecore Möglichkeiten zum *Import* und mit Hilfe des Generierungsmodells auch eine Realisierung ähnlich dem *Package Merge* an. Die beiden Metamodellierungstechnologien verwenden unterschiedliche Arten von Assoziationen, wobei EMF die Assoziationen als gegenseitige Referenzen und MOF als eigenständige Elemente betrachtet, die auf unterschiedliche Arten verfeinert werden können.

MontiCore erlaubt mit der Spracheinbettung die lose gekoppelte Komposition von Modellierungssprachen und bietet mit der Sprachvererbung ein Konzept zur Erweiterung existierender Sprachen. Beide Konzepte dienen zur Modularisierung der Sprachdefinition und orientieren sich an den *Imports* und *Package Merges* der MOF, wobei sich aufgrund der integrierten Definition von konkreter und abstrakter Syntax weitere Einschränkungen ergeben.

7.3 Modellierungssprachen zur Beschreibung der konkreten Syntax

Um die konkrete Syntax von Modellierungssprachen zu beschreiben, wurden Sprachen entwickelt, die die Modellierung der konkreten Syntax für eine gegebene abstrakte Syntax in Form eines Metamodells erlauben [JBK06, MFF+06]. Eine vergleichende Studie einzelner Ansätze findet sich auch in [GBU08]. Dazu werden für jede Klasse des Metamodells Schablonen definiert, die einerseits die Erzeugung eines Parsers und andererseits eines Unparsers erlauben. Die Sprache xText wird innerhalb des Frameworks OpenArchitectureWare [OAW] dazu verwendet, um textuelle Sprachen zu definieren. Eine automatische Erzeugung eines Unparsers ist nicht möglich. Die OMG veröffentlichte mit der Human-Usable Textual Notation [OMG04] einen Standard der eine generische textuelle Syntax für Modellierungssprachen bietet, die von der Nutzbarkeit jedoch nicht an explizit modellierte konkrete Syntaxen heranreicht. Die existierende Implementierung [RPKP08] unterstützt jedoch noch nicht den gesamten Standard.

Die dargestellten modellbasierten Ansätze zur Modellierung der konkreten Syntax verlangen teilweise die Existenz einer expliziten abstrakten Syntax, was wiederum die bereits dargestellten Redundanzprobleme mit sich bringt. Einzig xText erlaubt auch eine Ableitung der abstrakten Syntax. Die Ansätze erlauben es nicht, flache Ausdrucksbäume ohne Zwischenknoten zu erzeugen. Keiner der Ansätze unterstützt direkt die Modularitätskonzepte, die die zugrunde liegende Beschreibung der abstrakten Syntax in Form von *Imports* und *Package Merges* bietet. Das MontiCore-Grammatikformat verwendet direkt in der Sprachdefinition dieselben Konzepte wie Metamodellierung, unterstützt jedoch mit Sprachvererbung und –einbettung zusätzlich auch die modulare Sprachdefinition.

7.4 Modulare Grammatiken

Im einflussreichen Artikel [KLV05] wird der Begriff *grammarware* zur Beschreibung von Software geprägt, die auf Sprachen basiert. Dabei wird der Begriff des Grammatikfragments als eine kontextfreie Grammatik formalisiert, die keine explizite Startproduktion verwendet und nicht definierte Nichtterminale verwenden kann. Diese Veränderung trägt dazu bei, Grammatiken als Artefakte im Softwareentwicklungsprozess zu verstehen, die ingenieurmäßig erstellt werden. Diese Begriffsbildung wurde für das MontiCore-Grammatikformat übernommen und bildet die Grundlage für die kompositionale Entwicklung.

Shuly Wintner beschreibt in [Win02] eine komplett abstrakte (fully abstract) Semantik von kontextfreien Grammatiken. Er beschreibt dabei die Schnittstelle einer Grammatik, wobei die Nichtterminale in exportierte, importierte und interne unterteilt werden. MontiCore greift diese Idee auf, verwendet jedoch keine internen Nichtterminale, weil somit jede Regel extern wieder verwendet werden kann, was im Allgemeinen nicht bereits geplante, sondern erst zum Verwendungszeitpunkt gewünschte Wiederverwendung fördert. Im Gegensatz zu Wintners Ansatz ist die Struktur der Regeln aufgrund der damit verbundenen

Struktur des ASTs von Bedeutung und hat somit Auswirkungen auf die Semantik. Somit kann in MontiCore nicht vom Regelaufbau wie in [Win02] abstrahiert werden.

Robert Adams beschreibt in seiner Dissertation [Ada91] die fein granulare Wiederverwendung von Grammatikregeln, die eine Modularisierung der Grammatik erlaubt. Die einzelnen Bausteine werden durch Nichtterminale parametrisiert und können so in eigenen Grammatiken eingesetzt werden. Die Wiederverwendung innerhalb dieses Ansatzes beschränkt sich auf Muster, die im Sinne eines Preprozessors innerhalb einer Grammatik ersetzt werden. In MontiCore wird eher versucht größere Sprachfragmente zusammen mit darauf aufbauender Infrastruktur zu kapseln und so strukturiert wieder verwenden zu können.

Thomas Cleenewerk beschreibt in seiner Dissertation [Cle07] so genannte Linglets, welche relativ fein granulare Fragmente sind, die dann wiederum zu gesamten Sprachen kombiniert werden können. Die Linglets definieren die konkrete und abstrakte Syntax und verfügen über ein Meta Object Protocol, mit dem sie untereinander Informationen austauschen können. MontiCore verwendet kein eigenes MOP, sondern die Programmiersprache Java, um auf der abstrakten Syntax zu navigieren. So wird eine leichtere Integration der Sprachdefinition mit der Implementierung komplexer Codegenerierungen ermöglicht.

7.5 Modulare Parsealgorithmen

Kontextfreie Grammatiken sind bezüglich der natürlichen Komposition (Vereinigung der Produktionsmengen) abgeschlossen und auch die Varianten von Adams, Wintner und Lämmel zeichnen sich durch diese Eigenschaft aus. Dieselbe Eigenschaft gilt auch für kontextfreie Sprachen. Die durch die Komposition zweier Grammatiken ableitbare Sprache kann wie von Wintner [Win02] thematisiert mehr Wörter umfassen, als die Vereinigungsmenge der Sprachen, was sich durch die Wahl der Schnittstellen und somit der Form der Komposition steuern lässt.

Für die eindeutige Erkennung von Sprachen wird typischerweise immer nur eine Submenge der kontextfreien Sprachen wie LL(1) und LR(1) verwendet, die jedoch nicht unter der Vereinigung abgeschlossen sind. Es kommt das praktische Problem hinzu, dass die Komposition durch die zweistufige Trennung in Lexer und Parser erschwert wird, da insbesondere Lexer nicht kompositional sind. Es existieren Parsealgorithmen, die die Kompositionalität dadurch erreichen, dass auf einen expliziten Lexer verzichtet wird. Dieses als *Scannerless Parsing* bezeichnete Verfahren geht auf [SC89] zurück und kann mit verschiedenen Parsealgorithmen kombiniert werden. Eine Alternative stellt das *Context-Aware Scanning* [WS07] dar, bei dem der verwendete Parser die Menge der validen Lexeme für den Lexer einschränkt und so die Auswahl vereinfacht. Die Technik ist jedoch zum derzeitigen Stand noch wenig verbreitet und es ist unklar, wo die Grenzen des Verfahrens liegen.

Parsealgorithmen wie [Ear70, Vis97] verwalten eine Menge an validen Parsebäumen, die fortlaufend erweitert und reduziert wird. Bei der Reduzierung, also der Entfernung von Parsebäumen aus der Menge, können Prioritäten oder sogar Typanalysen verwendet werden [BVVV05]. Insbesondere akkurate Fehlermeldungen und der Umgang mit mehreren validen Parsebäumen nach Ende der Erkennungsphase sind jedoch noch wenig erforscht.

Packrat-Parsing [For02] basiert auf so genannten Expression-Grammatiken, in denen Alternativen explizit geordnet sind und zum Erkennen Backtracking eingesetzt wird. Eine gezielte Erhöhung des Speicherbedarfs erlaubt hier eine Reduktion auf lineare Laufzeit.

Jan Bosch beschreibt in [Bos97] so genannte Delegating Compiler Objects (DCOs). DCOs erlauben den Wechsel von Lexern, Parsern oder des aktuellen DCOs, gesteuert aus

einen Parser heraus. Dieses entspricht in MontiCore dem Wechsel von einer Sprache in eine andere bei der verwendeten Spracheinbettung.

Das für MontiCore verwendete Parseverfahren basiert auf einem LL(k)-Parser mit einem expliziten Lexer. Die Einschränkungen durch die Verwendung von LL(k) relativieren sich durch den Einsatz von semantischen und syntaktischen Attributen, wie Antlr [PQ95] sie bietet. Das Verfahren ähnelt den DCOs stark, verwendet jedoch mit LL(k) eine andere Parsetechnologie als die dort verwendete LR(1)-Technik. Insbesondere die Backtracking-Fähigkeiten des LL-Parsers erlauben eine größere Flexibilität beim Übergang zwischen zwei Sprachen, weil auch Vorrangsregeln genutzt werden können. Die Möglichkeit, mehrere DCOs abhängig von den bereits geparsten Eingabedaten auszuwählen, wurde übernommen. Es existieren in MontiCore jedoch mit dem Zugriff auf Attribute und globale Stacks mehr Möglichkeiten als bei DCOs. Von der Verwendung moderner Parsealgorithmen ohne Lexer wurde abgesehen, weil sie in Bezug auf Fehlermeldungen schlechtere Ergebnisse liefern. Zusätzlich ist die Unterstützung der Entwickler in Bezug auf Debugging der entstehenden Parser und statische Analysen bezüglich Mehrdeutigkeiten der Grammatik deutlich besser. Diese Unterstützung ist bei der Entwicklung von Sprachen essentiell, da die Sprachdefinition wie jedes Artefakt im Entwicklungsprozess fehlerhaft sein kann und nur so mit vertretbarem Aufwand ein qualitativ hochwertiges Ergebnis zu erreichen ist.

7.6 Modulare Attributgrammatiken

Attributgrammatiken [Knu68] bezeichnen einen Formalismus, der es erlaubt, zusätzlich zur kontextfreien Syntax Attribute für Nichtterminale zu spezifizieren. Diese bezeichnen zusätzliche Werte, die aus der Baumstruktur und den darin enthaltenen Daten berechnet werden.

Es haben sich Abwandlungen herausgebildet. Besonders interessant zur Realisierung von eingebetteten DSLs ist die Technik des *Forwardings* [WMBK02] in *Higher Order Attribute Grammars* [VSK89, Vog93] innerhalb des MELT-Ansatzes [Gao07, VBKG07]. Die Technik des Forwardings stellt dabei die technische Grundlage für das *Intentional Programming* [SCC06] dar, welches die Realisierung und Integration unterschiedlicher DSLs ermöglicht. Eine alternative Technologie sind die *Referenced Attribute Grammars* (RAG) [Hed00], denen ein objekt-orientiertes Paradigma zu Grunde liegt. Eine Weiterentwicklung, die eine Umstrukturierung des ASTs erlaubt, sind die *Rewriteable Referenced Attribute Grammars* innerhalb des Werkzeugs JastAdd [EH07]. Das Werkzeug (Aspect) Lisa [MŽLA99, RMHV06] erlaubt die Definition von Attributgrammatiken unter der Nutzung von Sprachvererbung.

MontiCore bietet eine zu Lisa ähnliche Art der Grammatikvererbung, die zusätzlich die abstrakte Syntax mit einschließt. Ein Attributgrammatikmechanismus ähnlich zu den RAGs wurde in MontiCore als ein Konzept realisiert. Dieses Konzept kann in MontiCore als eine Möglichkeit zur Namens- und Typanalyse eingesetzt werden (vgl. Kapitel 6). Modularität bei Attributgrammatiken bezieht sich vor allem auf die unabhängige Definition von Gruppen von Attributen, die auch in MontiCore erreicht wird (vgl. [KW94]). Dadurch können Teile der Spezifikation, wenn Attribute zum Beispiel zur Festlegung der Typanalyse genutzt werden, getrennt von anderen wieder verwendet werden.

7.7 Sprache als Komponentenschnittstelle

Der von Don Batory als GenVoca [BLS98] skizzierte Ansatz zum Einsatz von DSLs bezeichnet die Hintereinanderausführung von Generatoren. Die Schnittstelle der Generatoren ist jeweils eine Eingabe- und eine Ausgabesprache. Somit lassen sich die Generatoren miteinander kombinieren, wenn die jeweiligen Sprachen passend sind. Ähnlich werden in [JV00] Grammatiken als Komponentenschnittstelle in der komponentenorientierten Softwarentwicklung eingesetzt.

In MontiCore wurde dieser Gedanke bei der Realisierung des Frameworks aufgegriffen, indem die Gestaltung der Grammatik selbst mit Konzepten modular gestaltet ist und so die Kernsprache gezielt erweitert werden kann. Intern bilden einzelne Sprachen die Schnittstelle zwischen Systemteilen.

7.8 Grammatik-basierte Sprachbaukästen

Es existiert eine Vielzahl an Sprachbaukästen, die aus einer Sprachdefinition Werkzeuge wie einen Editor, Interpreter, Codegeneratoren und Prettyprinter erzeugen. Die folgende Auflistung der Ansätze umfasst die am häufigsten zitierten Werkzeuge.

Mentor [DHKL84] erlaubt die Spezifikation einer Programmiersprache mittels verschiedener Spezifikationssprachen. Centaur [BCD+89] stellt eine Weiterentwicklung zum Teil derselben Autoren dar. Sie erlauben die Spezifikation einer Sprache durch natürliche Semantik [Kah87] unter Verwendung der Sprache Typol [Des84]. Die Spezifikation und die Programme werden auf Prolog abgebildet, was eine Überprüfung der Typisierung erlaubt, sich jedoch durch ein schlechtes Laufzeitverhalten auszeichnet.

Beim Programming System Generator [BS86] werden Kontext-Relationen [Sne91] verwendet, um die Typinferenzregeln einer Sprache zu definieren. Im Vergleich zur natürlichen Semantik sind Kontext-Relationen weniger flexibel, haben nicht die Möglichkeit, Vorwärts-Referenzen in Programmiersprachen zu spezifizieren und eignen sich weniger zur Spezifikation der dynamischen Semantik einer Sprache, können dafür aber unvollständige Programme evaluieren [GZ04].

Der Cornell Program Synthesizer [TR81] bezeichnet ein Werkzeug zur syntaxgetriebenen Eingabe von Programmen. Seine Weiterentwicklung, der Synthesizer Generator [RT84, RT89], erlaubt eine solche Entwicklungsumgebung für neue Sprachen zu generieren. Dabei werden Attributgrammatiken [Knu68] genutzt, um Kontextbedingungen der Sprache zu formulieren. In [HT86] wird dieser Ansatz mit relationalen Algebren kombiniert.

Popart [Wil94] bezeichnet ein Werkzeug zur Erstellung von Parsern. Zusätzlich wird eine Lisp-Bibliothek verwendet, um die Typanalyse der Programmiersprache einfach definieren zu können.

IPSEN [Nag96, KS97] bezeichnet einen Ansatz, eng gekoppelte Softwareentwicklungsumgebungen für Sprachen zu erzeugen. Insbesondere zeichnet sich dieser Ansatz durch die Integration des Graphersetzungssystems PROGRES [Sch90] und die Möglichkeit aus, sowohl textuelle als auch grafische DSLs verarbeiten zu können.

ASF+SDF [BHD+01] ist ein Compilerframework, das die Gestaltung beliebiger Sprachen erlaubt. Es verwendet dabei den Formalismus Abstract Specification Formalism (ASF) zur Spezifikation der abstrakten Syntax und Bestimmungsgleichungen zur Formulierung der Semantik der Sprache. Zusätzlich erlaubt der Syntax Definition Formalism (SDF) die Spezifikation der konkreten Syntax und die kanonische Abbildung auf ASF-Datenstrukturen. Die Werkzeuge MetaBorg [BV04] und StringBorg [BDV07] basieren auf dem Framework ASF+SDF und der Rewrite-Engine Stratego/XT [BKVV08]. Sie erlauben die Realisierung

von modular entworfenen eingebetteten DSLs, indem die DSL-Modelle nach dem Parsen an die Hostsprache abgebildet werden. Das Meta-Programming-System [MPS] integriert textuelle DSLs und syntax-getriebene Editoren in eine Java-Entwicklungsumgebung.

MontiCore fokussiert sich wie einige der anderen Ansätze auch auf rein textuelle DSLs. Die Erzeugung von integrierten Entwicklungsumgebungen steht dabei nicht im Vordergrund, sondern eher die effiziente Entwicklung von generativen und analytischen Werkzeugen. Zur Unterstützung der Entwickler können die erzeugten Werkzeuge leicht in die freie Entwicklungsumgebung Eclipse integriert werden.

Die dargestellten Ansätze verfügen über Formalismen, die eine Formulierung von Kontextbedingungen und Prüfung der Typisierung erlauben. Dabei orientieren sie sich allerdings stark an den Erfordernissen einer Programmiersprache und es ist teilweise unklar, wie sich diese auch für Modellierungssprachen verwenden lassen. MontiCore verfügt derzeit nicht über vergleichbare deklarative Formalismen, bietet jedoch Erweiterungsmöglichkeiten, um entweder handcodierte oder auf DSL-Basis implementierte Algorithmen einzubinden und so in Zukunft ähnliche Formalismen dem Sprachentwickler anzubieten. Somit ist MontiCore wie in [Kli93] beschrieben eine offene Architektur und verhindert so, dass nur eine begrenzte Anzahl an Formalismen verfügbar ist. Dies ist insbesondere wichtig, weil Spezifikationssprachen oft nur für eine bestimmte Art von Sprachen geeignet sind und somit der Spezifikationsaufwand deutlich reduziert werden kann, wenn der passende Formalismus zur Sprache gewählt werden kann. MontiCore verwendet erweiterte kontextfreie Grammatiken zur Sprachdefinition, weil zum Beispiel Notationen wie ASF+SDF sie verwendet, für ungeübte Benutzer zu kompliziert erscheinen. Die Realisierung von internen DSLs ist nicht das Hauptaugenmerk des Frameworks, so dass eine weitergehende Unterstützung nicht integriert ist.

7.9 Metamodellierungs-Frameworks

Metamodellierungs-Frameworks bezeichnen Werkzeuge, die den Fokus einer Sprachdefinition auf die abstrakte Syntax der Sprache legen und diese meisten mit einer zu UML-Klassendiagrammen ähnlichen Notation modellieren. Die folgende Auflistung stellt die am häufigsten zitierten Ansätze dar.

Das Eclipse Modeling Framework (EMF) [BSM+03] basiert auf der Metamodellierungssprache Ecore, wobei die abstrakte Syntax sowohl grafisch als auch textuell [JB06] spezifiziert werden kann. OpenArchitectureWare (OAW) [OAW] ist ein auf EMF basierendes Framework zur Erstellung von DSLs. Es können ebenfalls Metamodelle, die auf XML DOM [XML] basieren, oder Datenstrukturen im JavaBeans-Format verarbeitet werden. Das Framework besteht aus einer Sammlung an DSLs, die verschiedene Aspekte der Sprachentwicklung vereinfachen. Check ist eine Sprache zur Formulierung und Überprüfung von Kontextbedingungen. Workflow ist eine auf XML basierende Sprache zur Beschreibung von Arbeitsabläufen eines Generators. Xpand bezeichnet eine Template-Engine zur Erstellung von Dokumenten aus Modellen. Xtend ist eine Sprache zur Beschreibung von Verhalten der Metamodelle und Modelltransformationen. Xtext bezeichnet eine Sprache zur Beschreibung der konkreten Syntax von textuellen Sprachen.

Die *Generic Modeling Environment* (GME) [LMB+01] ist ein universitäres Werkzeug zur Erstellung von DSLs. GME verwendet eine komponentenbasierte Architektur basierend auf Microsoft COM, um Erweiterungen zu ermöglichen. MetaEdit+ [KT08] ist ein kommerzielles Werkzeug zur Erstellung von graphischen DSLs, die erfolgreich in industriellen Anwendungen verwendet werden.

Die Metamodellierungssprache MOF [OMG06a] ist durch die OMG standardisiert und bildet die Grundlage für die Sprachdefinition der UML. Sie kann jedoch auch zur Spezifikation weiterer DSLs verwendet werden. Dabei bietet sich das Werkzeug Moflon [AKRS06] an, das eine MOF/JMI-kompatible Implementierung bietet. Das Werkzeug Moflon basiert auf Fujaba [KNNZ99], das ebenfalls eine Implementierung von DSLs basierend auf einem propritären Meta-Metamodell erlaubt. Beide Werkzeuge verwenden Graphgrammatiken, so dass eine Sprache zur Realisierung von Modelltransformationen ins Werkzeug integriert ist.

Die Software-Factories [GSCK04] beschreiben den Einsatz von Techniken zur effizienten Softwareerstellung, wobei DSLs ein Teilaspekt dabei sind. Die Microsoft-DSLTools [CJKW07] sind eine Erweiterung der Microsoft Entwicklungswerkzeuge zur Realisierung von grafischen DSLs, die zusammen mit den Programmiersprachen der .net-Plattform verwendet werden können. Eine Interoperabilität der verschiedenen Sprachen ist aufgrund des gemeinsamen Typsystems Common Type Systems [Gou01] aller Programmiersprachen der .net-Plattform gegeben.

Alle Werkzeuge verwenden eine zu UML-Klassendiagrammen ähnliche Notation zur Spezifikation der abstrakten Syntax. Darauf aufbauend können je nach Werkzeug Transformations- oder Templatesprachen genutzt werden, um Instanzen weiter zu verarbeiten. Das MontiCore-Framework verfolgt einen ähnlichen Ansatz wie die vorgestellten Metamodellierungsframeworks, wobei es sich jedoch ausschließlich auf textuelle Sprache beschränkt und insbesondere die modulare Entwicklung von DSLs unterstützt.

Ähnlich wie OAW bietet MontiCore dem Entwickler Unterstützung für die Programmierung von komplexen Generatoren mittels des DSLTool-Frameworks an. Im Gegensatz zu OAW zeichnet sich das MontiCore-Framework durch seine Modularität in der Sprachdefinition und seine Erweiterbarkeit aus, so dass es an spezifische Bedürfnisse durch zusätzliche Konzepte angepasst werden kann.

Teil III

Definition von Werkzeugen mit dem DSLTool-Framework

Kapitel 8

Methodiken zur Entwicklung von DSLs

Während in Kapitel 2 erklärt wurde, wie sich die Erstellung und Verwendung einer DSL innerhalb des Softwareentwicklungprozesses einordnet, wird in diesem Kapitel der Entwurf einer DSL näher untersucht. Dabei wird erklärt, wie die Ergebnisse der Domänenanalyse in eine Sprachdefinition umgesetzt werden können. Die Entwicklung einer DSL erfolgt innerhalb des MontiCore-Frameworks durch Angabe einer Grammatik, die die Sprache definiert. Der Entwickler kann dabei die Modularität der Sprachentwicklung nutzen, um bereits entwickelte Sprachelemente durch Vererbung oder Einbettung einzusetzen. Die Voraussetzung hierfür ist jedoch, dass der Sprachentwickler das MontiCore-Grammatikformat und die verfügbaren Grammatiken gut kennt. Im Folgenden sollen daher praxistaugliche Methoden dargestellt werden, die nicht von einer direkten DSL-Definition ausgehen, sondern auf einem anderen möglichen Analyseergebnis basieren:

- Einem Klassendiagramm, das die Konzepte der Domäne und ihre Zusammenhänge darstellt.
- Einer Grammatikdarstellung, die aus einer Spezifikation innerhalb der Domäne stammen kann.
- Quellcode, der eine prototypische Implementierung von zukünftig zu generierendem Code darstellt.

Das Ergebnis des Importvorgangs, der durch das Werkzeug MCImporter teilautomatisiert ist, ist jeweils eine MontiCore-Grammatik, die die Sprache innerhalb des Frameworks definiert. Diese Grammatik kann dann konstruktiv weiterentwickelt werden, so dass die abstrakte oder konkrete Syntax verändert wird und die Grammatik eindeutig zur Parsererzeugung verwendet werden kann.

Dieses Kapitel ist wie folgt gegliedert: In Abschnitt 8.1 werden zunächst die Transformationen der MontiCore-Grammatik erklärt, die eine gezielte Veränderung einzelner Aspekte erlauben. In Abschnitt 8.2 wird der Import aus UML/P-Klassendiagrammen erklärt, wohingegen in Abschnitt 8.3 die Importierung aus einer BNF-Grammatik erläutert wird. Zusätzlich wird in Abschnitt 8.4 beschrieben, wie prototypische Implementierungen zu vollständigen Generatoren entwickelt werden können. In Abschnitt 8.5 wird dieses Kapitel kurz zusammengefasst.

8.1 Transformationen für MontiCore-Grammatiken

In diesem Abschnitt werden Transformationen dargestellt, die eine MontiCore-Grammatik in eine andere überführen. Dabei sind die folgenden drei Aspekte von Bedeutung:

- Dass es möglich ist, aus der Grammatik eindeutig einen Parser abzuleiten, ohne dass dabei Warn- und Fehlermeldungen auftreten. Diese Eigenschaft wird im Folgenden kurz als Parsefähigkeit bezeichnet. Dabei wird zwischen MontiCore-Grammatiken (MCG) und MontiCore-Grammatiken, die die Parsefähigkeit besitzen (MCG^{P}), unterschieden, wobei MCG^{P} eine echte Teilmenge von MCG ist.
- Die definierte konkrete Syntax der Sprache.
- Die definierte abstrakte Syntax der Sprache.

Bei Transformationen von einer Grammatik g_1 in eine Grammatik g_2 wird, wie bei Modellen üblich, eine Transformation als Spezialisierung bezeichnet, falls $[\![g_1]\!] \supset [\![g_2]\!]$ gilt. Für Refactorings muss $[\![g_1]\!] = [\![g_2]\!]$ und für eine Generalisierung $[\![g_1]\!] \subset [\![g_2]\!]$ gelten. Diese Eigenschaften können aufgrund der in Kapitel 5 beschriebenen Art der Semantikdefinition nicht direkt verwendet werden, weil die Abbildungen $[\![.]\!]_{\text{cs}}$ und $[\![.]\!]_{\text{as}}$ keine lose mengenwertige Semantik darstellen, sondern die MontiCore-Grammatiken auf genau eine X-Grammatik beziehungsweise ein UML/P-Klassendiagramm abgebildet wird. Zudem können typischerweise als Refactorings für die konkrete Syntax bezeichnete Transformationen wie zum Beispiel Fold- und Unfold-Operationen bezüglich Nichtterminalen nicht direkt als semantikerhaltende Transformationen verstanden werden, weil sich die Struktur der Grammatik auch in der X-Grammatik widerspiegelt. Da eine solche Klassifikation dann unintuitiv wäre, wird für eine Grammatik g anstatt $[\![g]\!]_{\text{cs}}$ hier $\text{L}_{[\![g]\!]_{\text{cs}}}(n)$ als die von einem Startsymbol n erzeugte Menge aller Wörter betrachtet. Dadurch ergeben sich für typische Operationen die beschriebenen Mengenbeziehungen. Analog wird hier die Menge aller Objektstrukturen anstatt des UML/P-Klassendiagramms als semantische Domäne für abstrakte Syntax verwendet. Daher sind im Folgenden diese beiden Semantiken gemeint, falls von Spezialisierungen, Refactorings oder Generalisierungen bezüglich der konkreten oder abstrakten Syntax gesprochen wird.

In diesem Abschnitt wird eine Vorgehensweise dargestellt, die zur Modifikation einer MontiCore-Grammatik verwendet werden kann, um die Parsefähigkeit herzustellen und die abstrakte und konkrete Syntax anzupassen. Die Aktivität kann auf jede MontiCore-Grammatik angewendet werden, wobei diese insbesondere nach der im Folgenden dargestellten Importierung aus den verschiedenen Formaten sinnvoll ist. Bei den einzelnen Aktionen werden die beschriebenen Eigenschaften einer MontiCore-Grammatik verändert. Sofern eine spezielle Veränderung erreicht wird, wird dies explizit erwähnt. Die dargestellten Modifikationen orientieren sich an den Ausführungen in [Läm01a], wobei die Auswirkungen der einzelnen Operationen nur eingeschränkt vergleichbar sind, da sich die verwendeten semantischen Domänen stark unterscheiden. Die Abbildung 8.1 stellt die Abfolge der einzelnen Aktion und deren typische Kombination zusammenhängend dar.

Eine MontiCore-Grammatik bildet die Grundlage für die Parsergenerierung mit dem Werkzeug Antlr. Aufgrund des verwendeten LL-Parseverfahrens müssen Linksrekursionen in den Produktionen vermieden werden. Unter einer Linksrekursion wird verstanden, dass Produktionen existieren, bei denen der Produktionskörper so abgeleitet werden kann, dass er mit demselben Nichtterminal beginnt, das sich auf der linken Seite der Produktion befindet. Dadurch läuft der Erkennungsprozess eines LL-Parsers in eine Endlosschleife und

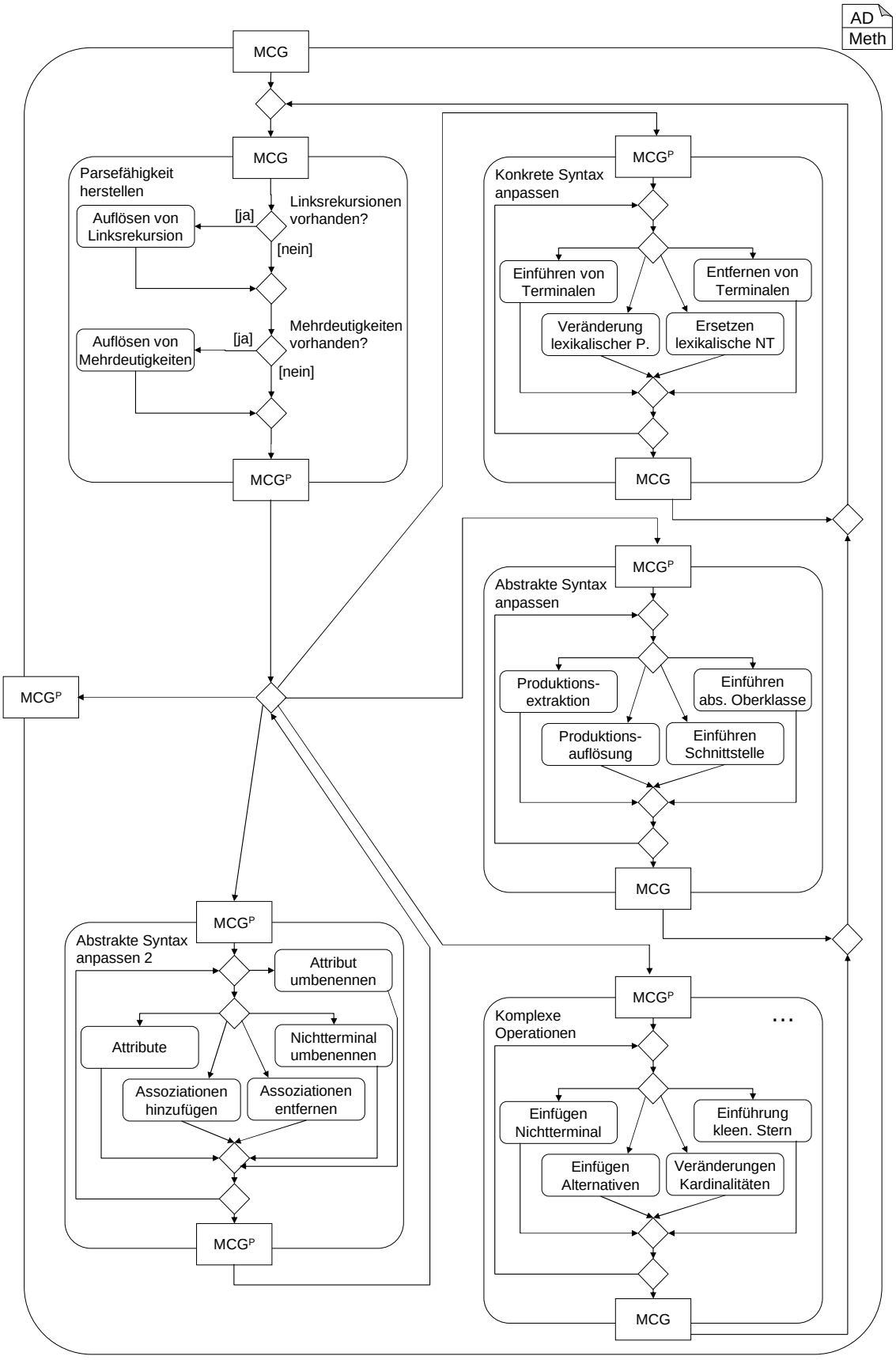


Abbildung 8.1: Transformationen einer MontiCore-Grammatik

terminiert nicht. Die Abbildung 8.2 zeigt zwei Ausschnitte aus MontiCore-Grammatiken, die beide dieselbe Sprache beschreiben, wobei die linke Grammatik jedoch eine Linksrekursion aufweist, die in der rechten entfernt wurde. Innerhalb der Aktion *Auflösen von Linksrekursion* werden alle Produktionen so verändert, dass keine Linksrekursion mehr auftritt. Dabei ist zu beachten, dass das Beispiel bezüglich der konkreten Syntax eine Refactoringoperation darstellt (wenn B als Startproduktion verwendet wird), aber eine andere abstrakte Syntax definiert.

MontiCore-Grammatik

```
1 B     = AList
2
3 AList = AList A | A
```

MontiCore-Grammatik

```
1 B = A*
2
3
```

Abbildung 8.2: Auflösen von Linksrekursion

Eine MontiCore-Grammatik kann mehrdeutig sein, was bedeutet, dass der entstehende Parser im Erkennungsprozess aus mehreren Möglichkeiten auswählen könnte und die Grammatik kein eindeutiges Auswahlkriterium festlegt. Dabei können typischerweise drei Arten von Mehrdeutigkeiten auftreten, die in der Aktion *Auflösen von Mehrdeutigkeiten* entfernt werden.

- Es existieren zwei Alternativen, die mit dem verwendeten Lookahead voneinander nicht unterschieden werden können. Hier sollte der Lookahead im Options-Block der Grammatik vergrößert werden (vergleiche Abschnitt 3.5.1).
- Es existieren zwei Alternativen, die mit einem festen Lookahead nicht unterscheidbar sind. Hier sollten Prädikate eingesetzt werden, die es erlauben, die Unterscheidung gezielt zu spezifizieren (vgl. Abschnitt 3.5.2). Alternativ kann durch eine Linksfaktorisierung die Grammatik entsprechend modifiziert werden, so dass die gemeinsamen Elemente vor die Alternative gezogen werden und so eine eindeutige Entscheidung möglich wird. Verändert sich dadurch die Struktur der Sprache, ist die Verwendung von Variablen und parametrisierten Produktionen hilfreich (vgl. Abschnitt 3.5.3).
- Eine Fortsetzung nach einem Block und die wiederholte Ausführung eines Blocks haben denselben Anfang. Hier muss sich zwischen einer wiederholten Ausführung und dem Abbruch entschieden werden, was durch entsprechende Optionen festgelegt werden kann (vgl. Abschnitt 3.5.8).

Die konkrete Syntax einer DSL lässt sich durch geeignete Modifikationen der MontiCore-Grammatik verändern. Im Allgemeinen besteht dabei die Möglichkeit, dass aus der Grammatik danach nicht mehr eindeutig ein Parser abgeleitet werden kann, so dass diese Eigenschaft gegebenenfalls korrigiert werden muss. Die wichtigsten Aktionen, die die abstrakte Syntax unverändert lassen, sind dabei die folgenden:

- Das *Einführen von Terminalen* und das *Entfernen von Terminalen* innerhalb von Produktionen.
- Die *Veränderung einer lexikalischen Produktion*, bei der der Produktionskörper verändert wird.
- Das *Ersetzen eines lexikalischen Nichtterminals*, bei dem ein Nichtterminal, das auf eine lexikalische Produktion verweist, durch ein Nichtterminal ersetzt wird, das sich auf eine andere lexikalische Produktion mit dem gleichen Rückgabewert bezieht.

Die abstrakte Syntax, die durch eine MontiCore-Grammatik definiert wird, lässt sich durch die folgenden Aktionen beeinflussen, wobei die konkrete Syntax unter gewissen Rahmenbedingungen unverändert bleibt, die Parsefähigkeit jedoch trotzdem aufgrund von internen Mechanismen des Parsergenerators verloren gehen kann.

- Die Aktion *Produktionsextraktion* bildet aus einem Ausdruck innerhalb eines Produktionskörpers eine neue Produktion, wobei das entsprechende Nichtterminal anstatt des Ausdrucks verwendet wird. Die Aktion *Produktionsauflösung* kehrt die Aktion um und ersetzt ein Nichtterminal durch den Produktionskörper. Beide Aktionen verändern die First- und Followmengenberechnung innerhalb der Parsergenerierung und können daher zusätzliche Mehrdeutigkeiten erzeugen. Insofern bleibt die Eigenschaft der Parsefähigkeit nicht zwangsläufig erhalten. Die konkrete Syntax bleibt typischerweise erhalten, wobei sich Änderungen aufgrund der parametrisierten Einbettung ergeben können, falls sich der Gültigkeitsbereich von Attributen ändert, die zur Auswahl des Parsers verwendet werden.
- Die *Einführung einer Schnittstelle oder abstrakten Oberklasse* erzeugt eine zusätzliche Produktion und somit ein neues Element in der abstrakten Syntax. Hierbei verändert sich die Produktionsmenge der Grammatik, was zum Verlust der Parsefähigkeit führen kann.

Die folgenden Aktionen haben keine Auswirkung auf die Parsefähigkeit der Grammatik:

- Die Aktion *Attribute* fügt Attribute in der Grammatik hinzu oder verändert sie entsprechend. Dabei bleibt die Parsefähigkeit erhalten und die abstrakte Syntax wird entsprechend verändert. Die konkrete Syntax ändert sich nur, wenn sich die Attribute auf die parametrisierte Einbettung auswirken. Insofern kann sich nicht nur die Struktur der Grammatik, sondern auch die Menge der erkannten Wörter verändern.
- Die Aktion *Assoziationen hinzufügen* fügt neue Assoziationen zur Grammatik hinzu, wohingegen die Aktion *Assoziationen entfernen* diese Aktion umkehrt.
- Die Aktion *Nichtterminal umbenennen* benennt Nichtterminale innerhalb der Grammatik um, was zu anderen Typnamen in der abstrakten Syntax führt, die Struktur jedoch erhält. Die Aktion *Attribut umbenennen* hat dieselben Auswirkungen auf die Attribute in der abstrakten Syntax.

Zusätzlich sind komplexere Operationen wie beispielsweise das *Einfügen von Nichtterminalen*, das *Einfügen von Alternativen*, die *Veränderung von Min- und Max-Werten* und die *Einführung von kleeneschen Sternen* möglich. Dazu passend gibt es jeweils die Umkehroperationen, die die Effekte rückgängig machen. Alle Operationen verändern sowohl die abstrakte als auch die konkrete Syntax sowie die Parsefähigkeit. Eine ausführlichere Darstellung und eine teilweise Automatisierung der Operationen findet sich in [Tor06].

8.2 Erzeugung einer Grammatik aus einem Klassendiagramm

Diese Methode ist geeignet, wenn als Ergebnis der Domänenanalyse ein Klassendiagramm erstellt wurde, das die wesentlichen Konzepte als Klasse und ihre Eigenschaften als Attribute darstellt. Assoziationen können verwendet werden, um Beziehungen zwischen den Konzepten auszudrücken. Die dargestellte Aktivität ist vollständig innerhalb des Werkzeugs MCImporter automatisiert.

8.2.1 Verwandte Arbeiten

Die Human Useable Textual Notation (HUTN) [OMG04] beschreibt, wie eine textuelle Notation für eine durch ein Metamodell gegebene abstrakte Syntax einer DSL aussehen kann. Der Ableitungsprozess kann dabei in engen Grenzen beeinflusst werden. Für den OMG-Standard mangelt es an Umsetzungen, wobei eine unvollständige Implementierung in [RPKP08] beschrieben wird. Die Vorteile eines solchen Vorgehens sind die automatische Erzeugung einer Syntax, die bei der Evolution des Metamodells ebenso automatisch neu erzeugt werden kann. Die konkrete Syntax reicht aufgrund ihrer automatischen Erzeugung jedoch nicht domänenspezifische an Varianten in Bezug auf Kompaktheit und Lesbarkeit heran, und die Vorteile der automatischen Evolution erstrecken sich nicht auf die Konfiguration des Ableitungsprozesses. In diesem Abschnitt wird hingegen ein Verfahren vorgestellt, das konstruktiv aus einer Spezifikation, die nur die abstrakte Syntax einer DSL umfasst, ein integriertes Format erzeugt. Dieses kann dann weiterentwickelt werden.

Andere modellbasierte Ansätze wie [MFF+06, OAW, JBK06] können zur expliziten Modellierung der konkreten Syntax verwendet werden, wobei eine domänenspezifische textuelle konkrete Syntax erreicht werden kann. Bei einer evolutionären Veränderung des Metamodells muss dieses Modell jedoch konsistent gehalten werden. MontiCore erreicht hier eine automatische Konsistenzerhaltung durch sein integriertes Format.

8.2.2 Vorgehensweise

Bei der Überführung eines Analyse-Klassendiagramms in eine MontiCore-Grammatik sind die folgenden Schritte notwendig. In Klammern sind dabei die Konventionen aufgeführt, die die automatische Transformation im MCImporter durchführt. Die Abbildung 8.3 zeigt übersichtsartig das Vorgehen.

In der Aktion *Anlegen einer MontiCore-Grammatik* wird eine MontiCore-Grammatik erzeugt, in der für jede Klasse und jede Schnittstelle eine entsprechende Produktion angelegt wird. (Klassenproduktionen beginnen mit einem Schlüsselwort, das der kleingeschriebene Klassenname ist.)

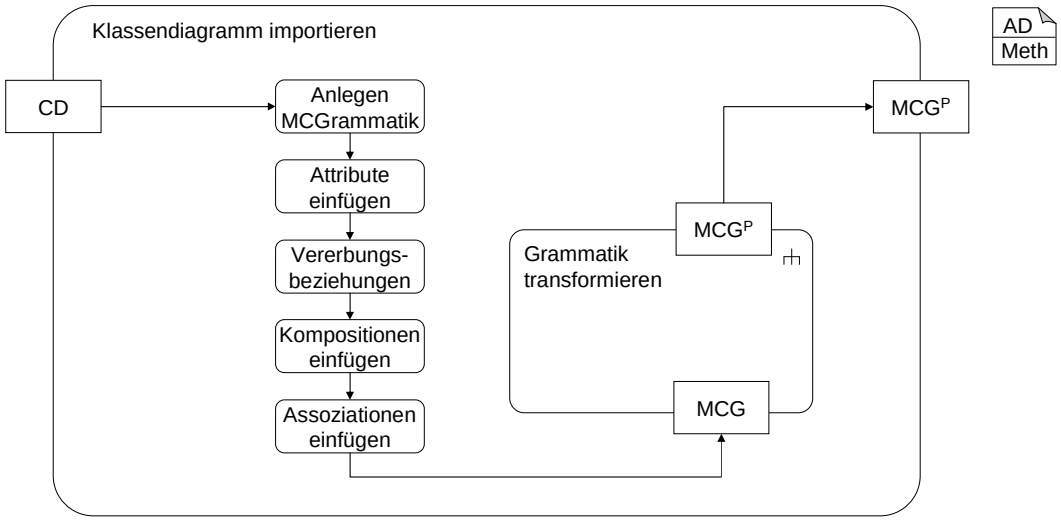


Abbildung 8.3: Importieren eines Klassendiagramms in das MontiCore-Framework

Die Aktion *Einfügen der Attribute* fügt die Attribute der Klassen als Elemente des Produktionskörpers ein. Bei abstrakten Klassen und Schnittstellen sind dazu Ast-Produktionen notwendig. Für jeden Datentyp ist dabei eine entsprechende lexikalische Produktion anzulegen. (Die Attribute werden ohne Kennzeichnung, in der Reihenfolge wie im Klassendiagramm gegeben, hintereinander aufgeführt. Für boolsche Attribute wird der Attributname als Schlüsselwort verwendet. Für andere lexikalische Produktionen werden Standardproduktionen verwendet, die für alle primitiven Datentypen zur Verfügung stehen und über eine entsprechende Obergrammatik zur Verfügung gestellt werden.)

Die Aktion *Einfügen der Vererbungsbeziehungen* nutzt die entsprechenden Elemente des Grammatikformats. (Diese Aktion ist durch das Einfügen von extends/implements-Beziehungen automatisiert.) Die Aktion *Einfügen der Kompositionsbeziehungen* fügt die Kompositionen in den Produktionskörpern für Klassen oder den AST-Regeln bei abstrakten Klassen und Schnittstellen ein. (Die Attribute werden durch ein Schlüsselwort, das dem kleingeschriebenen Kompositionsnamen entspricht, eingeleitet und nach den Attributen in die Produktionen eingefügt.) Die Aktion *Hinzufügen von Assoziationsbeziehungen* fügt die nicht-kompositionalen Assoziationen zur Grammatik hinzu. (Die Aktion ist durch Erzeugung von Assoziationen automatisiert, wobei die Etablierung der Links durch den Entwickler spezifiziert werden muss.) Abschließend wird die bereits dargestellte Aktivität *Grammatik transformieren* aus Abschnitt 8.1 verwendet, um die Struktur der Grammatik zu verändern und die Parsefähigkeit herzustellen.

8.2.3 Beispiel

In diesem Abschnitt wird das Beispiel aus [RPKP08] aufgriffen, wobei für das Metamodell aus Abbildung 8.4 eine konkrete Syntax definiert werden soll. Anstatt der spezifischen ECore-Datentypen wurden allgemeingültige primitive Datentypen verwendet und die Kardinaltät `0..*` wird durch `*` verkürzt dargestellt.

Das Beispiel beschreibt ein fiktives Datenmodell, in dem Familien durch einige Eigenschaften beschrieben werden. Dabei wird neben den üblichen Daten, Name und Adresse, auch das Durchschnittsalter und die Anzahl der Haustiere betrachtet, um verschiedene Datentypen innerhalb der Attribute zu erhalten. Boolsche Attribute werden durch die Eigenschaften verwendet, ob es sich um eine Kernfamilie, die ausschließlich aus Eltern und Kindern besteht, handelt (`nuclear`). Eine Familie besteht ihrerseits aus einer Menge an Personen, wobei nur die Kinder betrachtet und zwischen eigenen und adoptierten Kindern unterschieden wird. Personen ihrerseits haben einen Namen, ein Alter und ein Geschlecht, das durch eine Enumeration `Gender` repräsentiert wird.

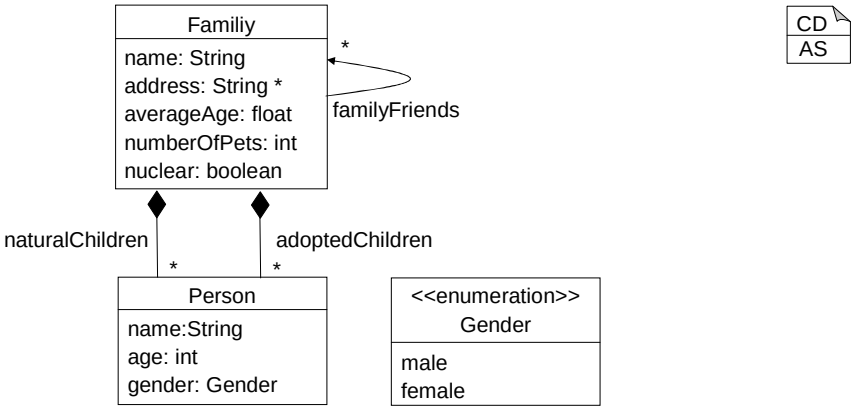


Abbildung 8.4: Adaptiertes Beispiel aus [RPKP08]

Die Abbildung 8.5 zeigt die automatisch abgeleitete Grammatik, die durch den Import eines UML/P-Klassendiagramms in das Werkzeug MCImporter entstanden ist. Die Abbildung 8.6 zeigt die manuell modifizierte Version, die eine automatische Etablierung der Links für die Assoziation `FamilyFamily` erlaubt, wobei der Assoziationsname sich dabei aus den Namenskonventionen für unbenannte Assoziationen ergibt (vgl. Abschnitt 3.4). Es wird ein zusätzliches Attribut `friends` vom Typ `String*` hinzugefügt. Zusätzlich wird über das Konzept `sreference` spezifiziert, dass zwei Familien verbunden sind, wenn ein Eintrag aus der Liste `friends` in einem Objekt vom Typ `Family` mit dem Attribut `name` eines Objekts vom Typ `Family` übereinstimmt.

MontiCore-Grammatik

```
1  grammar Family extends mc.standard.numbers.Numbers {
2
3    association
4      Family.familyFriends -> * Family;
5
6    enum Gender =
7      "male" | "female" ;
8
9    Person =
10     "person" name:STRING age:INT gender:Gender;
11
12   Family =
13     "family" name:STRING address:STRING*
14     averageAge:FLOAT numberOfPets:INT (nuclear:["nuclear"])?
15     ("naturalchildren" naturalChildren:Person*)?
16     ("adoptedchildren" adoptedChildren:Person*)?;
17 }
```

Abbildung 8.5: Automatisch abgeleitete MontiCore-Grammatik

MontiCore-Grammatik

```
1  grammar Family extends mc.standard.numbers.Numbers {
2
3    association
4      Family.familyFriends -> * Family;
5
6    concept sreference {
7      FamilyFamily : Family.friend * = Family.name;
8    }
9
10   enum Gender =
11     "male" | "female" ;
12
13   Person =
14     "person" name:STRING age:INT gender:Gender;
15
16   Family =
17     "family" name:STRING address:STRING*
18     averageAge:FLOAT numberOfPets:INT (nuclear:["nuclear"])?
19     ("naturalchildren" naturalChildren:Person*)?
20     ("adoptedchildren" adoptedChildren:Person*)? friends:STRING*;
21 }
```

Abbildung 8.6: Modifizierte Version der MontiCore-Grammatik aus Abbildung 8.5

8.3 Überführung einer BNF-Grammatik in eine MontiCore-Grammatik

Diese Methode ist geeignet, wenn während der Domänenanalyse eine DSL identifiziert wurde, die in der Domäne bereits etabliert ist. Für diese Sprache muss eine BNF- oder EBNF-Spezifikation existieren, die im Folgenden als Ausgangspunkt für die Sprachdefinition verwendet wird und schrittweise in eine MontiCore-Grammatik überführt wird.

8.3.1 Verwandte Arbeiten

In [LV01] werden Verfahren beschrieben, die Grammatik einer Sprache aus verschiedenen Artefakten wie Sprachreferenzen, Prettyprintern oder Compilern zu extrahieren. Dabei wird zunächst eine rudimentäre Grammatik extrahiert, die dann schrittweise konkretisiert wird. Dieser Abschnitt beschreibt ein ähnliches Vorgehen, beschränkt sich dabei jedoch auf die Extraktion aus einer BNF-Form (wie sie in Sprachreferenzen oft verwendet wird) und erklärt dabei die spezifischen Aspekte, die innerhalb des MontiCore-Framworks beachtet werden müssen.

Verschiedene Ansätze wie [WH95, LZ09] haben Transformationskalküle und konkrete Transformationen für Grammatiken entwickelt, die Grammatiken gezielt verändern, um sie für spezielle Anwendungszwecke besser verwenden zu können.

8.3.2 Vorgehensweise

Die Spezifikation der DSL wird schrittweise in eine MontiCore-Grammatik überführt, wobei zunächst Modifikationen durchgeführt werden, die eine generelle Übernahme in das MontiCore-Format erlauben. Daraufhin kann die Grammatik so modifiziert werden, dass sie sich eignet, um die Modelle korrekt zu parsen. Die Abbildung 8.7 stellt das Vorgehen übersichtsartig dar. Die einzelnen Aktionen bezeichnen die grundlegenden Arbeitsschritte, eine Grammatiknotation in ein für MontiCore verarbeitbares Format zu übersetzen, das dann durch die bereits vorgestellte Aktivität *Grammatik transformieren* verändert wird.

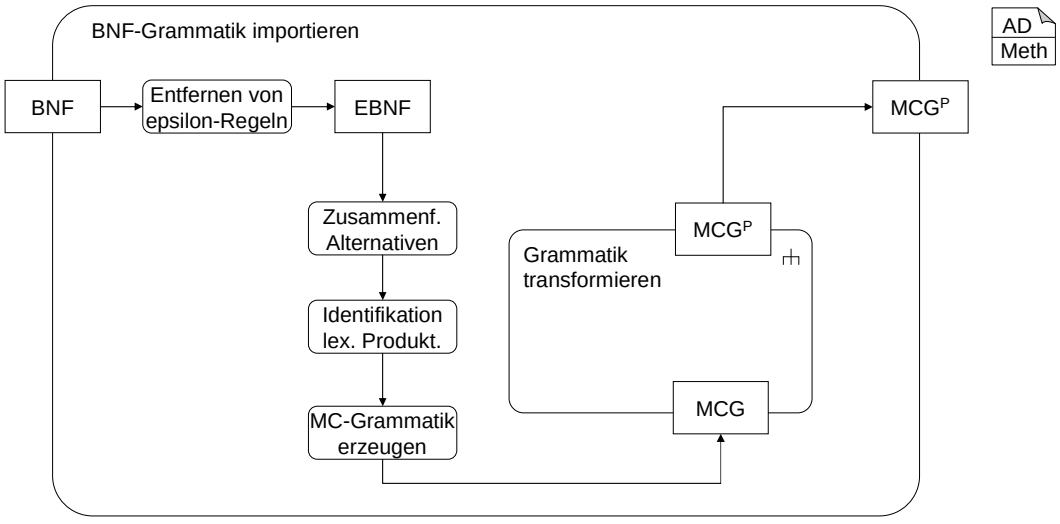


Abbildung 8.7: Importieren einer BNF-Grammatik in das MontiCore-Framework

Innerhalb der Aktion *Entfernen von ϵ-Regeln* werden Produktionen entfernt, die ausschließlich aus dem leeren Wort bestehen. Dieses geschieht durch Ersetzung aller Vorkommen des Nichtterminals durch die optionale Form. Dieses Vorgehen sollte transitiv fortgesetzt werden, falls so eine Produktion entsteht, die aus einem einzigen optionalen Nichtterminal besteht. Die einzige (theoretische) Beschränkung ist dabei, dass die Sprachen das leere Wort nicht enthalten können, was jedoch nicht von praktischer Bedeutung ist, weil ein leeres Modell mit einer bestimmten Bedeutung in DSLs selten auftritt und nötigenfalls durch ein spezielles Schlüsselwort ersetzt werden könnte.

Die Aktion *Zusammenfassung von Alternativen* dient dazu, dass jedes Nichtterminal nur noch durch eine Produktion definiert wird. Die verschiedenen Körper der Produktionen mit demselben Nichtterminal auf der linken Seite werden zu einer Produktion mit Alternativen zusammengefasst. Die Aktion *Identifikation von lexikalischen Produktionen* dient dazu, die Grammatik so zu organisieren, dass die lexikalische Analyse von der syntaktischen Analyse getrennt wird. Dabei müssen häufig natürlichsprachliche Definitionen für valide Bezeichner der DSL in reguläre Ausdrücke umgesetzt werden. Die Aktion *MontiCore-Grammatik erzeugen* dient dazu, aus der EBNF-Spezifikation formal eine MontiCore-Grammatik zu erzeugen. Für Produktionen, die ausschließlich aus alternativen Nichtterminalen bestehen, sollten Schnittstellenproduktionen verwendet werden. Die Aktivität *Grammatik transformieren* dient dazu, die Parsefähigkeit herzustellen und geeignete Modifikationen an der Grammatik durchzuführen (vgl. Abschnitt 8.1).

8.3.3 Beispiel

Das Beispiel entwickelt eine BNF-Beschreibung endlicher Automaten schrittweise in eine MontiCore-Grammatik. Dazu werden die folgenden Schritte ausgeführt, um die BNF-Grammatik aus Abbildung 8.8 umzuformen.

In der Aktion *Entfernen von ϵ-Regeln* werden die beiden Produktionen `StateList` und `TransitionList` entfernt und dafür ein kleenescher Stern und das Nichtterminal `State` beziehungsweise `Transition` eingesetzt. In der Aktion *Einführen von Alternativen* werden die einzelnen Produktionen für die Nichtterminalen `State` und `Expression` zu jeweils einer Produktion mit Alternativen zusammengefasst. In der Aktion *Identifikation von lexikalischen Produktionen* werden die Produktionen `Identifier` und `Literal` expliziert, in dem ein regulärer Ausdruck angegeben wird. In der Aktion *MontiCore-Grammatik erzeugen* wird eine MontiCore-Grammatik `Automaton` erzeugt und die Schreibweise der lexikalischen Produktionen angepasst. Das Ergebnis des Umformungsprozesses findet sich in Abbildung 8.9.

Daraufhin wird die Grammatik angepasst, um die Parsefähigkeit herzustellen. Dazu muss in der Produktion `State` eine Linksfaktorierung durchgeführt werden. Eine aufwändigere Änderung ist die Anpassung der Ausdruckstrukturen, um die unterschiedlich stark bindenden Operatoren darstellen zu können, die hier aber nicht ausgeführt wird.

BNF

```
Automaton      ::= StateList TransitionList

StateList      ::= State StateList
StateList      ::=

TransitionList ::= Transition TransitionList
TransitionList ::=

State          ::= "state" Identifier ";"
State          ::= "state" Identifier "{" StateList TransitionList "}"

Transition     ::= Identifier "->" Identifier ":" Expression

Expression     ::= Identifier "=" Expression
Expression     ::= Expression "+" Expression
Expression     ::= Expression "*" Expression
Expression     ::= Identifier
Expression     ::= Literal

Identifier     ::= << Menge aller Bezeichner >>

Literal        ::= << Positive Zahlen >>
```

Abbildung 8.8: Initiale Grammatik

MontiCore-Grammatik

```
grammar Automaton {

  Automaton      = State* Transition*;

  State          = "state" IDENTIFIER ";"
                 | "state" IDENTIFIER "{" State* Transition* "}";

  Transition     = IDENTIFIER "->" IDENTIFIER ":" Expression ;

  Expression     = IDENTIFIER "=" Expression
                 | Expression "+" Expression
                 | Expression "*" Expression
                 | IDENTIFIER
                 | LITERAL

  IDENTIFIER     = ("a".."Z")+;

  LITERAL        = ("0".."9")+;

}
```

Abbildung 8.9: MontiCore-Version nach der Aktion *MontiCore-Grammatik erzeugen*

8.4 Erzeugen einer Sprachdefinition aus existierendem Code

Diese Methodik ist geeignet, wenn Java-Quellcode existiert, der einen Prototyp für automatisch zu erzeugenden Produktcode darstellt. Die daraus zu erstellenden Varianten müssen dem Prototyp ähneln, wobei die Unterschiede der Varianten durch DSL-Modelle beschrieben werden. Die Struktur dieser Modelle wird durch eine DSL festgelegt, die die Variationspunkte des entstehenden Generators definiert, so dass aus einem Modell eine konkrete Variante des Prototypen abgeleitet wird. Die DSL kann dabei aus dem exemplarischen Produktcode extrahiert werden, wobei dafür Direktiven der MontiCore-Template-Engine (vgl. Abschnitt 9.3.9) hinzugefügt werden müssen. Das Ergänzen der Direktiven ist ein manueller Schritt, weil der Entwickler die Variabilität selbst festlegen muss und dieser Schritt somit nicht automatisierbar ist. Die Ableitung der DSL hingegen ist innerhalb des Werkzeugs MCImporter automatisiert.

8.4.1 Verwandte Arbeiten

In [CJKW07] wird beschrieben, wie methodisch aus exemplarischem Quellcode ein Generator auf Template-Basis entwickelt werden kann. Im Gegensatz zum hier dargestellten Ansatz ist das Template kein compilierbarer Quellcode mehr, so dass nachträgliche Veränderungen des konstanten Anteils nur ohne Unterstützung der üblichen Entwicklerhilfsmittel wie Autovervollständigung möglich sind und eine automatische Ableitung des Datenmodells wird nicht angeboten.

Für verschiedene Template-Engines [OAW, TOP] existiert Werkzeugunterstützung, die es erlaubt, das Datenmodell bei der Erstellung des Templates zu berücksichtigen und so eine statische Prüfung der Direktiven auf Korrektheit zur Entwicklerunterstützung zu verwenden. Die im Folgenden dargestellte Template-Engine erlaubt jedoch darüber hinaus eine statische Prüfung der Daten des Templates und die Entwicklung von Testfällen.

8.4.2 Vorgehensweise

Die Abbildung 8.10 zeigt das im Folgenden beschriebene Vorgehen als Aktivitätsdiagramm. In der Aktion *Templates hinzufügen* wird die prototypische Implementierung manuell um Template-Informationen der MontiCore-Template-Engine (vgl. Abschnitt 9.3.9) angereichert. Dabei müssen die Punkte innerhalb des exemplarischen Quellcodes markiert werden, die sich bei der Generierung ändern werden, indem Zeichen des exemplarischen Quellcodes durch die Elemente der Modelle ersetzt werden. Dieser Schritt wird manuell vom Entwickler ausgeführt und ist nicht automatisierbar, weil hierbei die Variationspunkte des Generators festgelegt werden.

Die Besonderheit der hier verwendeten MontiCore-Template-Engine ist, dass das Template selbst bereits compilierbarer Java-Code ist. Die Template-Direktiven (Tags) haben die Form von Java-Kommentaren und lassen sich so an beliebigen Stellen im Quellcode einsetzen. Beim Ersetzungprozess werden die Tags durch die Daten der Modelle ersetzt und zusätzlich das nächste Wort innerhalb der Java-Datei entfernt. Eine weitere detaillierte Beschreibung der Template-Engine befindet sich in Abschnitt 9.3.9. Der wesentliche Vorteil bei der Verwendung ist, dass bei der Implementierung des Templates weiterhin sämtliche Werkzeugunterstützung für die Programmiersprache Java genutzt werden kann, wie sie in integrierten Entwicklungsumgebungen wie Eclipse zur Verfügung steht. Dieses stellt einen deutlichen Komfortgewinn für die Entwickler dar, weil Funktionen wie Autovervollständigung und Refactorings benutzt werden können.

Innerhalb der Aktion *Ableitung des Klassendiagramms* wird das Template vom MCImporter mittels eines speziellen Parsers eingelesen. Bei der Analyse wird ein Datenmodell aufgebaut, das initial eine einzelne Klasse enthält. Dabei werden jeweils die Elemente ausgewertet, die eine direkte Ersetzung hervorrufen, und hierfür Attribute in der Hauptklasse des Datenmodells erzeugt. Als Typ dieser Attribute wird stets `String` verwendet. Die entsprechenden Attribute lassen sich dadurch erkennen, dass sie in Prozentzeichen eingeschlossen sind, das heißt zum Beispiel, die Direktive `/*C Test %Name% */` erzeugt ein Attribut Name in der Hauptklasse. Die konstante Zeichenkette „Test“ wird nicht in das Datenmodell aufgenommen.

Innerhalb eines Templates kann das aktuell betrachtete Objekt gewechselt werden, indem `/*for` oder `/*foreach`-Elemente verwendet werden. Dabei geht der Fokus auf ein Kind oder nacheinander auf die Elemente einer Menge von Kindern über. Bei der Extraktion der Datenstruktur aus dem Template wird diese Information genutzt, um eine abhängige Klasse zu erstellen, die mit einer Kompositionsbeziehung verbunden wird. Zum Beispiel wird durch die Direktive `/*forall %A% */` eine Klasse `A` erzeugt, die mit der aktuellen Klasse über eine Kompositionsbeziehung mit Namen `A` verbunden ist. Ist bereits eine solche Klasse im entstehenden Datenmodell enthalten, wird diese verwendet. Die innerhalb der `/*for` oder `/*foreach`-Blöcke enthaltenen Elemente erzeugen wiederum die Attribute dieser Klasse. Der vorliegende Algorithmus wird rekursiv angewendet, so dass weitere Klassen entstehen können.

Die resultierende Datenstruktur wird als UML/P-Klassendiagramm ausgegeben und kann daher vom Entwickler bezüglich der Vollständigkeit beurteilt werden oder gegebenenfalls verändert werden. Das Klassendiagramm kann dann über die bereits erklärte Aktivität *Klassendiagramm importieren* in das MontiCore-Framework übernommen werden. Generell gilt, dass die kanonisch aus dem Template enstandene Grammatik nicht besonders gut geeignet ist, um als Eingabesprache eines Generators verwendet zu werden. Sie kann jedoch manuell verbessert werden, so dass die abstrakte Syntax unverändert, die konkrete Syntax jedoch für Nutzer zugänglicher ist. Ändert sich die abstrakte Syntax, ist zu beachten, dass die durch die Grammatik definierte abstrakte Syntax jedoch weiterhin zum Template passen muss. Alternativ kann eine anders entstandene Sprache als Eingabesprache verwendet

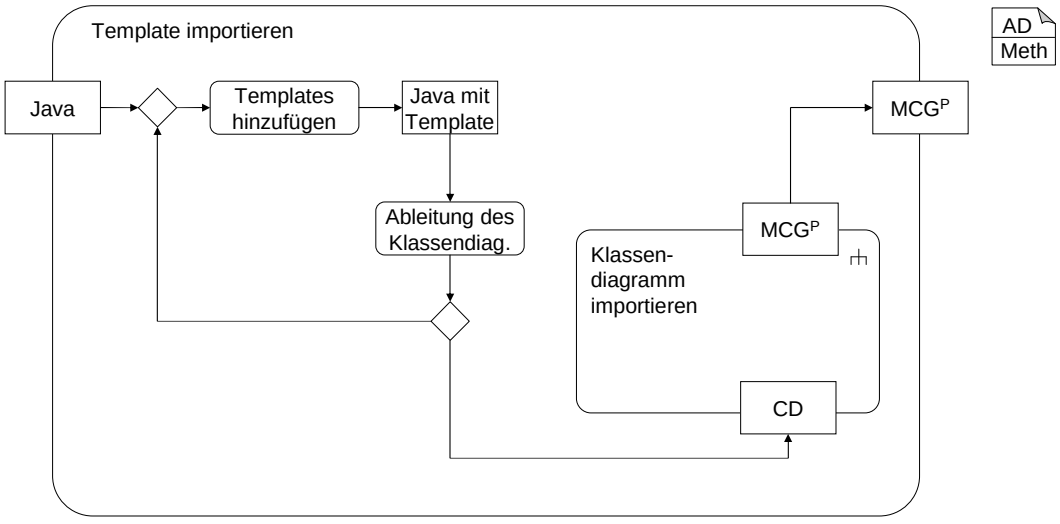


Abbildung 8.10: Importieren eines Templates

werden. Dann dient der auf dem Template basierende Generator ausschließlich zur Codeerzeugung. Die Überführung der Eingabe in eine zum Generator konforme Eingabe wird dann durch eine Modelltransformation realisiert. Die Abbildung 8.11 zeigt schematisch dieses Vorgehen.

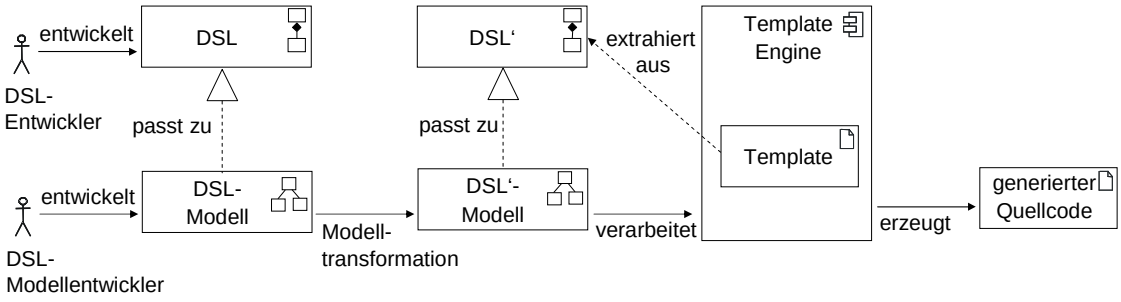


Abbildung 8.11: Einsatz von Templates und Modelltransformationen

Die Ableitung verwendet eine sehr einfache Heuristik, da die Kompositionsnamen gleichzeitig als Klassennamen verwendet werden. Dieses kann dazu führen, dass Attribute in einer Klasse zusammengeführt werden, die bei manueller Erstellung durch einen Entwickler nicht zusammengelegt werden würden. Dieses muss dann nach der automatischen Ableitung nachträglich manuell korrigiert werden. Dazu bietet der MCImporter die Möglichkeit, eine MontiCore-Grammatik und ein Template einzulesen. Die Konformität lässt sich automatisiert prüfen und es lassen sich automatisiert konstruktive Fehlermeldungen ausgeben.

8.4.3 Beispiel

Die Abbildung 8.12 zeigt einen kurzen Auszug aus einer manuell programmierten RootFactory. Diese dient innerhalb des DSLTool-Frameworks dazu, die Root-Objekte zu erzeugen und entsprechend mit dem Parser und dem passenden PrettyPrinter zu instanzieren (vgl. auch Abschnitt 9.2). Bei der Implementierung des Beispiels wurde darauf geachtet, dass alle Möglichkeiten, die strukturelle Veränderungen am Quellcode hervorrufen, wenigstens einmal verwendet wurden.

Ausgehend von diesem exemplarischen Quellcode wurden die Variationspunkte identifiziert, die diese konkrete RootFactory von anderen unterscheidet. Der exemplarische Quellcode wurde dann um die nötigen Template-Direktiven erweitert, um daraus ein Template für die Wrapperklassen zu erzeugen. Das daraus resultierende Template befindet sich in Abbildung 8.14. Aus diesem Template kann ein Datenmodell wie in Abbildung 8.13 gezeigt extrahiert werden, das die Schnittstelle des Generators bildet. Hieraus wurde eine DSL extrahiert, die jedoch nicht direkt verwendet wird, sondern wie in Abbildung 8.11 gezeigt, als Schnittstelle des Generators verwendet wird.

Java-Quellcode

```
1  package mc.hw.template;
2
3  public class HWRootFactory
4    extends mc.DSLRootFactory<HWRoot > {
5
6    // ...
7
8    protected mc.grammar.MonticoreParser setupParser
9      (java.io.Reader reader, String filename) {
10
11     mc.grammar.MonticoreParser overallparser =
12       new mc.grammar.MonticoreParser(filename, reader, errorHandler);
13
14     overallparser.addMCConcreteParser(
15       new mc.hw.template._parser.HWDSLHWUnitMCConcreteParser ( "hw" ));
16
17     overallparser.addExtension("outer", "inner" , "ext");
18     overallparser.addExtension("outer", "inner", "ext", "parameter");
19
20     overallparser.setStartParser("hw");
21
22     overallparser.getAnnotationStack().pushAnnotation("test","startvalue");
23
24     return overallparser;
25   }
26 }
```

Abbildung 8.12: Auszug aus einer prototypischen RootFactory

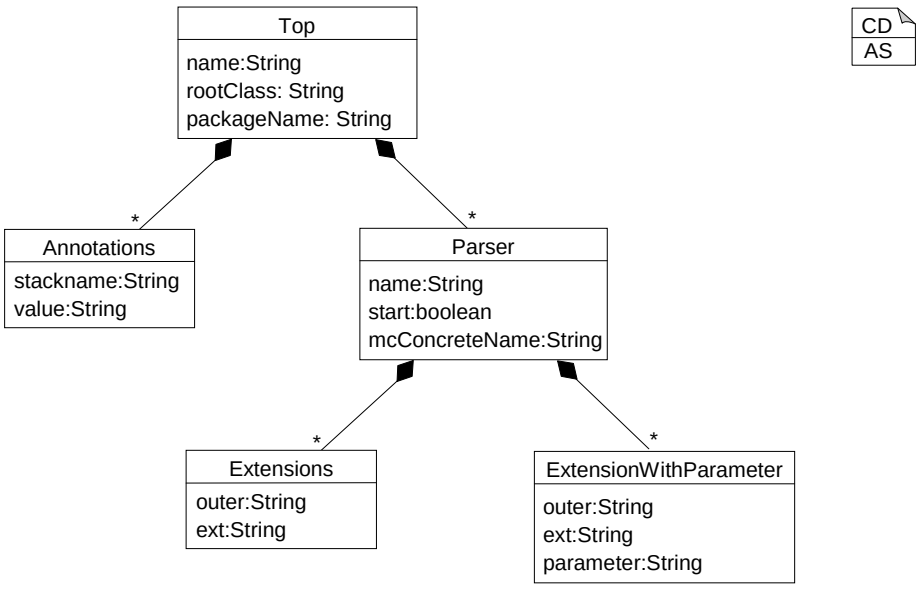


Abbildung 8.13: Das aus dem Template aus Abbildung 8.14 extrahierte Datenmodell

Java-Quellcode

```
1  package /*C %PackageName% */ mc.hw.template ;
2
3  public class /*C %Name% */ HWRootFactory
4    extends mc.DSLRootFactory< /*C %RootClass% */ HWRoot > {
5
6    // ...
7
8    protected mc.grammar.MonticoreParser setupParser
9      (java.io.Reader reader, String filename) {
10
11     mc.grammar.MonticoreParser overallparser =
12       new mc.grammar.MonticoreParser(filename, reader, errorHandler);
13
14     /*foreach %Parsers% */
15       overallparser.addMCConcreteParser(
16         new
17         /*C %MCConcreteName% */ mc.hw.template._parser.HWDSLHWUnitMCConcreteParser
18         ( /*C " %Name% "  */ "hw" )
19       );
20     /*end*/
21
22     /*foreach %Parsers% */
23       /*foreach %Extensions% */
24         overallparser.addExtension(
25           /*C " %Outer% "  */ "outer" ,
26           /*C " %1%Name% "  */ "inner" , /*C " %Ext% "  */ "ext"
27         );
28       /*end*/
29     /*end*/
30
31     /*foreach %Parsers% */
32       /*foreach %ExtensionsWithParameter% */
33         overallparser.addExtension(
34           /*C " %Outer% "  */ "outer" , /*C " %1%Name% "  */ "inner" ,
35           /*C " %Ext% "  */ "ext" , /*C " %Parameter% "  */ "parameter"
36         );
37       /*end*/
38     /*end*/
39
40     /*foreach %Parsers% */
41       /*if %Start% */
42         overallparser.setStartParser( /*C " %Name% "  */ "hw" );
43       /*end*/
44     /*end*/
45
46     /*foreach %Annotations% */
47       overallparser.getAnnotationStack().pushAnnotation(
48         /*C  " %Stackname% " */ "test" , /*C " %value% " */ "startvalue"
49       );
50     /*end*/
51
52     return overallparser;
53   }
54 }
```

Abbildung 8.14: Auszug aus dem aus Abbildung 8.12 entwickelten Template

8.5 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie das MontiCore-Framework verwendet werden kann, um verschiedene mögliche Ergebnisse einer Domänenanalyse zu nutzen und diese konstruktiv in eine MontiCore-Grammatik zu überführen. Dabei wurden drei Methoden näher beschrieben: Erstens die Importierung von Klassendiagrammen als ein Beispiel für strukturelle Beschreibungen, die abstrakte Syntax der DSL festzulegen, ohne die konkrete Syntax einer Sprache zu beschreiben. Zweitens die Importierung von BNF-Grammatiken als ein Beispiel für eine Beschreibungsform, die nur die konkrete Syntax einer Sprache festlegt. Und schließlich drittens die konstruktive Entwicklung eines Generators aus prototypischem Quellcode als eine Form der Entwicklung einer DSL, die nicht eine Spezifikation der DSL als Ausgangspunkt verwendet, sondern Beispiele als Startpunkt nutzt.

Zu den gewählten Beispielen gibt es Alternativen: So können durch ähnliche Mechanismen, wie sie für die UML/P-Klassendiagramme beschrieben wurden, EMF- oder MOF-Metamodelle importiert werden. Bei der Importierung von anderen Grammatikformaten, wie sie zum Beispiel durch andere Parsergeneratoren verwendet werden, greifen ähnliche Prinzipien wie bei der Importierung der BNF-Grammatiken. Anders als exemplarischen Quellcode als Ausgangspunkt zu verwenden, ist auch die Verwendung von exemplarischen Dokumenten der DSL denkbar.

In [Tor06] wurde näher untersucht, wie sich exemplarische Dokumente gezielt in eine Grammatik überführen lassen. Zusätzlich können weitere Modelle eingelesen werden, um die Unterschiede zwischen Modell und Grammatik zur konstruktiven Weiterentwicklung zu nutzen. Konkret wird ein modifizierter Parser genutzt, um die Position der Erkennungsfehler zur Etablierung zusätzlicher Alternativen zu nutzen. So lässt sich schrittweise aus den exemplarischen Modellen eine DSL entwickeln.

Alle Methoden werden dabei durch Transformationen für das MontiCore-Grammatikformat unterstützt, die gezielt bestimmte Teile der Grammatik verändern. Dabei modifizieren einige Transformationen nur die konkrete Syntax und lassen die abstrakte Syntax unverändert oder anders herum. Wichtig für eine fehlerfreie Verwendung ist dabei, die Parsefähigkeit herzustellen, die die fehlerfreie Erkennung der Modelle sicherstellt und eine eindeutige Überführung in einen AST garantiert.

Kapitel 9

Verarbeitung von Sprachen mit dem DSLTool-Framework

Dieses Kapitel beschreibt das DSLTool-Framework, das die automatische Verarbeitung von Modellen innerhalb einer modellgetriebenen Entwicklung durch die Erstellung generativer Werkzeuge unterstützt. Daher wurden bei der Erstellung von Werkzeugen zur Verarbeitung von verschiedenen DSLs stets wiederkehrende technische Fragestellungen identifiziert, für die qualitativ hochwertige Lösungen bereitgestellt werden sollten. Das DSLTool-Framework fasst diese Lösungen zu einer Referenzarchitektur zusammen, so dass es als Grundlage für eine Vielzahl an generativen Werkzeugen dienen kann. Das Framework wurde insbesondere bei der Entwicklung des MontiCore-Generators verwendet und so kontinuierlich verbessert.

Die Architektur eines solchen Frameworks unterscheidet sich dabei von der Architektur eines Compilers, weil nicht nur eine homogene Eingabesprache mit einer einzelnen Codegenerierung verwendet wird, sondern aufgrund der verschiedenen Zielsetzungen heterogene Eingabesprachen mit jeweils mehreren Codegenerierungen zu einem Werkzeug kombiniert werden sollen. Zusätzlich wurde eine einheitliche Zwischencodeerstellung wie zum Beispiel die Single-Static-Assignment-Form für GPLs bei Modellen bisher nicht identifiziert.

Die wichtigsten Anforderungen, so genannte Architekturtreiber [BCK03], die an ein solches Framework gestellt wurden, sind die folgenden:

- Modulare, voneinander entkoppelte Entwicklung von Algorithmen für eine Sprache.
- Integration verschiedener Sprachen und darauf basierter Algorithmen innerhalb eines Werkzeugs.
- Flexible Konfiguration eines Werkzeugs, so dass es mit unterschiedlichen Parametersätzen verwendet werden kann.
- Einfach nutzbare APIs für häufig wiederkehrende Aufgaben innerhalb der generativen Entwicklung.
- Ausführbarkeit des Frameworks und darauf basierender Werkzeuge auf verschiedenen Plattformen wie die Eclipse-Entwicklungsumgebung und die Kommandozeile.

Aus dieser Liste an Architekturtreibern wurden detaillierte Funktionalitäten abgeleitet, für die das Framework Lösungen anbieten soll. Die folgende Auflistung beschreibt sie übersichtsartig, wobei sie in Abschnitt 9.3 detailliert beschrieben werden:

Ablaufsteuerung. Die Ablaufsteuerung der verschiedenen Arbeitsschritte soll konfigurierbar sein. Somit können Werkzeuge erstellt werden, die je nach Parametrisierung ein spezifisches Verhalten zeigen.

Dateierzeugung. Bei der Dateierzeugung innerhalb eines generativen Werkzeugs ist zu beachten, dass Dateien nicht immer sondern nur bedarfsorientiert bei Änderungen und für die Testfälle gar nicht geschrieben werden können. Ersteres erhöht die Geschwindigkeit bei der Kopplung mit weiteren Werkzeugen wie Compilern und zweites verhindert das Auftreten von Seiteneffekten bei der Testfallausführung durch Dateien auf der Festplatte, die ein Testfall erzeugt und ein weiterer verarbeitet.

Fehlermeldungen. Generative Werkzeuge müssen verständliche Fehlermeldungen bei fehlerhaften Eingaben an den Nutzer weiterreichen, so dass dieser seine Eingaben korrigieren kann. Dabei sollen diese Fehlermeldungen trotz unterschiedlicher Ausführungsplattformen nur einmal entwickelt werden müssen.

Funktionale API. Die Manipulation von Datenstrukturen lässt sich elegant mittels des funktionalen Programmierparadigmas formulieren. Innerhalb der verwendeten Programmiersprache Java wird daher eine API benötigt, die dieses Vorgehen nachahmt, und den Entwickler auch Funktionen höherer Ordnung zur Verfügung stellt.

Inkrementelle Codegenerierung. Die Erstellung von Dateien, die auf mehreren Eingabedateien basieren, verhindert oft eine Form der Übersetzung, die nur die veränderten Eingabedateien betrachtet. Hier ist eine effektive Zwischenspeicherung der Teilinformationen notwendig, um eine inkrementelle Erzeugung zu erreichen und so die Verarbeitung zu beschleunigen.

Logging. Logging stellt eine Möglichkeit zur nachträglichen Analyse der Programmausführung dar, die die Ausführungszeit nicht wesentlich erhöht. Diese sollte innerhalb des Frameworks zentral zu Verfügung gestellt werden, damit ein integriertes Logging verschiedener Systemteile möglich ist.

Modellmanagement. Die Verarbeitung verschiedener Modelle, die sich untereinander referenzieren, macht es notwendig ein Namenssystem für Modelle zentral zu etablieren, so dass eine prinzipielle Interoperabilität zwischen verschiedenen in MontiCore definierten Modellarten besteht. Des Weiteren sollen diese Modelle aufgrund ihres Namens bei Bedarf aus verschiedenen Quellen transparent nachladbar sein.

Plattformunabhängigkeit. Die entstehenden Generatoren sollen nur einmal implementiert, aber auf verschiedenen Plattformen ausführbar sein: Als Kommandozeilenwerkzeug, als Online-Version innerhalb des SSElabs[1] und als Eclipse-Version.

Template-Engine. Die Codeerzeugung innerhalb eines modellgetriebenen Ansatzes lässt sich durch den Einsatz von Templates explizit machen und somit leichter innerhalb einer evolutionären Entwicklung einsetzen. Die Verwendung von Template-Engines sollte daher innerhalb eines generativen Werkzeugs möglich sein.

Traversierung der Datenstrukturen. Verschiedene Formen der Verarbeitung innerhalb eines generativen Werkzeugs bedürfen der konfigurierbaren Traversierung der Eingangsdaten.

[1] www.sselab.de

Der Abschnitt 9.1 vergleicht die Anforderungen und Funktionalitäten des DSLTool-Frameworks mit verwandten Ansätzen. In Abschnitt 9.2 wird die Architektur des Frameworks übersichtsartig beschrieben, wohingegen im Abschnitt 9.3 die Funktionen des Frameworks detailliert beschrieben werden. Der Abschnitt 9.4 fasst das Kapitel kurz zusammen.

9.1 Verwandte Arbeiten

Die Implementierung von DSLs und darauf aufbauenden Werkzeugen ist die Zielsetzung verschiedener Frameworks. Dieser Abschnitt stellt die wichtigsten Ansätze übersichtsartig dar, wobei bei der Beschreibung der einzelnen für das DSLTool-Framework identifizierten Querschnittsfunktionalitäten nochmals auf die spezifischen Gemeinsamkeiten und Unterschiede hingewiesen wird. Dabei wird die strukturierte Verarbeitung von Modellen oft durch eine spezielle Sprachdefinition wie [BSM+03, OMG06a] erst möglich, durch Modelltransformationssprachen wie [JK05, Tra06] explizit modellierbar und durch Template-Engines wie [Vel] vereinfacht. Dieser Abschnitt konzentriert sich jedoch auf Ansätze, die die einzelnen Werkzeuge integrieren und so eine effiziente Entwicklung von Generatoren ermöglichen.

Sprachbaukästen wie [GME, Meta, CJKW07, MPS, TOP] integrieren Komponenten für die Editoren von Modellen, die Prüfung von Kontextbedingungen und die Codegenerierung in eine konsistente Oberfläche, die gleichzeitig zur Entwicklung und Verwendung der Komponenten genutzt wird. Dadurch wird meistens eine Integration in automatische Buildsysteme erschwert. Der Entwickler wird durch spezielle Template-Engines bei der Ausgestaltung einer Codeerzeugung unterstützt, wobei weitergehende Funktionen zum Beispiel in Bezug auf inkrementelle Codeerzeugung selbst implementiert werden müssen.

IPSEN [Nag96, KS97] bezeichnet einen Ansatz, eng gekoppelte Softwareentwicklungsumgebungen für verschiedene Sprachen zu erzeugen. Dabei eignet sich der Ansatz, um textuelle als auch grafische DSLs in einer konsistenten Oberfläche verarbeiten zu können. Die Architektur unterscheidet sich vom DSLTool-Framework vor allem dadurch, dass die Entwicklungsumgebung sich an Projekten orientiert und das DSLTool-Framework eher wie ein Compiler eine Menge an Eingaben in einer Art Stapelverarbeitung in compilierbaren Quellcode umsetzt.

Das auf EMF basierende Framework OpenArchitectureWare (OAW) [OAW] unterstützt den Entwickler bei der Ausgestaltung von Generatoren. Die verwendete Strukturierung des Verarbeitungsprozesses in Workflows ähnelt sich bei OAW und MontiCore. Die OAW-Variante wird durch XML-Dokumente konfiguriert und ist nicht ohne eine Eclipse-Version lauffähig. Dieses erschwert den Einsatz von auf OAW basierenden Generatoren in Werkzeugketten. Die verwendete XML-Konfiguration stellt eine nicht erweiterbare DSL dar, wohingegen das DSLTool-Framework hier auf die Verwendung von Java setzt. Dieses erlaubt eine Erweiterung und flexible Modifikation der Ablaufsteuerung. Das Framework bietet mit XText eine Unterstützung für die textuelle DSL und mit XPand eine Template-Engine, die mit den jeweiligen Komponenten in MontiCore vergleichbar ist. Weitergehende Unterstützung zum Beispiel in Bezug auf inkrementelle Codeerzeugung oder Dateierzeugung existiert nicht.

In [Gao07] wird ein Modellierungsframework vorgestellt, das die Erstellung von DSLs erlaubt, die als Hostsprache die synchrone datenflussorientierte Sprache LUSTRE [HCRP91] verwenden. Dabei können insbesondere eingebettete DSLs mit der Forwarding-Technik erstellt werden, wobei die Verwendung als Programmiersprache im Vordergrund steht und die spezifischen Anforderungen von Modellierungssprachen nicht betrachtet werden.

Für die Programmiersprache Java existieren verschiedene erweiterbare Compiler wie Polyglot [NCM03] und JastAddJ [EH07]. Diese fokussieren sich vor allem auf die Erwei-

terungen der Programmiersprache, die sich nur bedingt auf den allgemeinen generativen Einsatz von DSLs übertragen lassen.

Die UML-F [FPR00] bezeichnet ein UML-Profil zur Modellierung und Implementierung von Frameworks. Bei der Ausgestaltung des DSLTool-Frameworks wurde das Prinzip der Adaption durch Spezialisierung übernommen. Bei den im Folgenden gezeigten Diagrammen wurden, wo passend, die UML-F Stereotypen ≪template≫ und ≪hook≫ zur Kennzeichnung von Template- und Hook-Methoden verwendet.

9.2 Architektur

Aus der identifizierten Querschnittsfunktionalität und den verwandten Arbeiten wurde das DSLTool-Framework entwickelt, dessen Architektur im Folgenden dargestellt wird. Ein DSLTool stellt dabei die Referenzarchitektur für ein generatives Werkzeug dar, dessen Erstellung dadurch vereinfacht wird. Zusätzlich kann die Qualität gesteigert werden, da bewährte Funktionalitäten wieder verwendet werden können. Ein Entwickler kann sich somit auf die Implementierung der eigentlichen Codegenerierung und Analysen konzentrieren, ohne wiederkehrende Aufgaben erneut lösen zu müssen. Die Implementierung als Framework ergibt sich daraus, dass die Ablaufsteuerung zu einer solchen Aufgabe gehört und über die Framework-Architektur gut Erweiterungspunkte zur Verfügung gestellt werden können, die der Programmierer überschreiben kann. Ein auf Basis des Frameworks erstelltes generatives Werkzeug wird dabei als DSLTool bezeichnet. Die Eingabedateien werden innerhalb des DSLTool-Frameworks durch Objekte von Typ `mc.DSLRoot` repräsentiert (kurz: Root-Objekte). Für jede Art von Eingabedatei wird eine entsprechende Unterklasse gebildet, um diese für das Framework voneinander unterscheidbar zu machen. Die Root-Objekte werden von der RootFactory des DSLTools erzeugt, woraufhin sequentiell Algorithmen auf dieser Datenstruktur gestartet werden. Die Abbildung 9.1 zeigt den Zusammenhang zwischen den beteiligten Klassen.

Ein Algorithmus wird in Form einer zustandslosen `mc.ExecutionUnit` implementiert und vom `mc.ADSLToolExecuter` auf den Root-Objekten ausgeführt. Dabei kann zwischen `mc.Workflow`, die ein einzelnes Root-Objekt verarbeitet, und `mc.Pass`, die mehrere (also potentiell alle) Root-Objekte eines Typs verarbeitet, unterschieden werden. Die beiden Klassen verfügen jeweils über eine Hook-Methode, die in den Unterklassen überschrieben wird. Damit stellen sie jeweils eine Strategie im Sinne des gleichnamigen Entwurfsmusters [GHJV95] dar. Die Klassen DSLRoot, ADSLToolExecuter und ExecutionUnit bilden eine Variante des Blackboard-Architekturmusters [BMR+96]. Die Root-Objekte nehmen dabei die Rolle der Datenspeicherung ein. Die ExecutionUnits verändern die Daten und fungieren daher als Informationsquellen, wobei die Ausführungsreihenfolge vom ADSLToolExecuter bestimmt wird, was der Rolle der Kontrolle entspricht. Im Gegensatz zum publizierten Blackboard-Muster ist die Reihenfolge der ExecutionUnits teilweise durch die Parametrisierung vorbestimmt und nicht sonderlich variabel zur Laufzeit, obwohl sich leichte Variationen aufgrund von Nachladen, Erzeugen oder Entfernen von Root-Objekten ergeben können.

Die Abbildung 9.2 zeigt die Klasse `mc.DSLRoot` und ihre direkte Umgebung, die innerhalb der zu implementierenden ExecutionUnits genutzt werden kann. Die Klasse DSLRoot ist dabei generisch, weil der Parameter den Typ des AST festlegt. Jedes Root-Objekt verfügt dabei über einen MontiCoreParser, der zur Erzeugung des AST verwendet werden kann, und einen PrettyPrinter, der den Prozess umkehrt und einen Textstrom aus dem AST erzeugt.

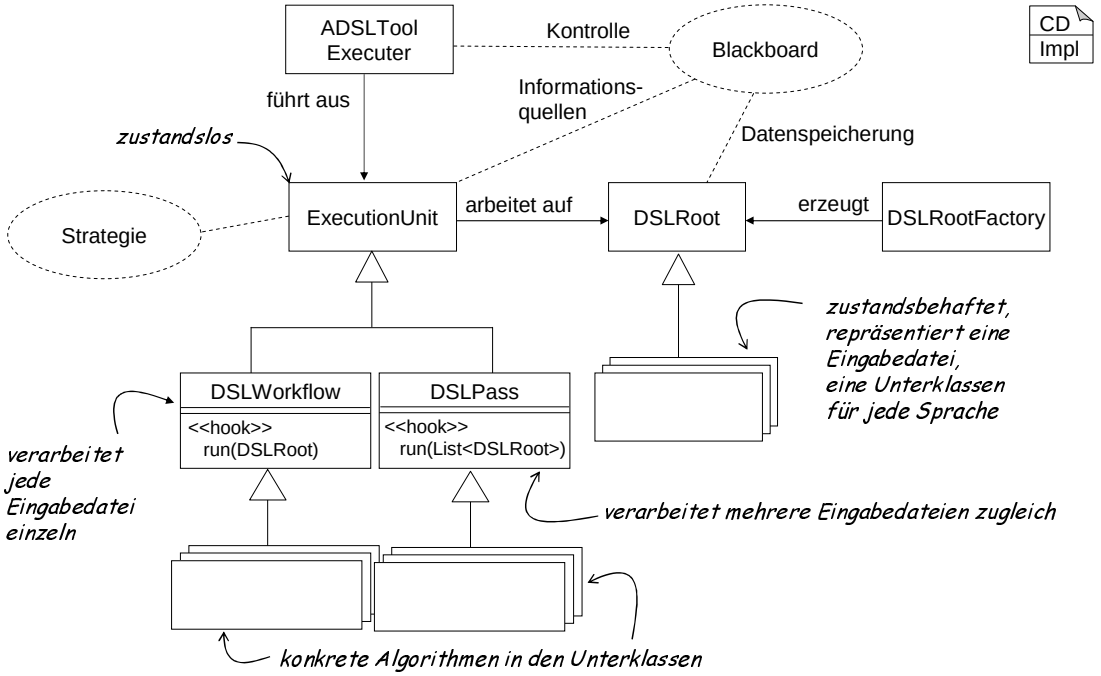


Abbildung 9.1: Zusammenhang zwischen Root-Objekten und ExecutionUnits

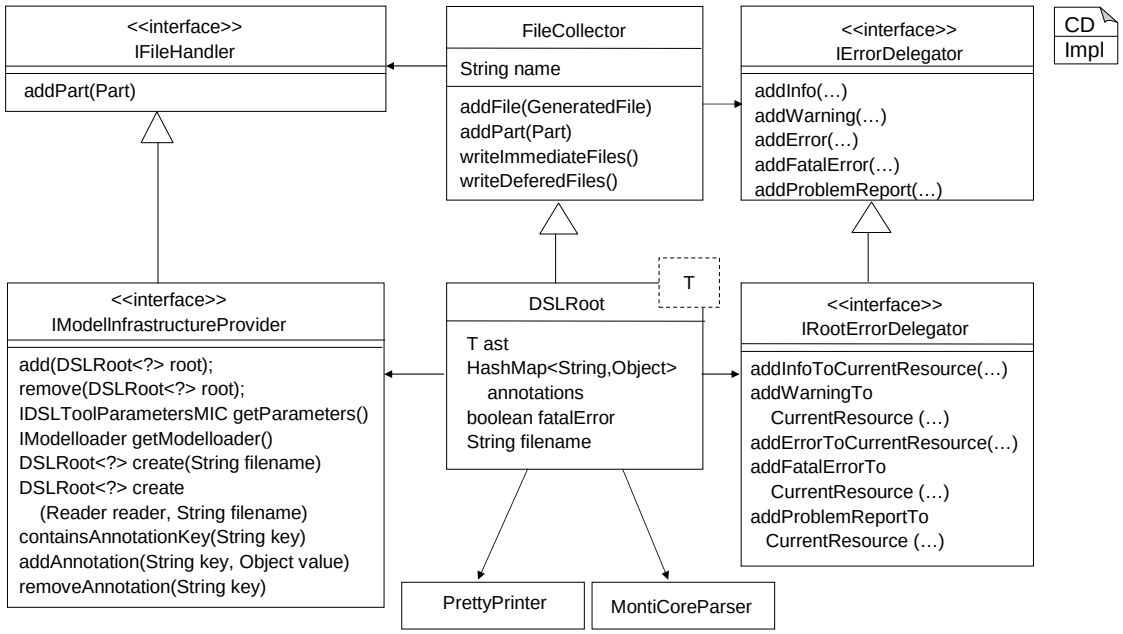


Abbildung 9.2: Umgebung der Klasse `mc.DSLRoot`

Die Root-Objekte werden innerhalb der Architektur als Datenspeicher genutzt. Treten bei einem Algorithmus Zwischenergebnisse auf, die für weitere Verarbeitungsschritte von Bedeutung sind, können diese in den Root-Objekten gespeichert werden. Dabei können in der jeweiligen Unterklasse von DSLRoot-Klassen Attribute hinzugefügt werden, um Zwischenergebnisse typisiert abzulegen. Alternativ können Schlüssel/Wertpaare in den Root-Objekten oder zentral für alle Root-Objekte im ModelInfrastructureProvider (MIP) als so genannte Annotationen abgelegt werden.

Über die Root-Objekte können Fehler gemeldet werden, die dann durch Mechanismen des Frameworks weiterverarbeitet werden. Dafür stehen innerhalb der Schnittstelle `mc.IErrorDelegator` und der Spezialisierung `IRootErrorDelegator` Methoden zur Verfügung, um Status- und Fehlermeldungen sowohl allgemeiner Natur als auch speziell auf das aktuelle Root-Objekt bezogen ausgeben zu können. Fatale Fehlermeldungen setzen das Flag `fatalerror` auf falsch, so dass das Root-Objekt nicht weiter in der Verarbeitung betrachtet wird. Das Attribut `filename` dient dabei zur internen Verarbeitung und hilft bei der Ausgabe der Fehlermeldungen.

Für ein Root-Objekt können durch die vererbten Methoden eines FileCollectors Dateien und so genannte Parts, also Anteile einer Datei, erzeugt werden. Details dazu finden sich in Abschnitt 9.3.5. Über den MIP können andere Modelle aufgrund ihres Namens nachgeladen werden. Hierfür steht der IModelloader über die Methode `getModelloader()` zur Verfügung. Wie in Abschnitt 9.3.1 noch näher erklärt wird, werden während des Nachladens alle ExecutionUnits des Blocks *Analysis* ausgeführt, so dass die resultierende AST zusammen mit Hilfsdatenstrukturen wie Symboltabellen zur Verfügung steht, jedoch für diese Instanz keine Codegenerierung gestartet wird. Es ist innerhalb der Workflows ebenfalls möglich, weitere Root-Objekte zu erzeugen, die dann verarbeitet werden, oder Root-Objekte zu entfernen. Beide Operationen erlauben die Ausgestaltung von Modelltransformationen, bei der ein Modell in ein anderes transformiert wird, das dann weiter vom DSLTool verarbeitet wird, als sei es ein Eingabemodell gewesen.

Der bereits erwähnte MIP stellt die Schnittstellen zwischen den Root-Objekten und ExecutionUnits auf der einen Seite und dem Framework auf der anderen Seite her. Es stellt daher eine Fassade im Sinne des gleichnamigen Entwurfsmusters [GHJV95] dar. Dadurch kann auf die Subsysteme nur durch diese definierte Schnittstelle zugegriffen werden, die den Zugriff auf die internen Strukturen kapselt. Die Abbildung 9.3 zeigt die Architektur eines DSLTools als ein UML-Kompositionsstrukturdiagramm, wobei die einzelnen Systemteile und die vorgesehenen Kommunikationsschnittstellen eingezeichnet sind. Die Systemteile haben dabei die folgenden Verantwortlichkeiten:

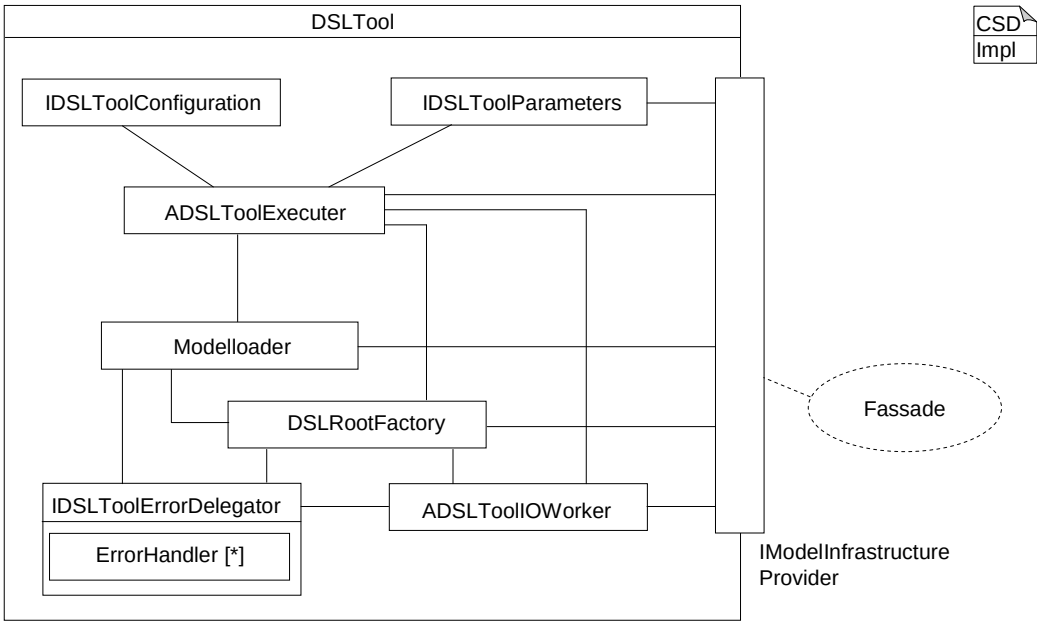


Abbildung 9.3: Architektur eines DSLTools

ADSLToolExcecuter. Der ADSLToolExcecuter koordiniert das Verhalten der anderen Komponenten, prüft die Validität der Parameter und ist für die Ablaufsteuerung zuständig (vgl. Abschnitt 9.3.1).

IDSLToolConfiguration. Die IDSLToolConfiguration verwaltet die einzelnen Execution-Units und RootKlassen, die dieses DSLTool verarbeiten kann und legt somit die validen Parameter fest.

IDSLToolParameters. Die IDSLToolParameters enthalten die Konfiguration des gewählten Durchlaufs durch das Werkzeug sowie Daten wie Eingabedateien und Ausgabeordner.

DSLRootFactory. Die DSLRootFactory ist für die Erzeugung der Root-Objekte aus den Eingabedateien zuständig.

Modelloader. Der Modelloader ist für die Verwaltung der Root-Objekte und das transparente Nachladen zuständig.

ADSLToolIOWorker. Diese Komponente kapselt Schreib- und Lesezugriffe auf Datenquellen, also im Allgemeinen das Dateisystem innerhalb des DSLTools.

IDSLToolErrorDelegator. Diese Komponente ist zentral für die Aggregation von Fehlermeldungen innerhalb des Frameworks zuständig.

ErrorHandler. Die Errorhandler sind für die Ausgabe von Status- und Fehlermeldungen verantwortlich.

IModellInfrastructureProvider. Dieser Port bildet die Schnittstelle des Werkzeugs aus Sicht der Root-Objekte innerhalb der ExecutionUnits.

Bei der Instanzierung eines DSLTools werden die in Abbildung 9.4 dargestellten Standardimplementierungen verwendet. Zur Anpassung an individuelle Bedürfnisse können

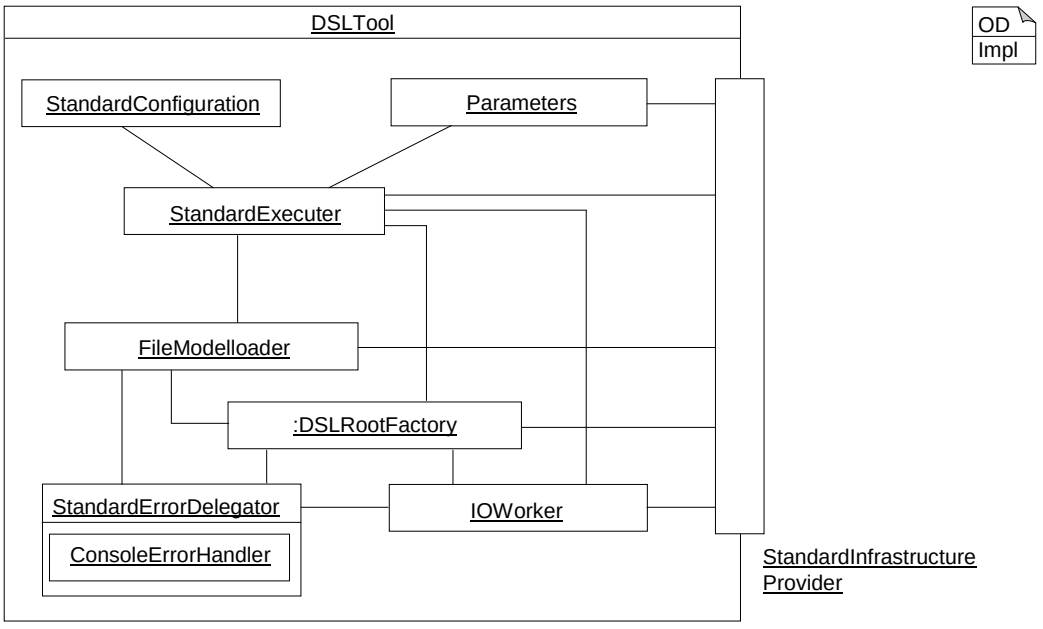


Abbildung 9.4: Standardimplementierungen innerhalb eines DSLTools

Unterklassen vom Typ `mc.DSLTool` gebildet werden. Für jede Komponente existiert eine `initXY()` Methode, wobei `XY` für den Namen der Komponente steht. Dadurch können die einzelnen Standardkomponenten ggf. unterschiedlich parametrisiert werden oder durch geeignete Spezialisierungen ersetzt werden. Einzig die RootFactory muss individuell eingesetzt werden, da diese spezifisch für die verwendeten Sprachen ist. Die Systemteile untereinander sind nicht direkt verbunden, sondern die Verbindungen werden dynamisch bei der Nutzung aufgebaut. Dadurch kann ein DSLTool stets rekonfiguiert werden und die Reihenfolge der Instanzierung spielt keine Rolle.

StandardExcecuter. Diese Klasse StandardExcecuter wendet die in Abschnitt 9.3.1 beschriebene Ablaufsteuerung an.

StandardConfiguration. Diese Komponente verwaltet einzelne Workflows und Root-Klassen, die dieses DSLTool verarbeiten kann.

Parameters. Diese Komponente enthält die Konfiguration des gewählten Durchlaufs und Daten wie Eingabedateien und Ausgabeordner.

FileModelloader. Der FileModelloader lädt Modelle aus den über Kommandozeilenparameter festgelegten Verzeichnissen oder jar-Archiven.

StandardIOWorker. Diese Komponente verwendet die Java-IO-API zum Zugriff auf das Dateisystem.

StandardErrorDelegator. Diese Komponente leitet die Fehlermeldungen an die Error-Handler ohne Modifikation weiter.

ConsoleErrorHandler. Der ConsoleErrorHandler stellt die Status- und Fehlermeldungen in der Konsole dar.

StandardInfrastructureProvider. Der StandardInfrastructureProvider verwaltet generische Name/Wert-Daten und delegiert weitere Aufgaben an die Systemteile des DSL-Tools.

9.3 Funktionalität des Frameworks

In den folgenden Abschnitten werden die identifizierten Querschnittsfunktionalitäten des DSLTool-Frameworks detailliert dargestellt.

9.3.1 Ablaufsteuerung

Ein generatives Werkzeug verarbeitet typischerweise eine Menge an Eingabedateien, die in eine Menge an Ausgabedateien überführt werden. Im Gegensatz zu einem klassischen Compiler werden Dateien mit unterschiedlichem Eingabeformat verarbeitet und die Ausgabe besteht neben GPL-Code oft auch aus Dokumentation, Analyseergebnissen und Testfällen. Daher können in einem DSLTool verschiedene Root-Klassen in der `IDSLToolConfiguration` registriert werden, die jeweils eine Dateiart repräsentieren. Für jede Dateiart können mehrere ExecutionUnits unter jeweils einem bestimmten Namen registriert werden. Namensüberscheidungen für verschiedene Root-Klassen sind unproblematisch und werden sogar beabsichtigt eingesetzt.

Die Parametrisierung eines DSLTools bestimmt die ExecutionUnits der Konfiguration, die innerhalb eines Durchlaufs ausgeführt werden sollen. Die Konfiguration des Werkzeugs

ist von der Ablaufsteuerung entkoppelt, weil so Werkzeuge implementiert werden können, die prinzipiell verschiedene Codegenerierungen und Analysen ausführen können, wenn sie mit verschiedenen Parametern gestartet werden. Ein Beispiel hierfür ist die Angabe in der Parametrisierung des DSLTools, dass der Workflow mit Namen *parse* für alle Root-Objekte verwendet werden soll. Das Framework wählt für die jeweilige passende Dateiart dann den dazugehörigen Workflow aus. Die einzelnen ExecutionUnits sind bereits für eine bestimmte Modellart parametrisiert, insofern ergibt sich die Zuordnung automatisch und ist nicht in der Konfiguration direkt zu erkennen, da die verantwortliche Methode `getResponsibleClass()` bereits implementiert wurde.

Die Abbildung 9.5 zeigt einen Ausschnitt aus einem DSLTool und die Registrierung einer Root-Klasse und der ExecutionUnits. Dabei wird in Zeile 6 zunächst die Standardimplementierung der Konfiguration initialisiert. In Zeile 9 wird die Dateiart „faut“ mit der Rootklasse FlatAutomatonRoot assoziiert, so dass der Namen „faut“ in der Parametrisierung verwendet werden kann. In den Zeilen 12, 15 und 19 werden verschiedene ExecutionUnits unter dem Namen „parse“, „symtab“ und „generate“ registriert.

Java-Quellcode

```
1  public class FlatAutomatonTool extends DSLTool {
2
3    @Override
4    protected void initDSLToolConfiguration() {
5
6      super.initDSLToolConfiguration();
7
8      // Root-Class (can be used as "faut" on the command line)
9      configuration.addDSLRootClassForUserName("faut", FlatAutomatonRoot.class);
10
11     // Parse-Workflow (can be used as "parse" on the command line)
12     configuration.addExecutionUnit("parse", new FlatAutomatonParsingWorkflow());
13
14     // Building up symbol table (can be used as "symtab" on the command line)
15     configuration.addExecutionUnit("symtab",
16        new FlatAutomatonSimpleReferenceSymboltableWorkflow());
17
18     // Code generation (can be used as "generate" on the command line)
19     configuration.addExecutionUnit("generate", new CodegenWorkflow());
20   }
21 }
```

Abbildung 9.5: Konfiguration eines DSLTools

Unter der Ablaufsteuerung wird im Kontext eines DSLTools die Reihenfolge der Ausführung und Auswahl der ExecutionUnits auf den Eingabedateien verstanden. Die Ablaufsteuerung ist in der Klasse `mc.ADSLToolExecuter` gekapselt, die von jedem DSLTool verwendet wird.

Die Ablaufsteuerung kann auf verschiedene Arten parametrisiert werden:

- Programmatisch bei der Initialisierung des DSLTools.
- Durch die Angabe von Kommandozeilenparametern.
- Durch Angabe eines Konfigurationsskripts.
- Durch Programmierung einer Subklasse von `mc.ADSLToolExecuter`.

Die Standardimplementierung `mc.StandardExecuter` verwendet als Eingabeparameter eine Menge an ExecutionUnits, die für jeweils eine Root-Klasse und alle ihre Unterklassen ausgeführt werden. Dabei werden für die ExecutionUnits und die dazu passenden Root-Klassen jeweils die sprechenden Bezeichner verwendet, die in der Konfiguration festgelegt wurden. Dabei können die Workflows zwei Gruppen zugeordnet werden: *Analyses* oder *Syntheses*. Daraufhin werden die Eingabedateien in der Reihenfolge verarbeitet, wie sie dem Werkzeug übergeben werden. Für jede Eingabedatei werden zunächst alle Workflows der Gruppe *Analyses* ausgeführt, dann alle Workflows der Gruppe *Syntheses*, wobei jeweils nur die Workflows verwendet werden, die für diese Root auch registriert sind. Nach Bearbeitung aller Dateien werden die Passes der Gruppe *Syntheses* in der angegebenen Reihenfolge für alle Eingabedateien eines Typs ausgeführt, wobei eine Verwendung von Passes in der Gruppe *Analyses* nicht zulässig ist. Die Abbildung 9.6 zeigt das Verhalten der Standardimplementierung als Sequenzdiagramm.

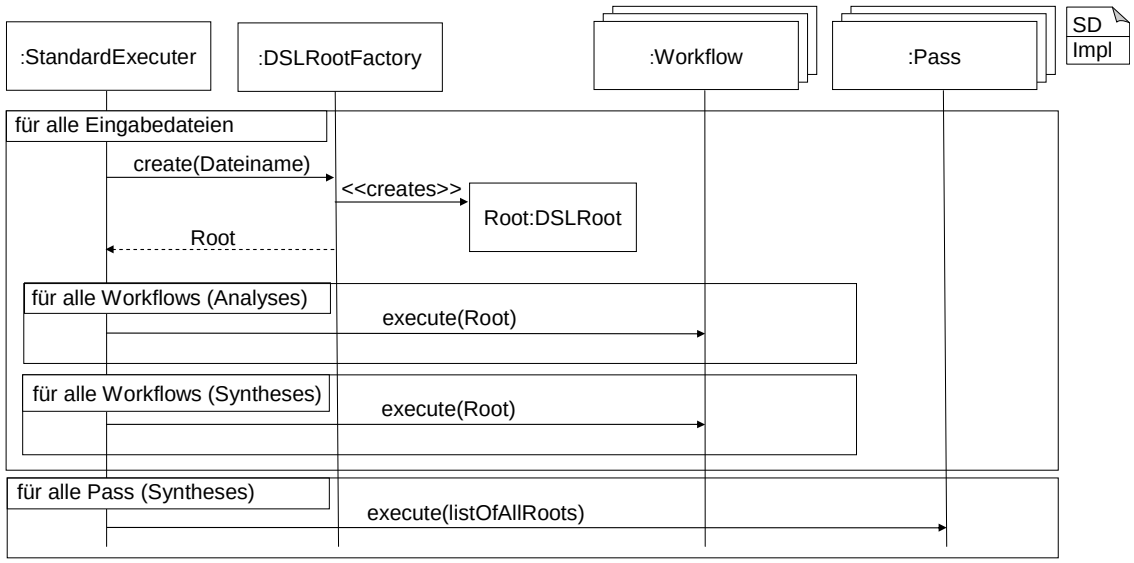


Abbildung 9.6: Verhalten der Klasse `mc.StandardExecuter`

Werden während der Verarbeitung Dateien über den MIP nachgeladen, werden für diese alle Workflows aus der Gruppe *Analyses* ausgeführt. Falls dabei ein Zyklus entsteht, wird dieser vom System erkannt und über eine Exception abgebrochen. Generell gilt, dass eine ExecutionUnit beim Auftreten von fatalen Fehlern normal zu Ende geführt wird, aber für diese Root keine weiteren ExecutionUnits gestartet werden.

Bei der Verarbeitung von Root-Objekten können dynamisch neue Root-Objekte über den MIP instanziert werden. Werden diese dem Werkzeug hinzugefügt, können so fortlaufend Modelltransformationen implementiert werden, wobei die entstandenen Root-Objekte normal von den Workflows der Gruppen *Analyses* und *Syntheses* verarbeitet werden. Werden die Root-Objekte erst innerhalb der Verarbeitung der Passes erzeugt, werden zwar alle Workflows der Gruppen *Analyses* und *Syntheses* ausgeführt, nicht jedoch die bereits ausgeführten Passes für das neue Root-Objekt wiederholt. Beim Entfernen von Root-Objekten über den MIP werden keine weiteren Workflows für dieses Root-Objekt ausgeführt.

Die Verwendung von Passes ist grundsätzlich problematisch, weil hierdurch alle Modelle zur gleichen Zeit verarbeitet werden müssen und kleine Änderungen einzelner Modellen stets zu einer vollständigen Verarbeitung aller Modelle führt, weil sonst den Passes nicht die benötigten Daten zur Verfügung stehen. Daher wird eine Strukturierung eines DSL-

Tools wie in Abschnitt 9.3.5 erklärt und die ausschließliche Verwendung von Workflows empfohlen. Ist einer derartige Strukturierung nicht möglich, kann alternativ die Klasse `mc.PassExecuter` verwendet werden, die Passes auch innerhalb der Gruppe *Syntheses* erlaubt. Dabei wird für jedes Root-Objekt sichergestellt, dass die Reihenfolge der ExecutionUnits der Parametrisierung entspricht und die Passes stets für alle Root-Objekte ausgeführt werden.

Grundsätzlich können Workflows zu komplexeren Workflows mittels des Entwurfsmusters Composite [GHJV95] über die Klasse `mc.CompositeWorkflow` kombiniert werden. Diese komplexeren Workflows können dann normal in die Konfiguration eingebunden werden und erlauben eine Strukturierung der Verarbeitung. Dabei kann es vorkommen, dass verschiedene CompositeWorkflows dieselben elementaren Workflows verwenden, um zum Beispiel benötigte Hilfsdatenstrukturen im Root-Objekt einzulesen. Dabei werden doppelte Berechnungen vermieden und jede ExecutionUnit für dasselbe Root-Objekt nur einmal ausgeführt. Über das Attribut `runOnce` kann eine Mehrfachausführung erreicht werden.

Die Abbildung 9.7 zeigt die Parametrisierung eines DSLTools im Java-Quellcode. Die Anweisung `parameters.addAnalysis(Parameters.ALL,"parse");` bewirkt, dass für jede Modellart eine ExecutionUnit mit Namen `parse` ausgeführt wird, sofern unter diesem Namen eine ExecutionUnit registriert ist. Darüber hinaus bewirkt die Anweisung `parameters.addSynthesis("faut","generate");`, dass für die unter `faut` registrierte Modellart eine ExecutionUnit mit Namen `generate` ausgeführt wird, sofern unter diesem Namen eine ExecutionUnit registriert ist. Für andere Modellarten werden unter `generate` registrierte ExecutionUnits für die Modelle nicht ausgeführt.

Java-Quellcode

```
1  public class FlatAutomatonTool extends DSLTool {
2
3    @Override protected void initStandardParameters() {
4
5      // Default Analyses (when no command line parameters are used)
6      if (parameters.getAnalyses().isEmpty()) {
7        parameters.addAnalysis(Parameters.ALL, "parse");
8        parameters.addAnalysis(Parameters.ALL, "symtab");
9      }
10     // Default Syntheses (when no command line parameters are used)
11     if (parameters.getSyntheses().isEmpty()) {
12       parameters.addSynthesis("faut", "generate");
13     }
14   }
15 }
```

Abbildung 9.7: Parametrisierung eines DSLTools

Alternativ kann das Werkzeug auf der Kommandozeile auf zwei verschiedene Arten aufgerufen werden, die beide dieselbe Bedeutung haben wie die Konfiguration im Java-Quellcode:

- `java FlatAutomatonTool`
  `-parse ALL parse -parse ALL symtab -execute faut generate`
- `java FlatAutomatonTool -conf MyConf.script`
  (Wobei die Konfigurationsdatei aus Abbildung 9.8 verwendet wird.)

DSLTool Konfigurationsskript

```
analyses {
  ALL parse;
  ALL symtab;
}

syntheses {
  faut generate;
}
```

Abbildung 9.8: Konfigurationsskript MyConf.script

9.3.2 Dateierzeugung

Generative Werkzeuge für DSLs erzeugen meistens eine Vielzahl an Dateien. Die Abbildung 9.9 zeigt exemplarisch das Anlegen einer Datei innerhalb eines Workflows und die Registrierung am jeweiligen Root-Objekt. Dabei wurden drei Eigenschaften identifiziert, die ein qualitativ hochwertiger Generator besitzen sollte und daher in der Klasse `GeneratedFile` gekapselt wurden.

- Die Dateierzeugung kann innerhalb des Frameworks verzögert werden, indem die Dateien erst geschrieben werden, wenn alle Verarbeitungsschritte erfolgreich abgeschlossen wurden. Dadurch werden im Fehlerfall unvollständige Generierungen vermieden.
- Mit `mc.helper.JavaFile` existieren Unterklassen von `mc.helper.GeneratedFile`, die automatisch garantieren, dass nur die plattformspezifischen Zeilenumbrüche verwendet werden und ein einheitlicher Kommentarkopf über den erzeugten Dateien generiert wird.
- Die Verwendung von `mc.helper.GeneratedFile` prüft vor dem Schreiben einer Datei, ob nicht bereits eine Datei mit demselben Inhalt vorhanden ist. Dieser Prüfungs-

Java-Quellcode

```
public class CodegenWorkflow extends DSLWorkflow<FlatAutomatonRoot> {

  // ...

  @Override
  public void run(FlatAutomatonRoot dslroot) {

    // Creates file First.java in package automaton
    GeneratedFile f = new GeneratedFile("automaton", "First.aut");

    // writes "test!" in file
    f.getContent().append("test!");

    // Add file for writing to root
    dslroot.addFile(f);
  }
}
```

Abbildung 9.9: Dateierzeugung innerhalb des DSLTool-Frameworks

schritt verhindert, dass in aufeinander folgenden Aufrufen des Generators inhaltlich gleiche Dateien nochmals geschrieben werden. Verwendet man den Generator zur Erzeugung von GPL-Code, lässt sich so eine erhebliche Performance-Steigerung erreichen, weil beim nachfolgenden Compilerdurchlauf nur die veränderten Dateien kompiliert werden müssen und diese Zeitersparnis den möglichen Performanceverlust durch den zusätzlichen Lesezugriff deutlich übersteigt.

Zur Bestimmung des Effekts der dritten Eigenschaft wurden Messungen am MontiCore-Generator durchgeführt, der auf dem DSLTool-Framework basiert: Es wurden 45 Grammatiken in einem Durchlauf verarbeitet, aus denen 1097 Java-Klassen oder -Schnittstellen erzeugt wurden. Bei einem gefüllten Zielverzeichnis zeigt der Generator eine durchschnittliche Laufzeit von 14,008s für das Einlesen und das anschließende Schreiben der Dateien. Die Strategie wurde dabei künstlich deaktiviert, indem auch bei gleichen Dateiinhalten ein Schreibvorgang erzwungen wurde, um den Effekt der zusätzlichen Lesevorgänge gezielt bestimmen zu können. Ohne die Lesevorgänge wurde eine durchschnittliche Laufzeit von 13,922s bestimmt. Bei gefülltem Zielverzeichnis und keinen Schreibzugriffen, weil die Dateien identisch waren, war die Überprüfung mit durchschnittlich 12,884s sogar schneller. Somit ergibt sich ein zusätzlicher Aufwand für die zusätzlichen Lesevorgänge von 0,6% der Generatorlaufzeit im ungünstigsten Fall, aber ein möglicher Performancegewinn von 7,5% bei unveränderten Zieldateien.

Beim anschließenden Compilerdurchlauf werden nur die veränderten Dateien übersetzt, wobei die Prüfung automatisch durch den Javac selbst erfolgt. Für die Prüfung benötigt er bei 1097 Java-Klassen oder -Schnittstellen durchschnittliche 829ms, wobei ein vollständiger Übersetzungvorgang durchschnittlich 7,519s dauert. Dadurch ergibt sich ingesamt ein maximaler Performance-Verlust von 0,4% und ein maximaler Performancegewinn von 36% bezogen auf die Gesamtlaufzeit von Generator und Compiler. Daher wird die Strategie im DSLTool-Framework standardmäßig verwendet, kann jedoch durch die Option `nocache` deaktiviert werden.

Zur Bestimmung der Laufzeit wurde die Systemuhr verwendet und der Mittelwert von 10 Durchläufen gebildet. Die Messung erfolgte auf einem PC mit Intel Core2 Duo CPU E6750 mit 2,67GHz Taktrate, Windows XP mit installiertem JDK 1.6.0_03 und 512MB zugesicherte Speicher für die VM und den Compiler. Die Quelldateien von MontiCore wurden mit dem Javac 1.6.0_03 übersetzt.

9.3.3 Fehlermeldungen

Für die Ausgabe von Fehlermeldungen wurden im DSLTool-Framework zwei Schnittstellen festgelegt. Die Schnittstelle `mc.IErrorDelegator` stellt Methoden zur Verfügung, die Fehlermeldungen innerhalb des Frameworks erlauben, wohingegen `mc.IRootErrorDelegator` zusätzlich Methoden hinzufügt, die Fehlermeldungen direkt für das aktuelle Root-Objekt erlauben (vgl. Abbildung 9.2). Innerhalb des Frameworks wird eine Fehlermeldung durch die Klasse `mc.ProblemReport` dargestellt. Die Klasse und ihre Attribute sind in Abbildung 9.10 dargestellt. Der dabei verwendete eindeutige Bezeichner soll die Testfallerstellung für generative Werkzeuge vereinfachen, weil sich so eine Fehlermeldung direkt identifizieren lässt, was die Implementierung von negativen Testfällen und die Prüfung vereinfacht, ob erwartete Fehlermeldungen erzeugt wurden.

Die Abbildung 9.11 zeigt, wie ein fataler Fehler innerhalb eines Workflows ausgelöst werden kann. Dieser Aufruf stoppt die weitere Verarbeitung des Root-Objekts, das heißt, die Ausführung weiterer ExecutionUnits für dieses Root-Objekt wird verhindert.

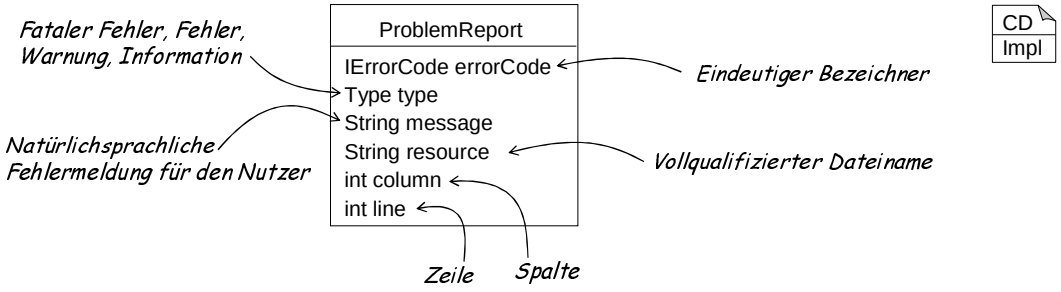


Abbildung 9.10: Attribute der Klasse `mc.ProblemReport`

Nach der Registrierung der Fehlermeldungen durch `mc.IErrorDelegator` oder aber einen verfügbaren `mc.IRootErrorDelegator` wird die Fehlermeldung auf eine einzige Methode innerhalb der abstrakten Klasse `mc.IErrorHandler` abgebildet. Es existieren für verschiedene Plattformen verschiedene Implementierungen, die für weitere Verarbeitung und meistens die Ausgabe der Fehlermeldungen sorgen.

`mc.SimpleErrorhandler:` Die Fehlermeldungen werden in internen Datenstrukturen verwaltet, so dass sich diese Klasse zur Ausgestaltung von Testfällen eignet.

`abstractTextEditor.editor.AbstractTextEditor:` Die Fehlermeldungen werden in der Problems-View des Eclipse-Frameworks dargestellt.

`mc.ConsoleErrorhandler:` Die Fehlermeldungen werden in der Konsole angezeigt.

9.3.4 Funktionale API

Im Gegensatz zur objektorientierten Programmierung, bei der Daten und sie manipulierende Methoden eine Einheit bilden, sind diese beiden Elemente in der funktionalen Programmierung entkoppelt. Daher können Funktionen als eigenständige Elemente der Sprache verwendet werden. Der funktionale Ansatz ist an manchen Stellen innerhalb der objektorientierten Entwicklung nützlich, was sich zum Beispiel in Programmiersprachen wie Scala [Sca] zeigt. Andere Programmiersprachen verwenden Funktionszeiger (C++) und Delegates (C#) als Hilfskonstrukte.

Java-Quellcode

```
1 public class CodegenWorkflow extends DSLWorkflow<FlatAutomatonRoot> {
2
3   // ...
4
5   @Override
6   public void run(FlatAutomatonRoot dslroot) {
7
8     dslroot.getErrorDelegator().
9       addFatalErrorToCurrentResource("test error", AutomatonErrorCode.Test);
10  }
11 }
```

Abbildung 9.11: Melden eines fatalen Fehlers innerhalb eines Workflows

Innerhalb des MontiCore-Frameworks wurde eine API mit funktionaler Ausrichtung realisiert, die sich an der Verwendung von Delegates in C# orientiert. Das zentrale Element ist dabei die generische Schnittstelle `mc.transform.MCDelegate<A,R>`, die eine Funktion mit einem Parameter von `A` nach `R` darstellt. Dazu besitzt jedes Objekt vom Typ `MCDelegate<A,R>` eine Methode `t(A)` mit Rückgabe `R`. Innerhalb der Entwicklung können dann implementierende Klassen implementiert werden, die die jeweiligen Funktionen realisieren. Mit `MCDelegate2<A,B,R>` und `MCDelegate3<A,B,C,R>` stehen auch mehrstellige Funktionen zur Verfügung. Objekte vom Typ `MCDelegate2<A,B,R>` haben entsprechend eine Methode `t(A,B)` mit Rückgabe `R`. Zusätzlich kann Funktion partiell ausgewertet werden, indem nur das erste Argument festgelegt wird und somit eine Funktion mit nur einem Argument entsteht. Dazu existiert die Methode `t(A)` vom Typ `MCDelegate<B,R>`. Die Abbildung 9.12 zeigt einzelnen Klassen und Schnittstellen im Überblick.

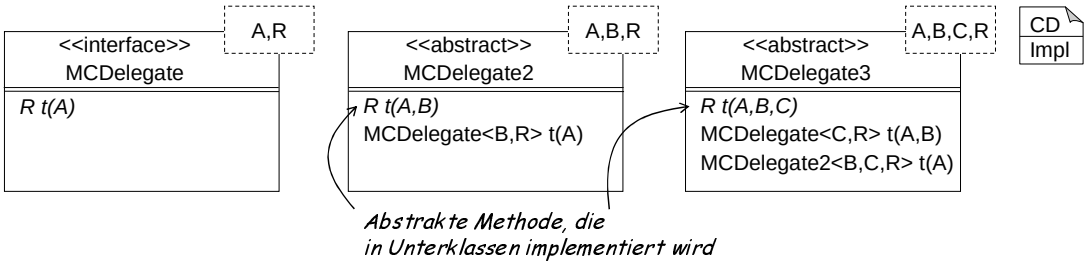


Abbildung 9.12: Übersicht über die MCDelegates

Aufbauend auf der Darstellung von Funktionen als Objekte vom Typ `MCDelegate<A,R>`, lassen sich weiterführende Algorithmen implementieren, die in der MontiCore Transformations-API im Paket `mc.transform` zusammengefasst sind. Dabei kann per Konvention jede Funktion mit der statischen Methode `f()` instanziert. Zusätzlich stehen mit `f(...)` verschiedene durch partielle Auswertung entstandene Versionen zur Verfügung. Die Verwendung von statischen Funktionen gegenüber der Verwendung von dynamischer Objekterzeugung und Fabriken hat drei Gründe: Erstens werden die Formeln dadurch kürzer. Zweitens funktioniert die Typinferenz bei statischen Methoden in weit mehr Fällen als bei dynamisch instanzierten Objekten, wodurch meistens auf explizite Typparameter verzichtet werden kann, was die Lesbarkeit erhöht (vgl. [GJS05, §15.12.2.7]). Drittens können IDEs den Typ einer Funktion automatisch ermitteln, was die praktische Programmierung von Funktionen stark vereinfacht.

Die realisierten Funktionen orientieren sich dabei grob an der Standardbibliothek prelude der Programmiersprache Haskell [Tho97]. Dadurch werden dem Entwickler Funktionen höherer Ordnung zur Verfügung gestellt, also solche, die Funktionen als Argumente verwenden oder Funktionen als Rückgabetyp haben. Die Schreibweise in der folgenden Übersicht wurde übernommen, so dass beispielsweise die Funktion `MyF: (X → Y) → X → Y` zwei Parameter hat: Erstens eine Funktion von `X` nach `Y` (`MCDelegegate<X,Y>`) und zweitens einen weiteren Parameter vom Typ `X`. Der Rückgabewert ist vom Typ `Y`. In Abbildung 9.13 zeigt die Realisierung dieser Funtkion als Klassendiagramm. Die folgenden Funktionen sind ein Ausschnitt der Transformations-API im Paket `mc.transform`.

`All: (? super T → Boolean) → Collection<T> → Boolean:` Die Funktion All wendet ein Prädikat auf alle Elemente einer Liste an und liefert genau dann wahr, wenn das Prädikat für alle Elemente wahr ist.

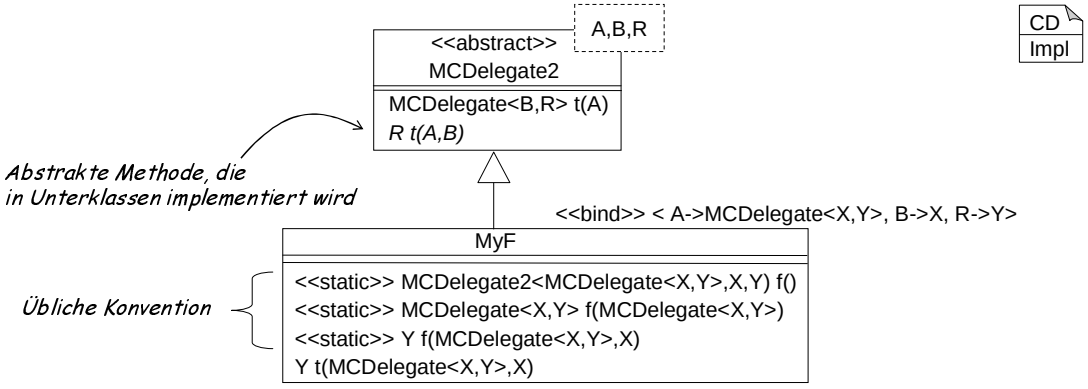


Abbildung 9.13: Klassendiagramm der Funktion `MyF`

`Any: (? super T → Boolean) → Collection<T> → Boolean:` Die Funktion Any wendet ein Prädikat auf alle Elemente einer Liste an und liefert genau dann wahr, wenn das Prädikat für mindestens ein Element wahr ist.

`ExactlyOne: (? super T → Boolean) → Collection<T> → Boolean:` Die Funktion ExactlyOne wendet ein Prädikat auf alle Elemente einer Liste an und liefert genau dann wahr, wenn das Prädikat für genau ein Element wahr ist.

`Filter: (? super T → Boolean) → Collection<T> → Collection<T>:` Die Funktion Filter wendet ein Prädikat auf alle Elemente einer Liste an und belässt nur die Elemente im Ergebnis, für die das Prädikat wahr ist.

`FilterAST: (ASTNode → Boolean) → ASTNode → Collection<ASTNode>:` Die Funktion FilterAST wendet ein Prädikat auf die Knoten einer AST-Baumstruktur an und erzeugt eine Liste mit Elementen, für die das Prädikat wahr ist.

`FilterFirst: (? super T → Boolean) → Collection<T> → T:` Die Funktion FilterFirst wendet ein Prädikat auf alle Elemente einer Liste an und liefert das erste Element, für das das Prädikat wahr ist.

`FilterFirstAST: (ASTNode → Boolean) → ASTNode → ASTNode:` Die Funktion FilterFirstAST wendet ein Prädikat auf die Knoten einer AST-Baumstruktur an und liefert das erste Element in Preorder-Reihenfolge, für das das Prädikat wahr ist.

`FoldL: (A → B → A) → A → List<B> → A:` Die Funktion FoldL wendet eine zweistellige Funktion zunächst auf einen Startwert (als linkes Argument) und das erste Element einer Liste an. Dann wird wiederholt das Ergebnis und das nächste Element der Liste verwendet, bis alle Elemente der Liste benutzt wurden.

`FoldR: (A → B → B) → B → List<A> → B:` Die Funktion FoldR wendet eine zweistellige Funktion zunächst auf einen Startwert (als rechtes Argument) und das letzte Element einer Liste an. Dann wird wiederholt das Ergebnis und das nächste Element der Liste (von hinten) verwendet, bis alle Elemente der Liste benutzt wurden.

`MCMap:(? super A → B) → Collection<A> → Collection<B>:` Die Funktion MCMap wendet eine Funktion elementweise auf eine Collection an. Die Namensgebung Map wurde aufgrund des Namenskonflikts mit `java.util.Map` vermieden, um qualifizierte Imports zu vermeiden.

`MCMapper:(? super A → B) → Collection<A> → Map<A,B>:` Die Funktion MCMapper wendet eine Funktion elementweise auf eine Liste an. Die Ausgangswerte und die Ergebnisse werden als Werte in einer `java.util.Map` abgelegt.

`Seq : (A → ? extends B) → (? super B → C) → (A → C):` Die Funktion Seq bildet eine neue Funktion aus der Hintereinanderschaltung zweier Funktionen mit passenenden Parametern und Typ.

`TransitiveClosure : (? super A → ? extends A) → Collection<A> → Set<A>:` Die Funktion TransitiveClosure wendet eine Funktion so lange auf alle Elemente einer Menge an und fügt somit neue Elemente hinzu, bis die Menge stationär wird.

`TypeMCMatcher: Class<? extends A> → Object → Boolean:` Das Prädikat TypeMCMatcher liefert den Wahrheitswert true für alle Elemente, die denselben Typ oder einen Subtyp einer gegebenen Klasse haben.

`Zip: List<A> → List<B> → List<Pair<A,B»:` Diese Funktion bildet Paare aus zwei gleichlangen Listen.

Die Abbildung 9.14 zeigt ein Beispiel zur Erstellung einer Hilfsdatenstruktur innerhalb einer Codegenerierung von endlichen Automaten. Die komplette Codegenerierung findet sich im bei MontiCore mitgelieferten Beispiel Automaton. Das Ziel des gezeigten Ausschnitts ist es, eine Hilfsdatenstruktur vom Typ `java.util.Map` zu erzeugen, die einen Zustandsnamen auf eine Menge von Superzuständen inklusive des eigenen Zustands abbildet. Dazu wird zunächst in den Zeilen 8-12 die Menge aller Zustände bestimmt. Zusätzlich werden zwei Hilfsfunktionen definiert: Eine, die einen Zustand auf seinen Namen (Zeilen 15-20), und eine zweite, die einen Zustand auf den direkten Superzustand abbildet (Zeilen 23-35). Dabei wird zweifach die `getParent()`-Methoden aufgerufen, weil die Zustände jeweils in Listen enthalten sind. Aufbauend auf diesen Hilfsfunktionen wird in den Zeilen 38-46 eine komplexe Funktion aufgebaut und und in Zeile 47 angewendet. Dabei wird aus der Collection von States zunächst eine Map mittels `MCMapper` gemacht, wobei die Schlüssel die Zustände sind und die Werte sich aus einer sequentiellen Funktionsausführung dreier Funktionen ergeben: Erstens der Überführung eines Elements in eine einelementige Menge (`Singleton`, Zeile 41), zweitens der transitiven Anwendung (`TransitiveClosure`, Zeile 42) der `ChildToParentState`-Funktion und drittens der Umwandlung der Zustände in ihre Namen durch `MCMap` und `StateToStateName` in Zeile 43. Abschließend werden die Schlüssel der Map in die Zustandsnamen durch die Funktion `TransformKeys` umgewandelt (Zeile 45). Zur sequentiellen Funktionsausführung wird jeweils die Funktion `Seq` verwendet.

Die Abbildung 9.15 zeigt die Verwendung einiger Funktionen zum Prüfen von Kontextbedingungen. In den Zeilen 7/8 werden zunächst die Objekte vom Typ `ASTState` aus dem AST herausgefiltert. Dabei wird das Prädikat `TypeMCMatcher` verwendet, das für alle Klassen und Subklassen einer gegebenen Klasse wahr zurückliefert. In den Zeilen 10-14 wird zunächst ein Prädikat programmiert, das prüft, ob ein Zustand initial ist. Dann wird in Zeile 16 dieses Prädikat zusammen mit der Funktion `Filter` genutzt, um nur die initialen Zuständen aus der Liste aller Zustände herauszusuchen. In der Zeile 18 wird geprüft, ob kein initialer Zustand vorhanden ist, um in den Zeilen 19/20 die entsprechende Fehlermeldung auszugeben. In der Zeile 21 wird weitergehend geprüft, ob mehr als ein initialer Zustand vorhanden ist. Falls ja, wird in den Zeilen 23/24 die Funktion `ReportFatalErrorForNode` zusammen mit `MCMap` auf alle Zustände angewendet, um eine Fehlermeldung zu liefern. Die ReportFatalErrorToRoot hat dabei drei Parameter: Ein Root-Objekt, eine Fehlermeldung und einen AST-Knoten, der zur Positionsbestimmung verwendet wird. Durch die statische Funktion `f(...)` wird die Funktion partiell ausgewertet, so dass eine einstellige Funktion

Java-Quellcode

```
1  public class ExecuteWorkflow2 extends DSLWorkflow<AutomatonRoot> {
2
3    // ...
4
5    @Override public void run(AutomatonRoot dslroot) {
6
7      // List of all states
8      List<ASTState> states =
9        FilterAST.f(
10         TypeMCMatcher.<ASTNode>f(ASTState.class),
11         dslroot.getAst()
12       );
13
14     // Maps a state to its name
15     class StateToStateName implements MCDelegate<ASTState, String> {
16
17       public String t(ASTState s) {
18         return s.getName();
19       }
20     }
21
22     // Maps a state to its parent
23     class ChildToParentState implements MCDelegate<ASTState, ASTState> {
24
25       public ASTState t(ASTState s) {
26
27         ASTNode sn = s.get_Parent().get_Parent();
28         if (sn != null && sn instanceof ASTState) {
29           return (ASTState) sn;
30         }
31         else {
32           return null;
33         }
34       }
35     }
36
37     Map<String, Collection<String>> map =
38       Seq.f(
39         MCMapper.f(
40           Seq.f(Seq.f(
41             Singleton.<ASTState>f(),
42             TransitiveClosure.f(new ChildToParentState())),
43             MCMap.f(new StateToStateName()))
44           ),
45         TransformKeys.<ASTState, String, Collection<String>>f(new StateToStateName())
46         )
47         .t(states);
48   }
49
50 }
```

Abbildung 9.14: Verwendung der MontiCore Transformations-API zur Erstellung von Hilfsdatenstrukturen

mit einzigen Parameter vom Typ `ASTNode` entsteht. Der Rückgabewert ist dabei `void`, da ausschließlich der Seiteneffekt (die Auslösung der Fehlermeldung) interessant ist.

9.3.5 Inkrementelle Codegenerierung

Die Compilierung von Software-Systemen wurde in den letzten Jahren durch die inkrementelle Compilierung des Quellcodes innerhalb der Entwicklungsumgebungen für die Programmierer deutlich beschleunigt. Dabei wird die Software jeweils inkrementell übersetzt, so dass der Compiler nur die Änderungen zum letzten Übersetzungsschritt betrachtet. Durch dieses Vorgehen wird die Compilierung beim ersten Mal meistens langsamer, weil die Compiler in dieser Richtung nicht optimiert sind. Da bei der Arbeit nur fortlaufend kleine Deltas an einem Projekt erstellt werden und der Quellcode häufig übersetzt wird, kann die Wartezeit insgesamt jedoch reduziert werden.

Eine Übertragung der existierenden Techniken für Programmiersprachen auf DSLs ist sehr kompliziert, weil das Schreiben von solchen Compilern spezifisch für die jeweilige Sprache ist und sich der Aufwand für die Entwicklung so deutlich erhöht. Das Vorgehen wird insbesondere dadurch erschwert, dass DSLs nicht modular sind und damit vielfältige Abhängigkeiten zwischen den Modellen und dem generierten Code existieren. Als eine Art Zwischenlösung sollen die Generatoren mit dem DSLTool-Framework dennoch so geschrieben werden, dass nur veränderte Quellcodedateien betrachtet werden müssen und nicht das gesamte Projekt. Konkret soll für einen mit dem DSLTool-Framework entwickelten

Java-Quellcode

```
1  public class ExecuteWorkflow extends DSLWorkflow<AutomatonRoot> {
2
3    // ...
4
5    @Override public void run(AutomatonRoot dslroot) {
6
7      List<ASTState> states =
8        FilterAST.f(TypeMCMatcher.<ASTNode>f(ASTState.class), dslroot.getAst());
9
10     final MCDelegate<ASTState, Boolean> isInitial = new MCPredicate<ASTState>() {
11       public Boolean t(ASTState s) {
12         return s.isInitial();
13       }
14     };
15
16     Collection<? extends ASTState> initialstates = Filter.a(isInitial, states);
17
18     if (initialstates.isEmpty()) {
19       ReportFatalErrorToRoot.f(dslroot, "Automaton needs an initial state!");
20     }
21     else if (initialstates.size() > 1) {
22       MCMap.a(
23         ReportFatalErrorForNode.f(dslroot, "Automaton has more than one initial state!"),
24         initialstates);
25     }
26   }
27 }
```

Abbildung 9.15: Verwendung der MontiCore Transformations-API zur Implementierung von Kontextbedingungsprüfungen

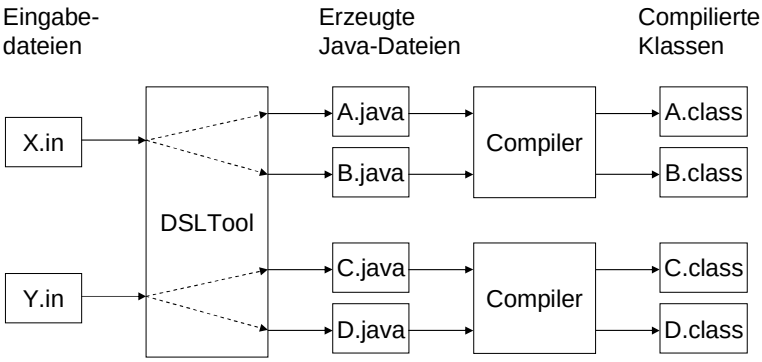


Abbildung 9.16: Unabhängige Codegenerierung aus den Eingabedateien X und Y

Generator gelten, dass das generierte Endergebnis dasselbe ist, wenn der Generator für alle Dateien gleichzeitig oder nacheinander für alle Eingabedateien aufgerufen wird.

Im einfachsten Fall werden aus jeder Eingabedatei eine oder mehrere Ausgabedateien erzeugt, die weitestgehend unabhängig voneinander sind. Die dadurch zu erreichende Form der unabhängigen Codegenerierung hängt von der Kompositionsform der weiteren Verarbeitungswerkzeuge ab. Handelt es sich bei den erzeugten Dateien um Java-Quellcode, können die verschiedenen erzeugten Klassen gegenseitig über Methodenaufrufe und Instanzierung miteinander kommunizieren, auch wenn sie unabhängig voneinander erzeugt wurden. Diese Form der Codegenerierung wird auch in [WK06] bevorzugt, weil so ein monolithischer Algorithmus mit langer Laufzeit vermieden wird. C# erlaubt weitergehend auch eine Kompositionalität innerhalb einer Klasse, indem eine Klasse über mehrere Dateien verteilt werden kann, was vor allem für die Mischung aus generierten GUI-Klassen und manuell programmierter Logik genutzt wird. Werden ausdrucksmächtigere Generierungsziele wie aspektorientierte Programmiersprachen oder Sprachen wie XMF [CSW08] oder Converge [Tra08] als Zielsprache (host language) verwendet, dann ist eine unabhängige Codeerzeugung aus Modellen leichter möglich, weil der generierte Quellcode auf vielfältige Weisen interagieren kann. Die Abbildung 9.16 zeigt eine Übersicht über diese Form der Codegenerierung. In diesem Beispiel könnten die Eingabedateien X und Y auch unabhängig voneinander verarbeitet werden und das generierte Ergebnis wäre dasselbe.

Kennen sich die Klassen über gegenseitige Namen, lassen diese sich zwar unabhängig voneinander erzeugen, jedoch nicht kompilieren. Die Abbildung 9.17 zeigt einen Spezialfall, in dem eine Laufzeitumgebung abstrakte Oberklassen oder Schnittstellen enthält, die die prinzipiellen Interaktion zwischen den generierten Klassen beschreibt. Dadurch werden die Klassen auch einzeln übersetzbar.

Es gibt jedoch Ziele für die Generierung von Software, die keine Form von Modularität und Verteilen auf mehrere Dateien zulassen. Beispiele hierfür sind Konfigurationsdateien wie die Manifest.mf von Java-Archiven und plugin.xml für Eclipse-Plugins. Hier ist der Idealfall, die Kompositionalität erst beim Laden herzustellen, nicht erreichbar. Damit hier nacheinander verschiedene Werkzeugdurchläufe dasselbe Ergebnis liefern, müsste der generierte Inhalt von der Codegenerierung verstanden werden und in einer Art Round-Trip-Engineering zurückgeführt werden, was technisch jedoch sehr aufwändig und bei Veränderungen des Generators praktisch kaum möglich ist. Daher wurde innerhalb des Frameworks ein anderer Weg beschritten, der es im Gegensatz zu [WK06] ermöglicht, auch für solche Fälle eine inkrementelle Erzeugung zu erreichen: In der ersten Phase werden aus den Eingabedateien jeweils die Informationen extrahiert, die für die Erzeugung einer bestimmten Ausgabedatei notwendig sind. Diese Informationen werden innerhalb eines Objekts vom

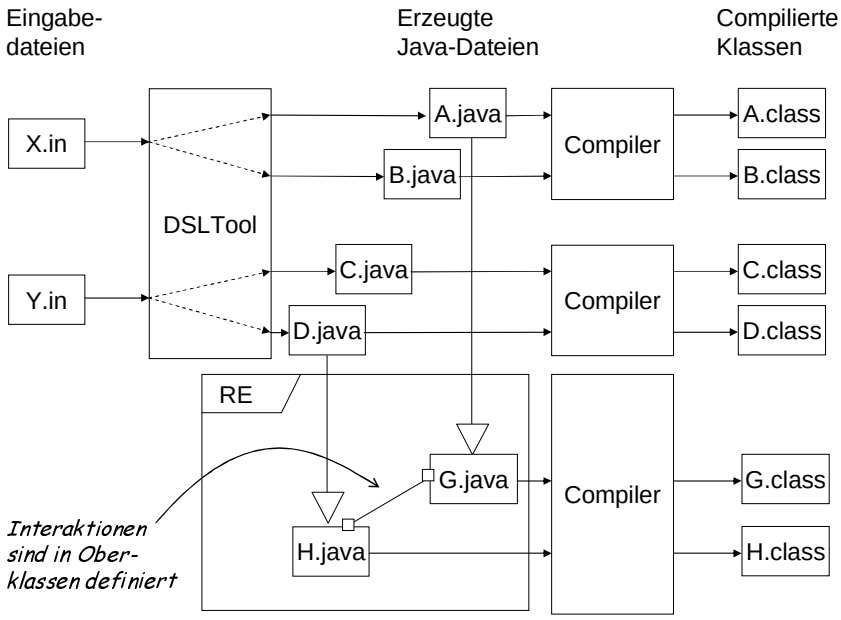


Abbildung 9.17: Verwendung einer Laufzeitumgebung zur unabhängigen Compilierung

Typ `mc.Part` gespeichert. Das DSLTool-Framework serialisiert diese Part-Dateien innerhalb einer Zieldatei und stellt einem PartMerger die Menge aller Parts für eine Zieldatei zur Verfügung. Dieses können Parts aus dem aktuellen Werkzeugdurchlauf sein oder aber auch teilweise aus früheren Verarbeitungsschritten entstandene Parts. Aus diesen wird dann stets auf dieselbe Weise die Zieldatei erzeugt.

Die Abbildung 9.18 zeigt exemplarisch das Vorgehen bei der Verwendung von Parts. Dabei werden im ersten Durchlauf vom DSLTool die Modelle `X.in` und `Y.in` insofern verarbeitet, als die wesentlichen Informationen extrahiert werden und als Parts `X.extr` und `Y.extr` vorkommen. Die Parts werden in der Datei `Manifest.mf` zusammengefügt und in der Partdatei `Manifest.mf.part` serialisiert. Vor den zweiten Verarbeitungsschritt wurde nur das Modell `X.in` verändert, so dass das Werkzeug nur `X.in` und die Datei `Manifest.mf.part` verarbeitet. Die Entscheidung, welche Modelle verändert wurden, kann durch durch Buildskripte automatisiert werden, die zum Beispiel auf [Ant] basieren. Aus den Eingaben werden wieder die Parts extrahiert bzw. erstellt und mit demselben Algorithmus wie im ersten Durchlauf zur Datei `Manifest.mf` zusammengefügt. Die veränderten Extrakte werden in der Partdatei `Manifest.mf.part` in serialisierter Form eingefügt.

Innerhalb des MontiCore-Frameworks werden die PartMerger nach allen Execution-Units ausgeführt, wobei es unterschiedliche Arten von Parts geben kann, die unterschiedliche PartMerger verwenden. Diese werden analog zu den Root-Klassen im Framework registriert und nach allen ExcutionUnits ausgeführt. Die Serialisierung/Deserialisierung erfolgt am einfachsten durch Nutzung der Java-Serialisierung, wenn die Klasse `mc.parts.SerializablePart<Content>` verwendet wird.

Das hier beschriebene Verfahren verwendet die Partdatei als eine Art Container, der die Extrakte der Eingabemodelle enthält. Dadurch kommt es formal zu einem Kreisschluss, weil die Partdatei innerhalb eines Verarbeitungsschritts sowohl gelesen als auch verändert werden kann. Daher sollten die Partdateien nicht zur Steuerung des Buildprozesses durch Abhängigkeiten verwendet werden, sondern ausschließlich die erzeugten Artefakte. Dies ist stets möglich, da zwischen beiden Artefakte eine Eins-Zu-Eins-Beziehung besteht und beide gleichzeitig modifiziert werden. Der Kreisschluss ist dabei niemals zwischen den Extrak-

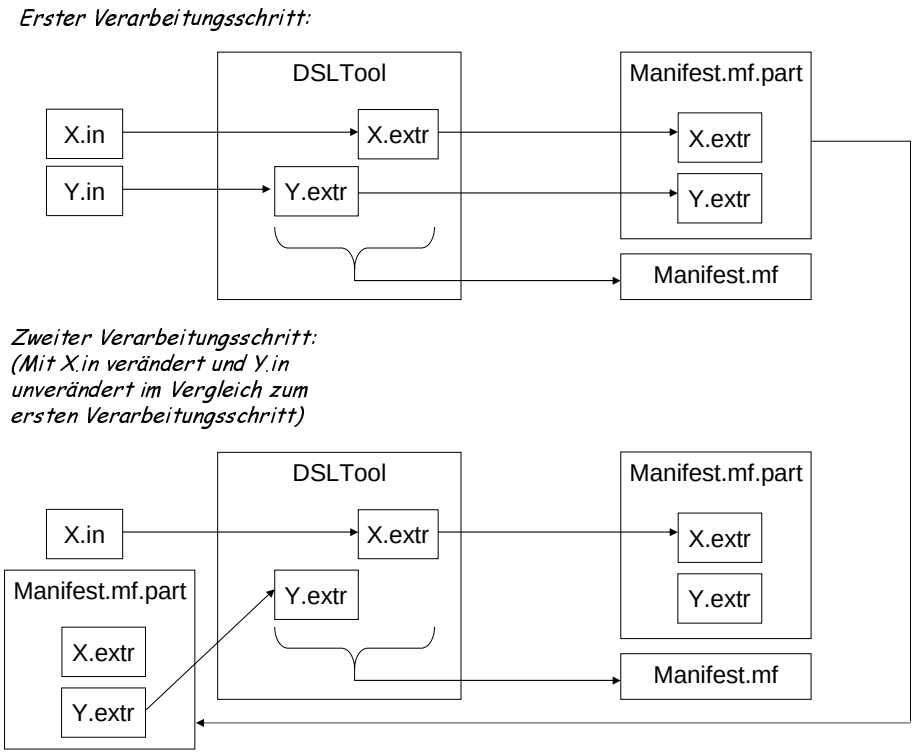


Abbildung 9.18: Verwenden von Parts innerhalb einer Codegenerierung

ten vorhanden, da ein Extrakt entweder neu erzeugt wird, weil das Modell neu eingelesen wird und daher nicht gelesen wird, oder aus der Partdatei eingelesen wird, weil das Modell unverändert ist und daher nicht geschrieben wird. Das hier beschriebene Verfahren könnte leicht modifiziert werden, indem keine Container-Datei verwendet wird, sondern jedes Extrakt in einer eigene Datei gespeichert wird. Dadurch kann der Kreisschluss gänzlich vermieden werden. Auf eine solche Realisierung wurde verzichtet, weil der Name des Eingabemodells dann im Dateinamen des Extrakts codiert werden müsste und dabei die zulässige Länge eines Dateinamens schnell überschritten werden könnte. Außerdem wird so die Anzahl der Partdateien auf ein vernünftiges Maß begrenzt, weil sie nur linear mit der Anzahl der erzeugten Artefakte zusammenhängt und nicht nur durch das Produkt der Anzahl der Eingabemodelle mit den erzeugten Artefakten begrenzt ist.

9.3.6 Logging

Das DSLTool-Framework stellt ein auf der Java-Logging-API basierendes Logging zur Verfügung. Logging dient dabei zur Erstellung von Protokollen über die Ausführung der Software und deren interne Prozesse. Das Logging-Verhalten kann zur Laufzeit aktiviert oder deaktiviert werden. Dieses ist insbesondere beim Betrieb von Generatoren über OSTP sinnvoll, weil hier eine nachträgliche Auswertung des Protokolls hilft, die Gründe für die Fehlerbeschreibungen der Nutzer zu identifizieren. Über den Aufruf `mc.MCG.getLogger()` kann ein Java-Logger erzeugt und für die Logging-Meldungen verwendet werden. Das DSLTool-Framework erlaubt über die Basisoperationen hinaus, am Ende der Logmeldung in Klammern eine Kategorie anzugeben. Dadurch können innerhalb der Konfigurationsdatei `logging.properties` spezielle Befehle als Filter für Log-Meldungen verwendet werden. Mit `java -Dmc.loggingfile=myfile` kann eine andere Konfigurationsdatei als die in der Laufzeitumgebung `MontiCoreRE` ausgelieferte Version benutzt werden.

Die Abbildung 9.19 zeigt einen Ausschnitt aus der Konfigurationsdatei, in dem in Zeile 2 festgelegt wird, dass die Loggingmeldungen auf die Konsole ausgegeben werden. Dabei werden Meldungen aller Stufen ausgegeben (Zeile 5), wobei durch den Einsatz der Filter nur Logs der Kategorien `part` und `grammar` angezeigt werden (vgl. Zeile 9). Die Formatierung geschieht dabei durch die Klasse `mc.logging.EvenSimplerFormatter` (Zeile 12).

logging.properties

```
1  # Handler handle log requests
2  handlers= java.util.logging.ConsoleHandler
3
4  # ERROR, WARNING, INFO, CONFIG, FINE, FINER, FINEST (=all logging messages)
5  .level= FINEST
6
7  # Restrict logging to a certain category
8  # here only part and grammar logs are printed
9  mc.logging.restrictions = part, grammar
10
11 # Which formatter to use
12 java.util.logging.ConsoleHandler.formatter = mc.logging.EvenSimplerFormatter
```

Abbildung 9.19: Ausschnitt aus Konfigurationsdatei logging.properties

9.3.7 Modellmanagement

Die Verwendung von Modellen als primäre Eingabe innerhalb eines textuellen modellgetriebenen Ansatzes macht es notwendig, Verweise zwischen einzelnen Modellen zu etablieren. Dadurch kann nicht nur aus einzelnen Modellen unabhängig voneinander Code erzeugt, sondern auch deren Interaktionen geeignet beschrieben werden. Für die Verlinkung gibt es verschiedene Realisierungsmöglichkeiten, wobei innerhalb von MontiCore ähnlich zu [WK06] die Modelle durch Namen verbunden sind. Das verwendete Verfahren orientiert sich an den Programmiersprache Java, wodurch jedes Artefakt nur aus einem einzelnen Modell besteht und somit die Dateistruktur und die Struktur der Modelle unifiziert wird. Jedes Modell erhält durch die Paketbezeichnung und den Namen des Modells einen eindeutigen vollqualifizierten Namen, wobei die Art des Modells keine Rolle spielt. Dieser Name dient zur eindeutigen Identifikation eines Modelle innerhalb der auf dem DSLTool-Framework basierten Werkzeuge. Die Verweise auf andere Modelle sind dabei entweder durch die Verwendung von vollqualifizierten Namen oder durch einfach qualifizierte Namen realisiert, die aufgrund von Imports aufgelöst werden.

Der Vorteil ist dabei, dass die Verbindungen direkt innerhalb der Modelle beschrieben sind und keine zusätzliche Werkzeugunterstützung notwendig ist. Die Modelle sind daher in sich abgeschlossen und unabhängig von weiteren Konfigurationsdateien, was eine agile Modellierung vereinfacht. Der Nachteil ist, dass die Abhängigkeiten nun in den Modellen vorhanden sind und daher eventuell bei einer Wiederverwendung entfernt werden müssen. Zur Unterstützung dieses Vorgehens ist innerhalb der MontiCore-Grammatik die Option `compilationunit` verfügbar. Diese fügt zusätzliche Produktionen in die Grammatik ein und erreicht einen einheitlichen Dateikopf, wie in Abbildung 9.20 vereinfacht dargestellt wird, ohne dass ein Sprachentwickler diese Produktionen immer wieder schreiben muss. Dieser einheitliche Kopf ermöglicht dem DSLTool-Framework Modelle auf eine einheitliche Art zu referenzieren. Die Logik der Namensauflösung orientiert sich an Java [GJS05, §6.3.1] und ist in der abstrakten Klasse `mc.Modelloader` implementiert. Dabei werden kei-

ne Sichtbarkeiten und statischen Imports behandelt. Beim Auflösen eines Namens aufgrund der Menge von Import-Befehlen wird folgendes Vorgehen verwendet:

1. Zunächst werden die Modelle aufgrund der vollqualifizierten Imports wie zum Beispiel `mc.examples.Java` geladen (In [GJS05] als *Single-Type-Imports* bezeichnet).
2. Dann werden die Modelle innerhalb desselben Pakets verwendet.
3. Schließlich werden alle *On-Demand-Imports* ausgewertet und dort nach dem Modell gesucht (zum Beispiel `mc.examples.*`).

Bei der Namensauflösung dürfen keine zwei Imports derselben Verarbeitungsstufe denselben nicht-qualifizierten Namen verwenden, was durch eine Fehlermeldung quittiert wird. Tritt dieser Fehler beim Laden eines Modells auf, weil zum Beispiel zwei Modelle mit demselben Namen aber unterschiedlichen Modellarten verwendet werden, dann wird eine `mc.dsltool.AmbiguousImportException` ausgelöst.

Neben der Klasse `mc.MockModelloader`, die zum Testen genutzt wird, wird innerhalb des DSLTool-Frameworks vor allem die Klasse `mc.FileModelloader` verwendet, um weitere Instanzen aus dem Dateisystem oder einzelnen Jar-Archiven zu laden. Diese ist zum Beispiel notwendig, wenn die DSL einen Vererbungsmechanismus bietet und somit für eine Analyse die Oberinstanz benötigt wird.

9.3.8 Plattformunabhängigkeit

Generative Werkzeuge in der Softwareentwicklung sollen möglichst an keine Plattform fest gebunden sein: Einerseits sollen sie gut in Ant-Skripte und Makefiles einbindbar sein und auf Kommandozeile ausführbar sein, um sie in automatischen Buildprozessen und nächtlichen oder kontinuierlichen Integrationssystemen einbinden zu können. Andererseits ist aber auch eine Eclipse-Einbindung sinnvoll, damit eine komfortable Benutzung durch

Abbildung 9.20: Auswirkungen der Option compilationunit auf die Sprachdefinition

die Entwickler und eine integrative Entwicklung von DSLs mit Java-Quellcode in einer Entwicklungsumgebung möglich ist.

Daher ist das DSLTool-Framework so konstruiert, dass Generatoren ohne weiteres direkt von der Kommandozeile ausgeführt werden können und so eine Integration in Build-Skripte und Serverumgebungen ohne installiertes Eclipse unkompliziert möglich ist. Zusätzlich gibt es eine Unterstützung für die Entwickler der DSL, die diese Generatoren in Eclipse integrieren können.

Die Komponente `MontiCoreTE_RE` erlaubt zusammen mit der Generierung von Texteditoren eine unkomplizierte Integration der Generatoren in Eclipse auf Grundlage eines Konzepts (vgl. Abschnitt 4.3). Die Abbildung 9.21 zeigt schematisch die wichtigsten beteiligten Klassen. Dabei wird für das Beispielprojekt `mydsl` eine Subklasse von DSLTool

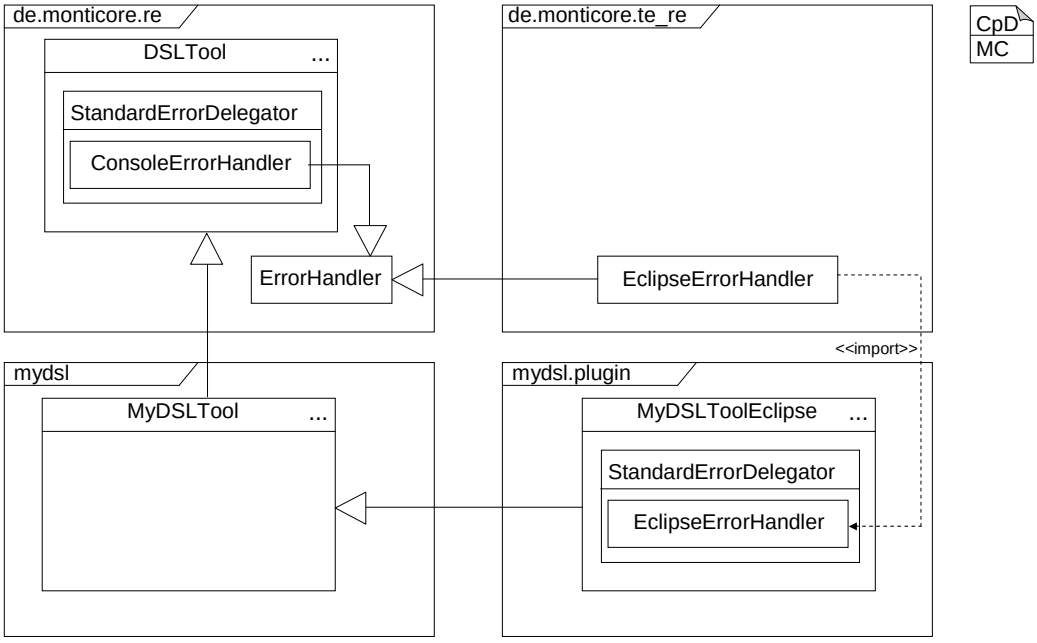


Abbildung 9.21: Plattformunabhängigkeit eines DSLTools und Eclipse-Integration

gebildet, die die gesamte Generatorlogik in verschiedenen ExecutionUnits anbietet. Dieses Werkzeug kann dann innerhalb von Buildskripten und kontinuierlichen Integrationsumgebungen genutzt werden. Die beiden Komponenten `de.monticore.re` und `mydsl` sind daher unabhängig von Eclipse oder anderen Laufzeitbibliotheken.

Dieses Werkzeug wird dann innerhalb des Eclipse Plugins `mydsl.plugin` mit einem zusätzlichen Errorhandler versehen. Dadurch werden die Status- und Fehlermeldungen direkt innerhalb der vorgesehenen Views in Eclipse angezeigt. Hinzu kommt beim Einsatz der Editorgenerierung auch ein weiterer Workflow zum Aufbau der Outline. Die Basisimplementierungen zum generierten Anteil kommen dabei aus der Laufzeitbibliothek `de.monticore.te_re`, die ihrerseits die notwendigen Eclipsebibliotheken verwendet (nicht dargestellt).

9.3.9 Template-Engine

Template-Engines verwenden als Templates ein spezielles Dateiformat, das sich im Wesentlichen am Format der zu erzielenden Ausgabe orientiert. Zusätzlich werden an verschiedenen Stellen Template-Informationen verwendet, die durch ein bereitgestelltes Datenmodell

mit Inhalt gefüllt werden. Das DSLTool-Framework erlaubt die Integration verschiedener Template-Engines für die es vorbereitete Adapter gibt. Im Folgenden wird die mit dem DSLTool-Framework ausgelieferte Template-Engine beschrieben, wobei die Darstellungen zum Teil auf [KR06b] basieren.

Bei der Verwendung von Codegeneratoren in agilen Projekten tritt das folgende Problem auf: Handcodierter Code wird oft durch Refactoring [Opd92, Fow99] und meistens sogar durch darauf basierende automatisierte Werkzeugen verändert. Dabei wird, um allen Quellcode konsistent zu halten, nicht nur der handcodierte Quellcode verändert, sondern als Folge auch der generierte Code, sofern dieser als Quellcode und nicht nur unveränderbarer Objektcode verfügbar ist. Daraus ergibt sich, dass das Template und der generierte Code nicht mehr konsistent sind. Diese Inkonsistenz kann auf zwei verschiedene Arten aufgelöst werden: Erstens wird die Codegenerierung als eine einmalige nicht-wiederholbare Operation angesehen. Diese macht jedoch die spätere Änderung der Modelle unmöglich und verhindert somit eine agile evolutionäre Entwicklung des Softwaresystems. Zweitens werden die Änderungen manuell am Template ausgeführt, was erfordert, dass geprüft wird, ob der Code und das Template wiederum konsistent sind. Dabei wird nochmals generiert und manuell geprüft, ob das Resultat konsistent ist, was dieses Vorgehen insgesamt kompliziert macht.

Dieser Ansatz ist jedoch zeitaufwändig und verhindert daher ein agiles Refactoring von generativer Software. Das Problem ist in Abbildung 9.22 nochmals genau illustriert, wobei die folgenden Schritte an einem Softwaresystem angewendet werden, das eine automatische Quellcodeerzeugung einsetzt:

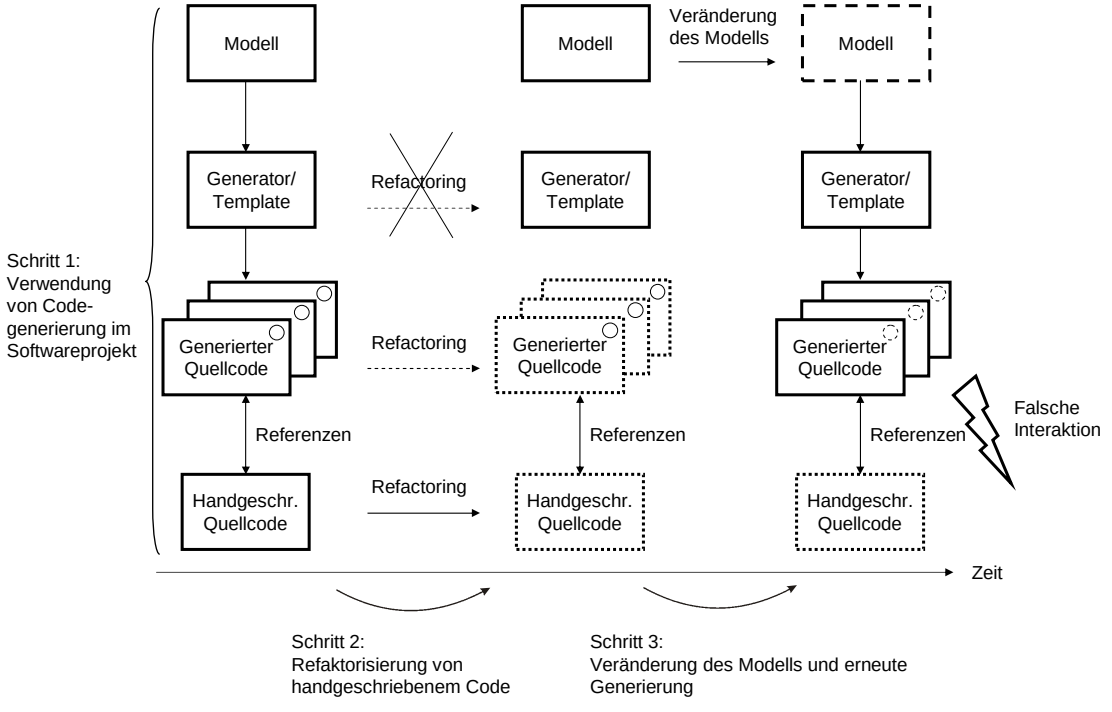


Abbildung 9.22: Inkonsistenz des Templates mit dem generierten Code durch Refactoring beim herkömmlichen Verfahren

1. Der Generator verarbeitet Modelle, in Folgenden auch Daten genannt, und nutzt eine Templatedatei, um den Quellcode zu erzeugen. Das Resultat dieses Prozesses sind

generierte Quellcodedateien, die typischerweise mit handgeschriebenem Quellcode interagieren, der technische Details, spezifische Algorithmen und wieder verwendbare Elemente eines Frameworks enthält.

2. Der handgeschriebene Quellcode wird refaktorisiert. Wenn der generierte Code Referenzen wie Methodenaufrufe oder Variableninstanzierungen enthält, die im handgeschriebenen Quellcode definiert wurden, werden Änderungen automatisch durch die Refactoring-Engine durchgeführt. Dieser Prozess wird als *updating references* [Opd92] für Quellcode bezeichnet, der nicht direkt refaktorisiert wird.

3. Das Modell wird manuell verändert, wodurch eine wiederholte Generierung notwendig wird. Die Änderungen der Refactoring-Engine aus Schritt 2 werden dadurch verworfen und der resultierende Quellcode interagiert nicht mehr notwendigerweise korrekt mit dem generierten Code.

Das Problem wäre in dieser Form nicht aufgetreten, wenn das Template auf dieselbe Art und Weise wie die generierten Klassen in Schritt 2 modifiziert worden wäre oder keine wiederholte Generierung möglich wäre. Die MontiCore-Template-Engine erlaubt das Schreiben von Templates in einer Form, die das automatische Refaktorisieren von einer beliebigen Refactoring-Engine erlaubt, die Referenzen aktualisiert und Kommentare erhält. Experimente zeigen, dass die Eclipse-Refactoring-Engine diese Bedingungen erfüllt. Die Abbildung 9.23 zeigt diesen Ansatz sowie die erfolgreiche Interaktion nach einem Refactoring und einer Neugenerierung der Klassen.

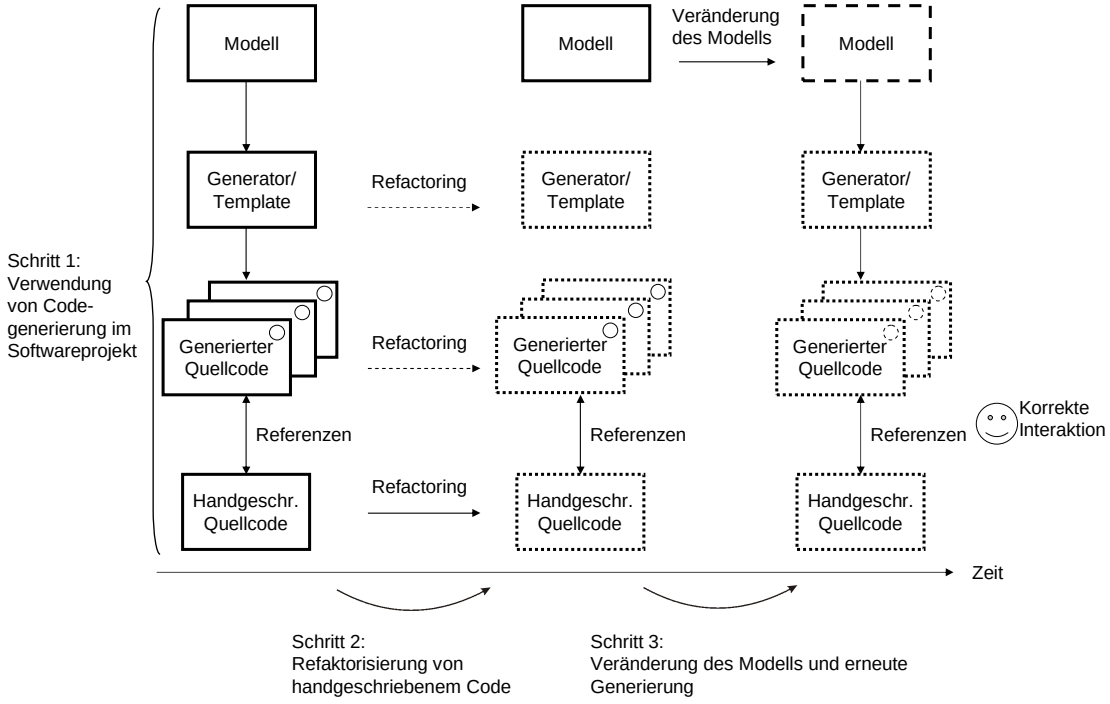


Abbildung 9.23: Koevolution des Templates mit dem generierten Code durch Refactoring

Ein Template ist eine Kombination von Teilen der Zielsprache mit Teilen, die Direktiven für die Template-Engine enthalten. Eine solche Datei ist typischerweise nicht kompilierbar, weil die Direktiven keine validen Teile der Zielsprache sind. Von einem praktischen Standpunkt aus ignoriert daher eine Refactoring-Engine der Zielsprache die Templates und ist

daher nicht in der Lage, dort Referenzen zu aktualisieren. Eine schwergewichtige Lösung wäre, die Refactoring-Engine derart zu verändern, dass sie auch diese Templates verstehen und verändern kann. Dieser Weg ist möglich, erfordert jedoch einen nicht zu unterschätzenden Aufwand. Es wurde daher einen leichtgewichtiger Weg gewählt, bei dem die Templates compilierbarer Sourcecode sind und daher existierende unmodifizierte Template-Engine die Templates erkennen und Referenzen aktualisieren können.

Der Ansatz wird *spracherhaltend* genannt, weil vom Standpunkt der Zielsprache aus das Template bereits syntaktisch korrekt ist, auch wenn es nicht direkt innerhalb der Software verwendet wird. Dem Autor ist kein anderer textueller Ansatz bekannt, der diese Idee für eine Template-Engine verwendet. Allerdings sind *model templates*, die in der Zielsprache geschrieben und mit Annotationen versehen sind, eine ähnlich Idee [Cza05].

Spracherhaltende Template-Engine

Die Hauptidee der im Folgenden dargestellten Template-Engine ist nicht Codegenerierung durch Ersetzen von Löchern durch Daten, sondern die Ersetzung von markierten Beispieldaten durch Produktivdaten. Der Hauptanteil des Zielcodes entsteht durch Kopieren aus dem Template. Eingebettet sind spezielle Kommentare der Zielsprache, so genannte Tags, die als Anweisungen an die Template-Engine interpretiert werden und so den Generierungsprozess steuern. Die Template-Engine nutzt die Attribute der Modellklassen zur Beschaffung der Daten und schreibt sie in die Ausgabe. Insgesamt werden die Kommentare und die Beispieldaten, welche typischerweise ein einzelnes Wort sind, in den Zielcode überführt.

Ein Beispiel für dieses Verhalten wird in Abbildung 9.24 dargestellt. Ein Tag in diesem Beispiel sieht wie ein spezieller mehrzeiliger Java-Kommentar aus (`/*C ... */`). Wie bereits beschrieben wird das darauf folgende Wort durch die wirklichen Daten ersetzt. Welche Daten genau ausgewählt werden, hängt vom Inhalt des Kommentars ab. Die Template-Engine erlaubt hier zwei Möglichkeiten, die alle direkt hintereinander ohne Leerzeichen kombiniert werden, um das Wort nach dem Kommentar zu ersetzen. Erstens können Wörter durch `%` eingeschlossen werden, was bedeutet, dass hier der Wert des gleichnamigen Attributs innerhalb des Modells verwendet wird. Zweitens werden alle anderen Wörter als einfacher Text interpretiert, der direkt in die Ausgabe kopiert wird. Im Beispiel wird dies verwendet, um valide Zeichenketten in der Ausgabe zu bilden.

In der MontiCore-Template-Engine hat stets eine Instanz der Modellklassen den Fokus. Das bedeutet, dass sich alle Anweisungen, die durch `%` gekennzeichnet sind, auf dieses Objekt beziehen. Dieses aktive Objekt kann durch die folgenden üblichen Kontrollstrukturen geändert werden:

`/*for %X% */ ... /*end*/` Die Voraussetzung für dieses Tag ist, dass das aktive Objekt eine Methode `getX()` bereitstellt, die von der Template-Engine aufgerufen wird. Der Rückgabewert wird bis zum schließenden Kommentar (`/*end*/`) das aktive Objekt. Danach wird das aktuelle Objekt wieder aktiv.

`/*foreach %X% */ ... /*end*/` Die Vorausetzung für dieses Tag ist, dass das aktive Objekt eine Methode `getX()` bereitstellt, die von der Template-Engine aufgerufen wird. Der Rückgabewert muss vom Typ `java.util.List` sein. Der erste Eintrag wird das aktive Objekt, welches für die Codeerzeugung genutzt wird, bis das Endzeichen (`/*end*/`) erreicht wird. Dieses Verhalten wird für alle Listenelemente wiederholt.

`/*if %X% */ ... /*else*/ ... /*end*/` Die Vorrausetzung für dieses Tag ist, dass das aktive Objekt eine Methode `isX()` bietet, die von der Template-Engine aufgerufen

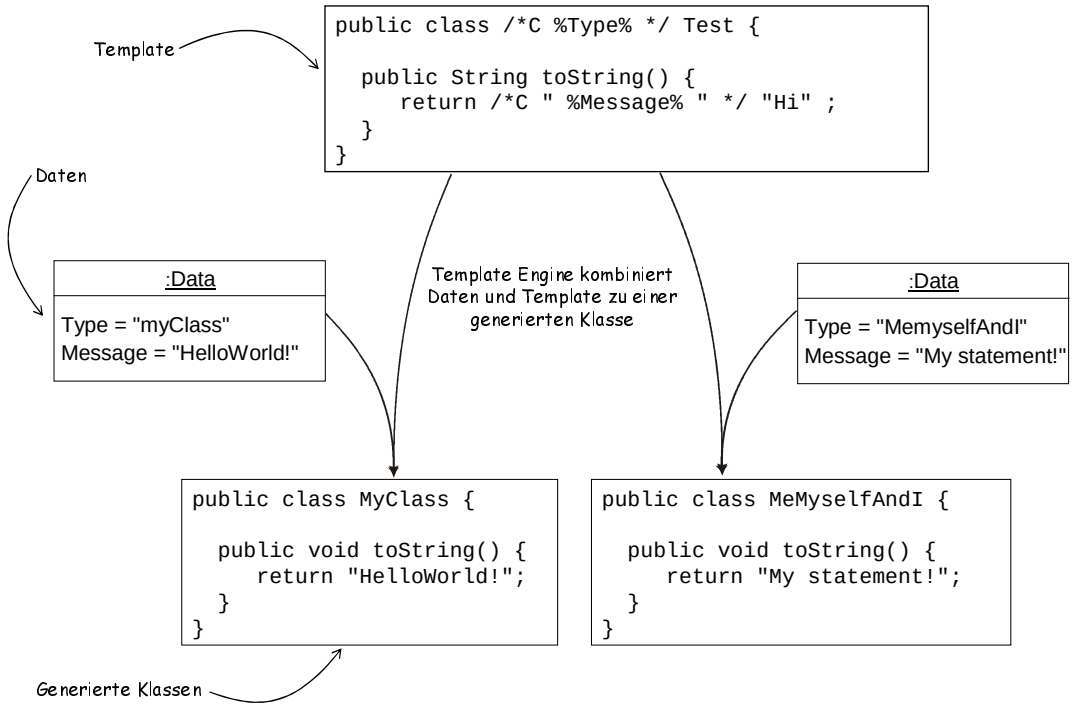


Abbildung 9.24: Anwendung der Template-Engine

wird und einen boolschen Wert liefern muss. Wird wahr zurückgegeben, wird der erste Teil des Codes zur Generierung genutzt, andernfalls der zweite Teil. Dieses Tag ändert nicht das aktive Objekt.

Die erwähnten Kontrollstrukturen können beliebig geschachtelt werden, um auf den Modellbaum zuzugreifen. Innerhalb der `for` oder `foreach` Umgebungen kann auf Knoten weiter oben in der Hierarchie durch `%number%name%` zugegriffen werden, wobei `number` eine natürliche Zahl ist und die Anzahl der Schritte nach oben bezeichnet (`1` für den direkten Vaterknoten) und `name` der Name des betrachteten Attributs ist.

In speziellen Situationen ist es einfacher, direkt die Ausgabedaten zu beschreiben, ohne ein zu ersetzendes Beispiel anzugeben. Dies entspricht dem üblichen Verhalten einer Template-Engine und kann durch `/*O ... */` erreicht werden.

Beispiel

Um die Fähigkeiten dieses Ansatzes zu demonstrieren, wird das Beispiel von Martin Fowler aus [Fow05] verwendet, das er nutzt, um auf verschiedene Weise Code zu erzeugen. Im Folgenden wird sich auf auf seinen Ansatz zur Nutzung von Velocity konzentriert. Das Beispiel ist aus dem Artikel, wurde jedoch von C# nach Java übersetzt und leicht angepasst, um die entscheidenden Punkte einfach und klar darzustellen. Sowohl Template als auch der entstehende Java-Code sind komplette Klassendateien, obwohl die Abbildungen aus Übersichtsgründen nur das Innere der Klasse zeigen.

Die Kernidee des Beispiels ist, einen so genannten Reader durch Objekte vom Typ `ReaderStrategy` zu parametrisieren. Die Reader werden dann genutzt um Dateien in einem zeilen-orientierten Dateiformat zu parsen, wobei das Schlüsselwort am Anfang der Zeile die Struktur der Daten bestimmt. In Abhängigkeit vom Schlüsselwort haben die Zei-

chen zwischen Start- und Endposition eine Bedeutung und sollen extrahiert werden. Eine ausführlichere Erklärung findet sich in [Fow05]. Das Beispiel in Abbildung 9.25 zeigt, dass es möglich ist, APIs zu programmieren, damit der Code in zwei Teile aufgeteilt werden kann: Einen, der die Basisfunktionen enthält, und einen, der diese Basisfunktionen in einer für das Problem spezifischen Weise nutzt.

Java-Quellcode

```
1 public void Configure(Reader target) {
2   target.AddStrategy(ConfigureServiceCall());
3   target.AddStrategy(ConfigureUsage());
4 }
5
6 private ReaderStrategy ConfigureServiceCall() {
7   ReaderStrategy result = new ReaderStrategy("SVCL", mc.examples.SCall.class);
8   result.AddFieldExtractor(4, 18, "CustomerName");
9   result.AddFieldExtractor(19, 23, "CustomerID");
10  return result;
11 }
12
13 private ReaderStrategy ConfigureUsage() {
14  ReaderStrategy result = new ReaderStrategy("USGE", mc.examples.Usage.class);
15  result.AddFieldExtractor(4, 8, "CustomerID");
16  result.AddFieldExtractor(9, 22, "CustomerName");
17  return result;
18 }
```

Abbildung 9.25: Generierter Quellcode (adaptiert aus [Fow05])

Der Quellcode aus Abbildung 9.25 kann so direkt verwendet werden, aber für komplexere Dateien wäre es wünschenswert, nur die notwendigen Informationen aufschreiben zu müssen und die technischen Details einer Codegenerierung zu überlassen. Die Abbildung 9.26 enthält nur die kondensierten Informationen aus dem Java-Code. Die Abbildungen 9.27 und 9.28 stellen Codegenerierungen für diese DSL mit Velocity und der MontiCore Template-Engine gegenüber.

DSL-Quellcode

```
1 mapping SVCL ServiceCall
2    4-18: CustomerName
3   19-23: CustomerID
4 mapping USGE Usage
5    4- 8: CustomerID
6    9-22: CustomerName
```

Abbildung 9.26: Domänenspezifische Beschreibung aus Abbildung 9.25

Die Erfahrung hat gezeigt, dass die Verwendung der MontiCore-Template-Engine den Vorteil hat, dass syntaktische Fehler schneller und leichter entdeckt werden, weil die Templates normale Java-Klassen sind, diese in modernen IDEs stets im Hintergrund kompiliert werden und so solche Fehler direkt gemeldet werden. Bei anderen Template-Engines ist hierfür erst eine Übersetzung mit Beispieldaten notwendig. Zusätzlich stehen bei der Entwicklung Hilfsfunktionen moderner IDEs zur Verfügung wie die Erzeugung von get/set-Methoden, Umbenennung von Variablen, Code-Assistenten wie Autovervollständigung und

Velocity Template

```
1 public void Configure(Reader target) {
2   #foreach( $map in ${config.Mappings})
3   target.AddStrategy(Configure${map.ClassName}());
4   #end
5 }
6
7 #foreach( $map in ${config.Mappings})
8   private ReaderStrategy Configure${map.ClassName}() {
9   ReaderStrategy result =
10    new ReaderStrategy("$map.Name", $map.ClassQName.class);
11
12    #foreach( $f in $map.Fields)
13    result.AddFieldExtractor($f.Start, $f.End, "$f.FieldName");
14    #end
15
16    return result;
17 } #end
```

Abbildung 9.27: Template mit Velocity (adaptiert aus [Fow05])

MontiCore Template

```
1 public void Configure(Reader target) {
2   /*foreach %Mappings% */
3   target.AddStrategy( /*C Configure %ClassName% () */ ConfigureSCall() );
4   /*end*/
5 }
6
7 /*foreach %Mappings% */
8 private ReaderStrategy /*C Configure %ClassName% () */ ConfigureSCall() {
9   ReaderStrategy result = new ReaderStrategy
10    ( /*C " %Name% " */ "SVCL" , /*C %ClassQName% */ mc.example.SCall .class);
11
12  /*foreach %Fields% */
13  result.AddFieldExtractor( /*C %Start% */ 4 ,
14    /*C %End% */ 18 , /*C " %FieldName% " */ "CustomerName" );
15  /*end*/
16
17  return result;
18 } /*end*/
```

Abbildung 9.28: Template mit MontiCore Template-Engine

die Implementierung von Methoden aus Oberklassen. Diese Features sind jedoch alle (willkommene) Nebeneffekte der verbesserten Möglichkeit, die Templates mit dem Quellcode zu refaktorisieren.

In den Experimenten werden eine Menge von Refactorings angewendet, inklusive der folgenden, um mittels der Eclipse-Refactoring-Engine den handgeschriebenen Quellcode, der mit dem generierten interagiert, zu refaktorisieren. Alle Refactorings führten automatisch zu korrekten Änderungen des Quellcodes im Beispiel. Die Liste ist nicht komplett, gibt aber einen Überblick über Refactorings, die sowohl den generierten als auch den handcodierten Code verändern.

- Umbenennen
  Zum Beispiel `Reader` nach `Parser` oder `ReaderStrategy` nach `ParserStrategy`.
- Ändern einer Methodensignatur
  Zum Beispiel Hinzufügen eines weiteren Parameters mit einem default-Wert zur Methode `ReaderStrategy.addFieldExtractor` oder das Löschen eines Parameters derselben Methode.
- Schnittstelle extrahieren
  Zum Beispiel Extrahieren einer Schnittstelle `Strategy` aus `ReaderStrategy` und deren Verwendung anstelle von `ReaderStrategy` im generierten Code.
- Bewegen
  Zum Beispiel Bewegen einer Klasse `ReaderStrategy` in ein anderes Paket. Die notwendigen Imports wurden im Template hinzugefügt.

9.3.10 Traversierung der Datenstrukturen

Die Traversierung von Objektstrukturen ist ein häufig benötigtes Hilfsmittel zur Implementierung von Analysen und Codegenerierungen. MontiCore erlaubt generell die Traversierung der Modelle entlang der aufspannenden Kompositionsstruktur und orientiert sich dabei am Visitor-Muster [GHJV95]. Die Realisierung der Traversierung basiert teilweise auf der Java-Reflection, weil so eine dynamische Erweiterbarkeit der Traversierung über Spracheinbettungsgrenzen möglich ist. Die zentrale Klasse bei der Traversierung ist die Klasse `mc.Visitor`, die mit verschiedenen `mc.ConcreteVisitor`-Instanzen parametrisiert wird. Methodisch passt dieses zur Einbettung von Sprachen, da so die Algorithmen für die einzelnen Fragmente einzeln implementiert und erst zur Konfigurationszeit variabel miteinander kombiniert werden. Die Sprachvererbung wird durch ein Ableiten der jeweiligen Visitorimplementierung unterstützt.

Die Abbildung 9.29 zeigt die Kombination zweier Teil-Visitoren und die Traversierung einer Objektstruktur. Dabei werden Automaten mit einer Aktionssprache als Beispiel verwendet. In den Zeilen 12 und 13 werden zwei Teil-Visitoren `CountEventsAutomaton` und `CountEventsAction` dem Visitor hinzugefügt, die auf einer gemeinsamen Datenstruktur `Counter` operieren. Die einzelnen ConcreteVisitors können unabhängig voneinander entwickelt werden und kooperieren nur über die gemeinsame Datenstruktur. Der Baum wird dann in Preorder-Reihenfolge (auch Depth-First genannt) durchlaufen, das heißt ein Knoten wird vor seinen Unterbäumen besucht. Die Unterbäume werden sequentiell verarbeitet, wobei ein Unterbaum zunächst komplett besucht wird, bevor der nächste Unterbaum betrachtet wird. Für die jeweiligen Knoten wird die passende Methode aufgerufen, wobei die Klassen aufgrund ihres vollqualifizierten Namens unterschieden werden können. In diesem Fall wird das Auftreten eines Events gezählt, das sowohl in der Automatensprache als auch in der Aktionssprache ausgelöst werden kann.

Die Traversierung erfolgt prinzipiell in einem Preorder-Durchlauf des aufspannenden Baums, wobei der Entwickler die folgenden drei Methoden mit dem einzelnen Parameter der verwendeten Modellklasse verwenden kann. Eine beliebige Kombination für verschiedene Modellklassen ist dabei möglich.

visit(...): Diese Methode wird aufgerufen, bevor die Kindknoten besucht werden.

endVisit(...): Diese Methode wird aufgerufen, nachdem die Kindknoten besucht wurden.

Java-Quellcode

```
1  public class Test extends DSLWorkflow<AutomatonRoot> {
2
3    // ...
4
5    @Override public void run(AutomatonRoot dslroot) {
6
7      ASTAutomaton ast = dslroot.getAst();
8
9      Counter x = new Counter();
10
11     Visitor v = new Visitor();
12     v.addClient(new CountEventsAutomaton(x));
13     v.addClient(new CountEventsAction(x));
14
15     // ...
16   }
17 }
18
19 public class CountEventsAutomaton extends ConcreteVisitor{
20
21   private Counter x;
22
23   public CountEventsAutomaton(Counter x) {
24     this.x = x;
25   }
26
27   public void visit(mc.examples.automaton.ASTEvent e){
28     x.increment();
29   }
30
31 }
32
33 public class CountEventsAction extends ConcreteVisitor{
34
35   private Counter x;
36
37   public CountEventsAction(Counter x) {
38     this.x = x;
39   }
40
41   public void visit(mc.examples.action.ASTEvent e){
42     x.increment();
43   }
44
45 }
```

Abbildung 9.29: Beispiel für die Verwendung des Visitor-Musters im MontiCore-Framework

ownVisit(...): Diese Methode wird aufgerufen, bevor die Kindknoten besucht werden. Sie verhindert eine Traversierung der Kinder, wodurch eine eigene Traversierungsstrategie implementiert werden kann.

Zusätzlich kann innerhalb der visit-Methoden die Traversierung der Unterbäume über den Aufruf `stopTraverseChildren()` verhindert werden, so dass sie sich dann wie eine ownVisit-Methode verhalten. Alternativ kann die gesamte folgende Traversierung durch den Aufruf `stopTraverse()` beendet werden. Alternativ ist es möglich, gezielt aus einer

	Traversierung			Transformation		Codegenerierung	
Anzahl Knoten	GoF	MCx	MC	GoF	MC	GoF	MC
1000	0,36	1,15	1,50	1,13	2,20	1,97	3,24
10000	6,59	14,81	18,33	18,71	28,55	32,30	45,22
25000	14,81	18,33	19,45	52,14	76,78	93,94	125,28
50000	41,10	80,30	98,04	134,45	182,50	355,65	481,06

Tabelle 9.1: Laufzeit einer einzelnen Traversierung eines AST in ms

`ownVisit(...)`-Methode heraus gezielt nur einige Unterbäume zu traversieren. Die Verwendung einer `ownVisit(...)`-Methode für eine Modellklasse schließt die Nutzung der Methoden `visit(...)`/`endVisit(...)` für diese Modellklasse aus.

Im Gegensatz zu anderen Ansätzen [PJ98, BRLG04] wird jedoch weiterhin wie im Grundmuster [GHJV95] eine `traverse()`-Methode innerhalb der Modellklassen genutzt. Dadurch ist die Verwendung invasiv, was jedoch aufgrund der automatischen Generierung der AST-Klassen kein Problem darstellt, es können allerdings die Performance-Einbußen (18- bis 256-mal langsamer! [PJ98, BRLG04]) der nicht-invasiven Methoden, wie im Folgenden gezeigt wird, deutlich reduziert werden.

Im Folgenden werden die Ergebnisse eines Performance-Tests dargestellt, bei dem drei verschiedene Algorithmen mit unterschiedlicher Komplexität jeweils einmal mit einer in [GHJV95] vorgeschlagenen Form der Implementierung (GoF) und einer auf MontiCore-Visitoren basierenden Implementierung (MC) erstellt wurden.

Nur Traversierung: Hierbei wurde ausschließlich der Baum traversiert. In MontiCore gibt es dabei zwei Möglichkeiten: Erstens, äquivalent zur Standard-Implementierung, die Verwendung leerer Methodenrümpfe (MC) und zweitens das Auslassen der Implementierung (MCx).

Transformation: Ein einfacher Algorithmus wird auf den Bäumen ausgeführt, der die Attribute des AST modifiziert.

Codegenerierung: Es werden typische Operationen für eine Codegenerierung und Erzeugung eines Ausgabestroms verwendet. Hier war zunächst ein kaum messbarer Unterschied zwischen den Varianten festzustellen, wenn Schreibaktionen auf Dateien Teil des Algorithmus sind, wie es in Codegenerierungen typischerweise der Fall ist. Die Messungen zeigten jedoch eine sehr große Streuung von etwa dem Faktor 10 von der schnellsten zur langsamsten Messung der Schreiboperationen auf Festplatte, so dass eine solche Messung aufgrund der großen Varianz kaum eine Aussagekraft über die Geschwindigkeitsunterschiede der Visitoren erlaubt, sondern nur die Schreibzugriffe beurteilt. Somit wurde die Codegenerierung auf Erzeugung von Quellcode im Arbeitsspeicher beschränkt.

Zur Bestimmung der Laufzeit wurde die Systemuhr verwendet. Um die Ungenauigkeiten durch diese Messmethode zu reduzieren, wurden die Algorithmen intern wiederholt aufgerufen, so dass sich die Gesamtdauer der Messung stets zwischen 300ms und 2s bewegt. Die Messung erfolgte auf einem Laptop mit Intel Core CPU T7200 mit 2GHz Taktrate, Windows XP mit installiertem JDK 1.6.0_03 und 512MB zugesicherte Speicher für die VM. Die Quelldateien wurden mit dem Java Eclipse Compiler 3.3 übersetzt. Die Tabelle 9.1 zeigt die Daten pro Durchlauf in Millisekunden.

Die Analyse der Ergebnisse zeigt, dass die Traversierung mit MC aufgrund der eingesetzten Reflection etwa um den Faktor 4 schlechter ist. Bei größeren Bäumen verbessert sich dieser Faktor, weil beide Implementierungen hier konstant mehr Zeit zur Verwaltung des Stacks benötigen, was für beide Varianten dieselbe Zeit in Anspruch nimmt. Bei der Messung mit realitätsnahen Algorithmen verringert sich der Anteil der Traversierung an der Gesamtlaufzeit beträchtlich, so dass die MC-Implementierung bei großen Bäumen nur noch doch einen zusätzlichen Aufwand von 30-40% bedeutet.

Der wesentliche Vorteil der MC-Variante besteht in der Flexibilität während der Implementierung. Durch das Abschneiden von Unterbäumen, die nicht mehr traversiert werden sollen, kann eine erhebliche Verkleinerung des traversierten Baums und somit eine Geschwindigkeitssteigerung erreicht werden.

Basierend auf der Basis-Implementierung des Visitors wurden die folgenden Abwandlungen realisiert, die alternative Traversierungen der Bäume erlauben.

mc.ast.CommonVisitor: Diese Form der Traversierung verzichtet auf die Typunterscheidung der verschiedenen Modellklassen, wobei diese einheitlich durch eine `visit(...)` und eine `endVisit(...)`-Methode besucht werden. Dabei wird anstatt der Klasse `mc.ast.ConcreteVisitor` die Klasse `mc.ast.CommonConcreteVisitor` verwendet, um die Methoden zu implementieren.

mc.ast.MultiVisitor: Diese Form der Traversierung erlaubt es, beliebig viele Objekte vom Typ `mc.ast.ConcreteVisitor` hinzuzufügen, die insgesamt auch mehrere Methoden für dieselbe Modellklasse zur Verfügung stellen. Dadurch können innerhalb eines Durchlaufs verschiedene Algorithmen kombiniert werden, was jedoch die Verwendung von `ownVisit(...)`-Methoden ausschließt.

mc.ast.AdditonalVisitor: Zusätzlich zum Standardverhalten wird vor dem Aufruf von `visit(...)` die Methode `prenode(ASTNode)` und nach `endVisit(...)` die Methode `postnode(ASTNode)` verwendet. Dieser Visitor eignet sich insbesondere zur Erstellung von Unparsern, da die Kommentare so einheitlich behandelt werden können.

mc.ast.ProtocolVisitor: Dieser Visitor verwaltet intern eine zusätzliche Datenstruktur, die den Zugriff auf bereits traversierte Knoten aufgrund ihres Typs erlaubt. Zur Implementierung werden `mc.ast.ProtocolConcreteVisitor`-Klassen verwendet.

mc.ast.InheritanceVisitor: Dieser Spezialvisitor verwendet auch Methoden, die für Oberklassen definiert sind, sofern für die direkt verwendeten Klassen keine `visit(...)` `endVisit(...)` oder `ownVisit(...)` Methode implementiert ist.

mc.ast.ReverseVisitor: Diese Variante traversiert den Baum ausgehend vom aktuellen Knoten von unten nach oben und stoppt beim obersten Knoten.

Die Kombination verschiedener Visitoren ist insofern möglich, als in jeder Methode innerhalb einer Visitor-Variante andere Varianten gestartet werden können. Des Weiteren können Visitoren über die Methode `fail()` fehlschlagen, so dass eine Kombination von Visitoren zu Strategien wie in [Vis01, Vis03] beschrieben damit realisierbar ist. Ein Visitor implementiert ein Prädikat im Sinne der funktionalen API (vgl. Abschnitt 9.3.4) und kann mit anderen Funktionen ähnlich wie in [LJ03] kombiniert werden.

9.4 Zusammenfassung

Dieses Kapitel hat das DSLTool-Framework dargestellt, das die technische Grundlage für mit MontiCore erstellte generative und analytische Werkzeuge darstellt. Dazu wurde eine Referenzarchitektur entworfen, die durch Spezialisierung an die konkrete Problemstellung angepasst werden kann. Dabei wurde eine komponentenorientierte Implementierung verwendet, um jederzeit auch während der Laufzeit Systemteile austauschen zu können. Besonders hervorzuhebende Eigenschaften des Frameworks sind dabei:

- Ein Konzept zur inkrementellen Codeerzeugung, das es ermöglicht, Generatoren so zu schreiben, dass mehrere Eingabedateien zum Inhalt einer Ausgabedatei beitragen. Die Eingabedateien müssen nicht gleichzeitig verarbeitet werden, und bei Änderung einzelner Dateien müssen nur die Veränderungen betrachtet werden.
- Ein Modellmanagement, das durch eine einheitliche Benennung von Modellen ermöglicht, Generatoren für heterogene Modelle zu gestalten. Diese können transparent nachgeladen werden und Referenzen über Dateigrenzen hinweg zwischen verschiedenen Modellen realisiert werden.
- Die MontiCore-Template-Engine, die durch einen speziellen Mechanismus Templates verwendet, die kompilierbaren Java-Code darstellen und die so bei Refactoring-Schritten der Laufzeitumgebung automatisch angepasst werden.
- Eine flexible Traversierung der Modelle, die die Modularitätsmechanismen des MontiCore-Grammatikformats um eine dazu passende Traversierung ergänzt, so dass die Algorithmen analog zur Sprachdefinition komponiert und spezialisiert werden können. Durch die technische Realisierung der Traversierung konnten die Performance-Nachteile gegenüber der Standardimplementierung begrenzt und gleichzeitig die Flexibilität für den Entwickler erhöht werden.

Das DSLTool-Framework wird derzeit innerhalb von MontiCore zur Strukturierung des Generators eingesetzt und hat sich dort insbesondere bei der Integration verschiedener unabhängiger Generatoren zu einem homogenen Werkzeug bewährt. Seine Struktur ist flexibel genug, um durch Angabe von zusätzlichen Eigenschaften Abwandlungen von Werkzeugen zu erstellen. Das DSLTool-Framework bildet die Grundlage für die Fallstudien in Kapitel 10. Dort werden die einzelnen Eigenschaften des Frameworks nochmals anhand von Beispielen detaillierter dargestellt.

Kapitel 10

Fallstudien

Mit dem MontiCore-Framework wurden und werden derzeit DSLs vorwiegend innerhalb der eigenen Arbeitsgruppe definiert und verwendet. Das Framework wird dabei fortlaufend an die identifizierten Anforderungen angepasst und entsprechend erweitert.

Der Abschnitt 10.1 listet die wichtigsten mit MontiCore realisierten DSLs auf. Davon werden im Folgenden drei DSLs detaillierter betrachtet: Erstens das MontiCore-Framework, das im Bootstrapping-Verfahren mit sich selbst entwickelt wird. Zweitens eine DSL zur Funktionsnetzmodellierung, die zeigt, wie sich analytische Werkzeuge mit MontiCore realisieren lassen. Drittens die Transformationssprache JavaTF, die eine Java-Erweiterung darstellt.

10.1 Verwendung des MontiCore-Frameworks

Die folgende Liste zeigt die aktuell in Entwicklung befindlichen und abgeschlossen Anwendungsfälle des MontiCore-Frameworks:

- Die *AutosarDSL* [Höw07] erlaubt die Beschreibung einer Autosar-Software-Architektur mittels einer C-ähnlichen konkreten Syntax.
- *ARMS* [GHK+07, GKPR08, GHK+08b] bezeichnet eine Sprache zur Funktionsnetzmodellierung, die im Folgenden noch detaillierter erklärt wird.
- In [Wit07] wurde eine *C++*-Grammatik entwickelt, die zur Überprüfung von Codierungsrichtlinien eingesetzt werden kann.
- In [FM07] wurden eine *Java 5.0*-Grammatik und -Symboltabelle entwickelt, die die Integration von Java-Sprachfragmenten in Modelle ermöglichen.
- In [The06, Mah06, Pau09] wurde schrittweise die Transformationssprache *JavaTF* entwickelt, die noch detaillierter erklärt wird.
- *MontiCore* [GKR+06, KRV07b, GKR+08, KRV08] umfasst ein Sammlung von DSLs, die zur modularen Sprachdefinition verwendet werden können, und wird mit sich selbst im Bootstrapping-Verfahren entwickelt.
- *MontiWeb* bezeichnet eine DSL zur Modellierung von auf Webtechnologien basierender Systeme, die auf Erfahrungen aus [RRSS08] aufbaut.
- Die *UML/P* [Rum04b, Rum04a, GKRS06] bezeichnet ein Teilmenge der UML, die sich besonders zur agilen Modellierung von Softwaresystemen eignet.

10.2 MontiCore-Bootstrapping

Das MontiCore-Framework stellt eine Sammelung von DSLs zur Verfügung, mit denen DSLs und Generatoren basierend auf dem DSLTool-Framework entwickelt werden können. Gleichzeitig wird das MontiCore-Framework wie diese DSLs mit dem MontiCore-Framework selbst entwickelt. Dieser Prozess wird als Bootstrapping bezeichnet wodurch MontiCore auch eine Fallstudie für sich selbst darstellt und darum in diesem Kapitel kurz beschrieben wird. Auf eine Beschreibung der Funktionalität und die beinhalteten DSLs wird hier verzichtet, da dieses bereits in den Kapiteln 3, 4, 5 und 6 ausführlich getan wurde.

Der Abschnitt 10.2.1 gibt einen Überblick über einige Kennzahlen, die helfen die Größe des Projekts und der Teilmodule einzuschätzen. Der Abschnitt 10.2.2 beschreibt das Bootstrapping des MontiCore-Frameworks. Der Abschnitt 10.2.3 erklärt die Qualitätssicherung des Frameworks, wohingegen im Abschnitt 10.2.4 kurz organisatorische Aspekte aufgeführt werden. Der Abschnitt 10.2.5 fasst mögliche weitere Entwicklungen kurz zusammen.

10.2.1 Kennzahlen

Die Tabelle 10.1 gibt einen Überblick über einige Kennzahlen, die helfen die Größe des Projekts und der Teilmodule einzuschätzen. Dabei wurde der Softwarestand des 31. Release also der Version 1.1.5 verwendet. Die erste Spalte nach dem Komponentennamen gibt die Anzahl der Grammatiken innerhalb der Komponente an. Darauf folgen die *Source Lines of Code* (SLOC) der Grammatiken, also die Anzahl der nichtleeren Zeilen, die keine Kommentare sind, innerhalb einer Grammatik. Danach folgt die Anzahl der handcodierten Klassen und Schnittstellen, wohingegen die nächste Spalte die Anzahl der automatisch aus Grammatiken und Sprachdateien generierten Klassen und Schnittstellen beinhaltet. Die nächsten beiden Spalten vergleichen für diese beiden Kategorien die SLOC. Daraus wird in der letzten Spalte der Anteil der generierten SLOC am Gesamtprojekt ausgerechnet.

Die Daten zeigen, dass durchschnittlich 80% von MontiCore generiert sind. Der größte Anteil davon befindet sich in der Komponente `MontiCoreFE`, die das Frontend für die Nutzung enthält. Hier lässt sich die größte Automatisierung erreichen. Im Gegensatz dazu

Komponente	Anzahl Grammatiken	SLOC Grammatiken	Anzahl handcodierte Klassen	Anzahl generierte Klassen	SLOC handcodierte Klassen	SLOC generierte Klassen	Anteil generiert
CTS	0	0	141	0	9393	0	0,00%
JavaDSL	5	991	109	354	10292	70286	87,23%
MontiCore	3	169	107	76	12245	14509	54,23%
MontiCoreConsole	0	0	2	0	134	0	0,00%
MontiCoreFE	31	1559	172	915	15219	173885	91,95%
MontiCoreRE	8	352	277	117	17384	18525	51,59%
MontiCorePlugin	10	167	49	191	4493	27402	85,91%
MontiCoreTE_RE	0	0	48	0	1156	0	0,00%
Summe	57	3238	905	1653	70316	304607	81,25%

Tabelle 10.1: Kennzahlen der einzelnen MontiCore-Komponenten ohne OSTP-Integration

ist der Prozentsatz bei der Komponente `MontiCore` geringer, der die Codegenerierung enthält. Vergleicht man die Anzahl der generierten SLOC mit den SLOC der Grammatiken ergibt sich ungefähr ein Faktor von 94. Diese zeigt insbesondere, dass Parsererzeugung und AST-Klassengenerierung Fachdomänen sind, die von einer modellbasierten generativen Entwicklung profitieren können.

10.2.2 Bootstrapping

MontiCore wird in einem partiellen Bootstrapping-Verfahren entwickelt, was bedeutet, dass MontiCore zu seiner eigenen Entwicklung eingesetzt wird. Als partiell wird es hier bezeichnet, weil die in der ersten Version von Antlr erzeugten Anteile durch MontiCore-Grammatiken ersetzt werden, die in Java entwickelten Anteile jedoch prinzipiell erhalten bleiben [Wat00]. Die Abbildung 10.1 zeigt T-Diagramme [Bra61, ES70], die die Entwicklung von MontiCore übersichtsartig darstellen. Ein T bezeichnet dabei eine einzelne Software mit einem Namen (oben zentriert), die die Eingabesprachen (links) in die Ausgabesprachen (rechts) übersetzt. Die Software ist dabei selbst in einer Sprache implementiert (unten). Weitergehende Erklärungen der Notation finden sich zum Beispiel in [Wat00].

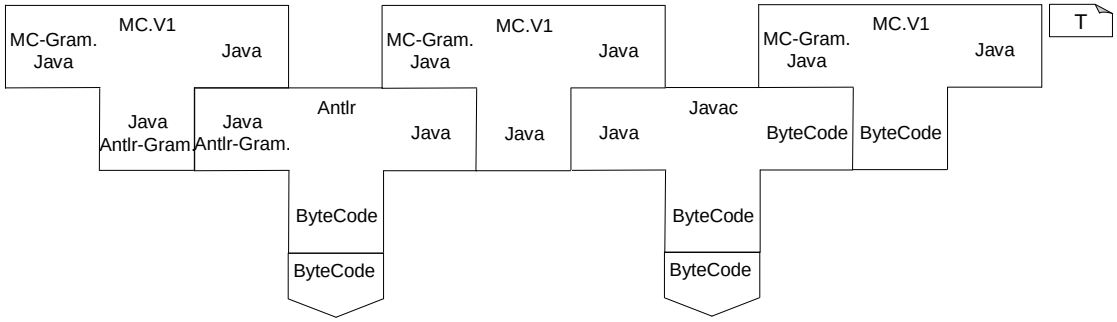
Die Abbildung 10.1(a) zeigt, wie die erste MontiCore-Version mit Hilfe von Antlr-Grammatiken und Java implementiert wurde. Die Grammatiken wurden dann durch Antlr in Java umgewandelt und zusammen mit dem handcodierten Anteil durch den Javac compiliert. Sowohl Antlr als auch der Javac sind Systeme, die im Bootstrapping-Verfahren entstanden sind, was hier allerdings aus Übersichtlichkeitsgründen nicht dargestellt wird. Die entstandene Software *MC.V1* kann die Eingabesprachen von MontiCore (MC-Gram.) verarbeiten und übersetzt sie nach Java. Sie selbst ist Bytecode implementiert (aus vorangegangener Compilierung entstanden) und ist daher auf einer Java-VM lauffähig.

Die Abbildung 10.1(b) zeigt, wie die folgenden Versionen mit einer Kombination von Modellen in MontiCore-Eingabesprachen und Java implementiert und durch die Vorgängerversion nach Java übersetzt wurden. Eine anschließende Compilierung mit dem Javac erzeugt ByteCode.

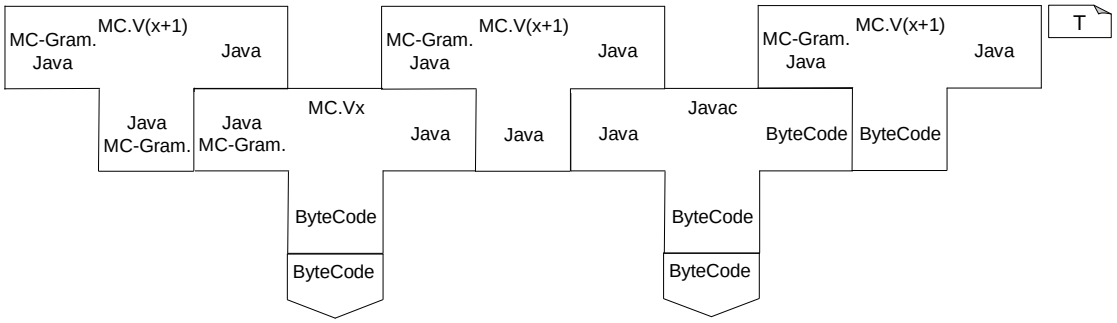
Beim Bootstrapping von MontiCore werden die Grammatiken der aktuellen Version Y mit der Vorgängerversion X (X = Y-1) verarbeitet. Bei der Ausführung der Version Y wird allerdings die Version Y der Laufzeitumgebung `MontiCoreRE` verwendet und nicht wie sonst die Laufzeitumgebung, die zur Version des eingesetzten Generators passt (X). Daraus leitet sich ab, dass für die Laufzeitumgebung zumindest immer für eine Version eine Aufwärtskompatibilität gewährleistet werden muss, weil ja auch die Klassen, die durch die letzte Version des Generators innerhalb der eigenen Entwicklung erzeugt wurden, mit der aktuellen Laufzeitumgebung funktionieren müssen. Die Kompatibilität über mehrere Versionen hinweg ist nicht notwendig.

Die Abbildung 10.1(c) stellt den Einsatz von MontiCore für die generative Entwicklung in einem Entwicklungsprojekt dar, bei dem immer ein zweistufiger Übersetzungsprozess verwendet wird. Dabei erzeugt MontiCore zunächst Java-Quellcode, der dann von einem Java-Compiler übersetzt wird. Auf diese Art kann ein DSLTool entwickelt werden, was DSL-Modelle einlesen kann und diese in Java-Quellcode überführt.

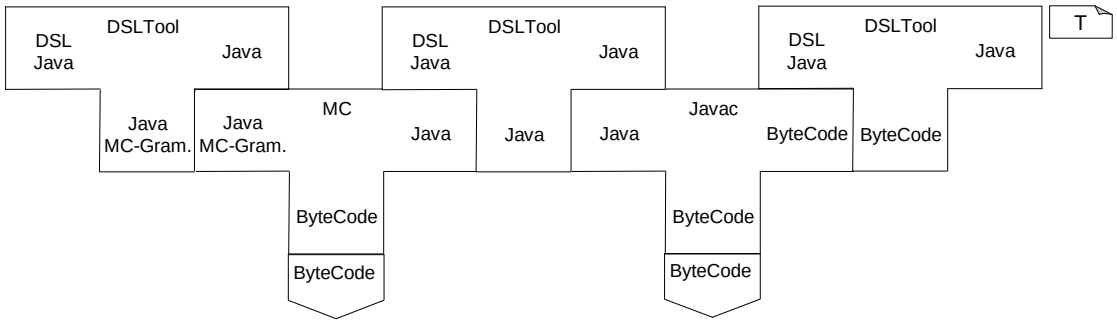
Dieses generative Werkzeug kann dann seinerseits zur Implementierung eines Programms eingesetzt werden, wie in Abbildung 10.1(d) dargestellt, indem DSL-Modelle zusammen mit Java den Quellcode des Programms bilden. Dieser wird durch das DSLTool in reinen Java-Code umgewandelt, indem die Codegenerierung für die DSL-Modelle genutzt wird. Das Resultat wird vom Java-Compiler in ein ausführbares Programm übersetzt.



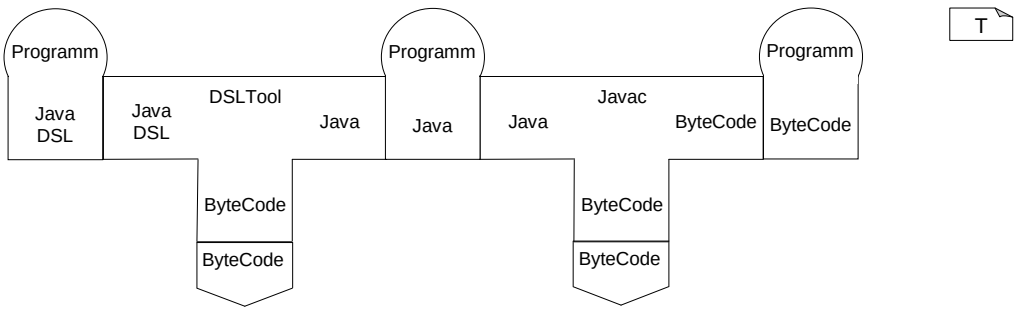
(a) Entwicklung der ersten MontiCore-Version mit Antlr



(b) Entwicklung der folgenden Versionen mit MontiCore



(c) Zweistufige Compilierung beim Einsatz von MontiCore zur Entwicklung eines DSLTools



(d) Zweistufige Compilierung beim Einsatz des DSLTools zur Entwicklung eines Programms

Abbildung 10.1: Entwicklung und Einsatz von MontiCore

Der Build-Prozess für die MontiCore-Entwicklung sowie die Verwendung von MontiCore zur Erstellung generativer Werkzeuge und deren Verwendung ist vollständig über Ant-Skripte [Mat05] automatisiert. Eine besondere Schwierigkeit innerhalb der MontiCore-Entwicklung ist, dass Module andere Module als Generator benutzen. In der normalen Softwareentwicklung müssen Module einer Software nur dann neu kompiliert werden, wenn Veränderungen an den Modulen selbst auftreten. Bei der generativen Entwicklung ist eine Neukompilierung aber auch dann notwendig, wenn sich Generatoren verändern, die in einem Modul eingesetzt werden. Die Ant-Skripte analysieren daher, ob sich eingesetzte Generatoren oder deren Quellen verändert haben, und erzeugen nötigenfalls die generierten Klassen neu aus den Modellen.

10.2.3 Qualitätssicherung

Die Qualitätssicherung des MontiCore-Generators erfolgt im Wesentlichen durch drei sich ergänzende Maßnahmen:

- Modultests werden für algorithmisch schwierige Elemente der Implementierung und für wieder verwendbare Anteile der Laufzeitumgebung eingesetzt.
- Die eigentliche Codegenerierung wird durch Integrationstests überprüft. Dazu werden in der Komponente `featureDSL` Grammatiken und Sprachdateien abgelegt, für die Quellcode generiert wird. Neben dem trivialen Test, dass die Dateien fehlerfrei durch MontiCore verarbeitet werden und anschließend compilierbar sind, gibt es in JUnit formulierte Testfälle, die Teile der generierten Software ausführen und charakteristische Eigenschaften überprüfen.
- Zur Überprüfung der Implementierung der Kontextbedingungen gibt es neben den positiven Tests auch negative Tests in der `featureDSL`. Da fehlerhafte Grammatiken und Sprachdateien stets durch die Prüfung von Kontextbedingungen auffallen sollen, wird für die entsprechenden negativen Beispiele geprüft, ob die erwarteten Fehlermeldungen auch wirklich erzeugt werden. Dabei wird der eindeutige Identifikator von Fehlermeldungen verwendet, um vom konkreten Fehlermeldungstext zu abstrahieren.

Die Vorwendung von Integrationstests hat sich als deutlich effizienter als die Formulierung von Modultests erwiesen, weil von der konkreten Realisierung abstrahiert werden kann und nur die gewünschten Eigenschaften überprüft werden. Dadurch sind die Tests auch zu einem gewissen Grad Änderungen der Codegenerierung gegenüber resistent, müssen also nicht verändert werden, wenn die Codegenerierung angepasst wird.

10.2.4 Organisation

Die Entwicklung des MontiCore-Frameworks erfolgt mittels eines agilen iterativen Prozesses. Dazu wird am Anfang einer Iteration eine Feature-Planung vollzogen, die die wesentlichen Arbeitspakete der nächsten ein bis zwei Monate festlegt. Dabei wurde das Bugtracking-System Bugzilla [Bug] bis zur Version 1.1.8 verwendet, das auf gleichberechtigte Art und Weise erkannte Defekte und neue Features in einem einheitlichen Format auflistet. Danach wurde auf das Bugtracking-System Trac [Trac] umgestellt, um eine genauere Releaseplanung vornehmen zu können. Nach Schließen aller relevanten Bugs und Implementierung der neuen Features wird die entsprechende Version erstellt und die Planung für die nächste Version begonnen.

Das Build-System wird bei jeder Veränderung des Quellcodes in der Versionsverwaltung gestartet und Berichte an den Autor werden versendet. Einmal am Tag wird ein Bericht mit statistischen Daten über die Software und die Ausführung der Testfälle erstellt, die Software im Auslieferungsmodus gebaut und bei Erfolg über die Versionsverwaltung den Entwicklern zur Verfügung gestellt. Somit können die Entwickler stets eine aktuelle Version mit einer vorinstallierten Arbeitsumgebung aus der Versionsverwaltung erhalten und die Fortschritte der Entwicklung validieren.

10.2.5 Zusammenfassung

Die MontiCore-Entwicklung selbst stellt in Bezug auf die Größe des Projekts und die Anzahl der damit erstellten Modelle die ausführlichste Fallstudie für das MontiCore-Framework dar. Die Erfahrungen durch die Nutzung von MontiCore haben direkt die Entwicklung beeinflusst, weil die MontiCore-Entwickler auch gleichzeitig Nutzer der Technologie sind.

Das DSLTool-Framework hat sich bei der Realisierung von MontiCore als nützlich erwiesen. Durch den modularen Aufbau lassen sich neue Codegenerierungen in das Framework integrieren ohne die existierende Funktionalität zu beeinträchtigen. Die in Kapitel 6 beschriebenen Erweiterungspunkte wie Konzepte und Plattformen konnten problemlos zur Referenzarchitektur des DSLTool-Frameworks hinzugefügt werden.

Die Eigenschaften der MontiCore-Eingabesprachen wurden zum Teil auch von ihrer eigenen Struktur beeinflusst. Einbettung passt gut zum Erweiterungsfähigkeit des MontiCore-Grammatikformats, weil hier modular Konzepte hinzugefügt werden sollten, die eventuell eine andere lexikalische Syntax verwenden und durch ein Schlüsselwort voneinander unterschieden werden. Sprachvererbung erlaubt die Realisierung einer kommandozeilenbasierten MontiCore-Version, die sich gut in Buildprozesse einbinden lässt. Die ebenfalls gewünschte Editor-Unterstützung wird so realisiert, dass die Sprachen abgeleitet und um die Konzepte zur Eclipse-Integration ergänzt werden. Das Konzepte zur Eclipse-Integration ist ausdrucksstark genug, um die komplette Eclipse-Integration von MontiCore zu erzeugen, so dass auch hier ein Bootstrapping stattfindet.

Bei der Entwicklung von MontiCore hat sich insbesondere gezeigt, dass integrierte Evolutionstechniken, die die Sprachdefinition, die darauf basierte Codegenerierung und existierende Modelle zusammen verändern können, einen Produktivitätsgewinn bei der Entwicklung bringen. Die Verwendung der MontiCore-Template-Engine ermöglicht dieses in bestimmten Konstellationen. Eine weitergehende Realisierung von Refactoring-Operationen wäre jedoch wünschenswert, um Veränderungen an den Eingabesprachen einfacher durchführen zu können.

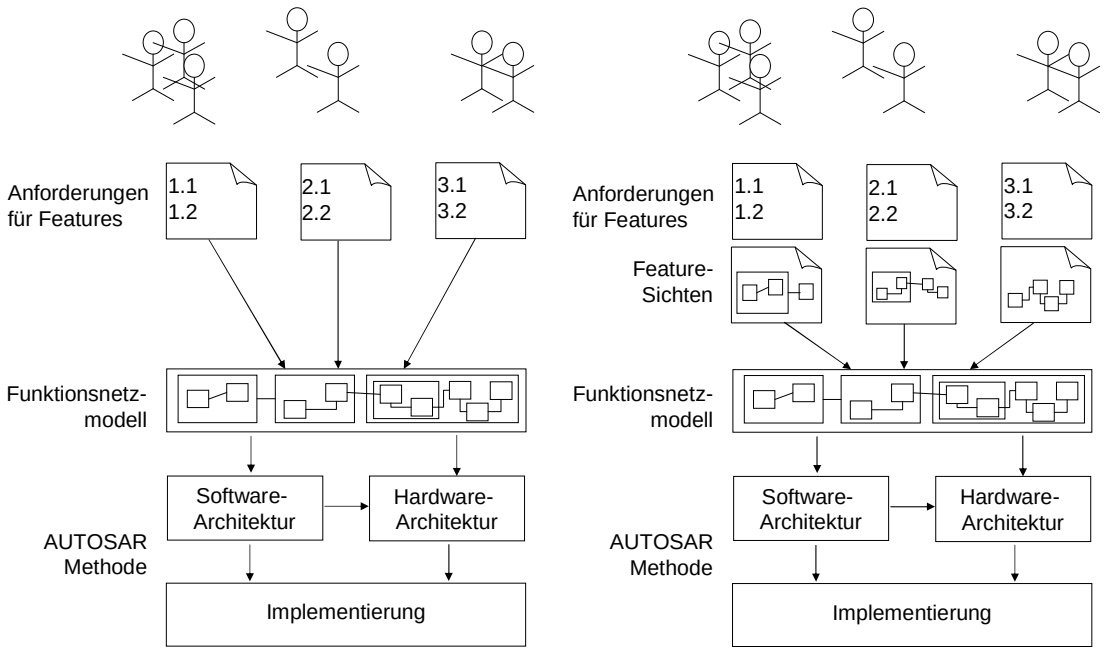
10.3 Funktionsnetze

Die zunehmende Verfügbarkeit preiswerter Elektronik und der gestiegene Bedarf nach Komfortfunktionen und intelligenten Sicherheitsfunktionen in Automobilen sorgen für eine zunehmende Bedeutung der Software innerhalb eines Autos [PBKS07]. Bei der arbeitsteiligen Entwicklung durch Automobilhersteller und Zulieferer, wie sie in der Automobilindustrie üblich ist, spezifiziert der Hersteller die Funktionalität der Software und integriert sie ins Gesamtsystem. Der Zulieferer ist für die Implementierung der Software zuständig. Für die zunehmend komplexeren in Software realisierten Funktionen sind gute Spezifikationstechniken notwendig, um die Kommunikation der beteiligten Partner zu erleichtern und qualitativ hochwertige Lösungen unabhängig von der genauen technischen Realisierung für zukünftige Baureihen verfügbar zu haben [GHK+07, GKPR08, GHK+08b].

Innerhalb des Entwicklungsprozesses für automotive Software gibt es verschiedene Abstraktionsebenen. Die Abbildung 10.2(a) zeigt übersichtsartig, dass zunächst die Anforderungen an so genannte Kunden- und Systemfunktionen von Entwicklern spezifiziert werden (hier einheitlich Features genannt). Dieses geschieht typischerweise unabhängig voneinander, weil viele Entwickler beteiligt sind, die jeweils nur eine Funktionalität beschreiben. Die Features umfassen das gesamte von einem Kunden beobachtbare Verhalten des automotiven Systems, wobei unter Systemfunktionen Elemente wie die Diagnose verstanden werden, die nicht direkt durch den Fahrzeugnutzer beobachtet werden können, aber an den Systemgrenzen beobachtbar sind. Die getrennt entwickelten Anforderungen werden dann zu einer logischen Architektur, dem Funktionsnetz integriert, das dann zu einer Software- und einer Hardwarearchitektur weiterentwickelt wird. Diese Beschreibungen bilden dann die Grundlage der Implementierung. Durch die im Folgenden dargestellte DSL wird die Etablierung einer Zwischenschicht erreicht, die den Übergang von den textuellen Anforderungen zur integrierten Modellierung erleichtert, indem sie die Modellierung eines Features ohne dessen Umgebung ermöglicht (vgl. Abbildung 10.2(b)).

Der Abschnitt 10.3.1 beschreibt zunächst die Rahmenbedingungen, unter denen die DSL eingesetzt werden soll. In Abschnitt 10.3.2 wird die existierende DSL und ihre erarbeiteten Erweiterungen vorgestellt. In Abschnitt 10.3.3 wird die Umsetzung in ein auf dem DSLTool-Framework basierenden Werkzeug dargestellt. Der Abschnitt 10.3.4 vergleicht die Umsetzung mit verwandten Arbeiten. Der Abschnitt 10.3.5 fasst die Darstellungen kurz zusammen.

Die folgende Darstellung basiert zum Teil auf den Veröffentlichungen [GHK+07, GKPR08, GHK+08b].



(a) Entwicklungsprozess ohne Feature-Sichten (b) Entwicklungsprozess mit Feature-Sichten

Abbildung 10.2: Vereinfachter automotiver Entwicklungsprozess (angelehnt an [GHK+08b])

10.3.1 Rahmenbedingungen

Traditionell wird die Entwicklung von automotiver Software nicht als Kernkompetenz der Automobilhersteller verstanden und daher durch Zulieferer erledigt. Die Zulieferer entwickeln in Eigenregie die vom Hersteller gewünschte Funktionalität als Bauteile oder –gruppen. Die Automobilindustrie profitiert stark von der Zusammenarbeit mit Zulieferern. Diese besitzen eine hohe Kompetenz bei der Erstellung einzelner Bauteile und -gruppen für bestimmte Domänen innerhalb eines Automobils, insbesondere auch weil sie untereinander in einer Konkurrenzsituation stehen und fachliches Wissen einer größeren Zahl von Herstellern anbieten können und so ihre Entwicklungskosten auf mehrere Projekte verteilen können. Bei gemeinsam entwickelten Funktionen wird typischerweise für eine gewisse Zeit eine exklusive Vermarktung vereinbart, um für den Hersteller ein Alleinstellungsmerkmal zu ermöglichen.

Mit zunehmender Leistungsfähigkeit und günstiger Verfügbarkeit von elektronischen Komponenten werden viele Funktionalitäten nicht mehr durch physische Bauteile, sondern durch Software realisiert. Dadurch lassen sich entstehende Kosten, Gewicht und der benötigte Bauraum reduzieren.

Anfänglich wurden die Funktionen komplett durch einen einzelnen Zulieferer entwickelt. Mit der Zeit wurden mehr Funktionen realisiert, die auf ähnlichen Eingabedaten basieren. Daher wurden Bussysteme innerhalb des Automobils notwendig, die diese Daten entsprechend verteilen. Für den Automobilbauer, der bisher für die physikalische Integration der Komponenten zuständig war, kam jetzt als zusätzliche Aufgabe hinzu, dass die einzelnen Softwarefunktionen und Busnachrichten erfolgreich integriert werden müssen. Die zunehmende Komplexität der Funktionen, die einen Softwareanteil haben, führte zu einer zunehmenden Anzahl an Steuergeräten in den Automobilen. Diese große Anzahl ist aus verschiedenen Gründen problematisch:

- Die Ausfallwahrscheinlichkeit eines Steuergeräts ist zwar klein, durch die große Anzahl an Geräten bei verteilten Funktionen steigt jedoch die Gesamtausfallwahrscheinlichkeit.

- Der Strombedarf und das Gewicht der Steuergeräte sind relativ hoch, was sich auf die Lebensdauer der Batterie auswirkt und den Spritverbrauch belastet.

- Der benötigte Bauraum und die Verkabelung der Geräte sind eine Herausforderung beim Entwurf der Karosserie und des Rahmens.

Die stetig wachsende Anzahl an Softwarefunktionen erschwert vor allem die Integration der einzelnen Funktionen. Hinzu kommt, dass innerhalb einer Baureihe verschiedene Konfigurationen und eventuell auch Versionen derselben Funktion beachtet werden müssen.

Die Kostenbetrachtung bei der Automobilentwicklung ist typischerweise von den Stückkosten getrieben. Zusätzlicher Entwicklungsaufwand für kleine Kostenersparnisse beim Produkt amortisiert sich bei Volumenmodellen leicht.

Dennoch gibt es folgende Gründe, die eine Wiederverwendung von Software- Spezifikationen wünschenswert machen:

- Mit der zunehmenden Komplexität der Funktionen belasten auch die Entwicklungskosten das Budget eines Fahrzeugs. Dieses gilt insbesondere für Modelle, die in kleineren Stückzahlen produziert werden. Daher ist hier eine Reduzierung der Entwicklungskosten durch Wiederverwendung wünschenswert.

- Die Wiederverwendung der Spezifikationen über Baureihengrenzen hinweg hat neben der Kostenreduzierung noch den Grund, dass für Fahrzeuge eines Herstellers Kundenfunktionen gleich spezifiziert sein sollten. Dadurch kann stärkere Identifikation des Kunden mit der Marke erreicht werden. Diese emotionale Bindung an das Produkt kann sich bei der nächsten Kaufentscheidung positiv auswirken.

- Ein weiterer Grund für die Wiederverwendung ist die Reduzierung der Entwicklungszeiten für Automobile, um so mit neuen Modellen schneller am Markt zu sein und so aktuellen Trends besser folgen zu können. Hier sind zum Beispiel die steigenden Verkaufzahlen so genannter Sport Utility Vehicle (SUV) zu nennen, bei denen eine kurzfristige Entwicklung neuer Modelle schnell zu guten Verkaufszahlen führte. Eine ähnliche Entwicklung ist zukünftig eventuell für besonders spritsparende Modelle oder rein elektrisch betriebene Fahrzeuge zu erwarten.

10.3.2 Funktionsnetze

Zur Modellierung der logischen Architektur der Software eines Automobils werden oftmals Funktionsnetze verwendet. Dabei soll die Kommunikation innerhalb des Softwaresystems dargestellt werden, ohne jedoch auf die physikalische Verteilung innerhalb des Automobils einzugehen. Diese domänenspezifische Notationsform besteht aus Funktionen, die miteinander über Signale kommunizieren. Die Funktionen können hierarchisch geschachtelt sein, um sie zu organisieren. Die Hierarchie stellt jedoch keine Modularität mit definierten Schnittstellen und Geheimnisprinzip dar, wie sie in der Informatik üblich ist [Par72]. Das Verhalten einer Funktion kann sowohl durch geeignete Modellierungssprachen wie Statecharts modelliert werden als auch durch natürlichsprachlichen Text beschrieben werden. Ein Signal beschreibt weniger einen Datentyp, sondern umfasst dessen Bedeutung. Beispielsweise ist die Fahrzeuggeschwindigkeit ein Signal innerhalb des automotiven Funktionsnetzes, das eindeutig von einem Sender ausgeht. Dabei wird von der Art der Nachrichtenübertragung abstrahiert, weil jedes Signal jederzeit einen Wert hat, der allen Empfängern zur Verfügung steht.

Die Abbildung 10.3 zeigt ein Funktionsnetz, das als internes Blockdiagramm der SysML [OMG06c] dargestellt ist. Dabei wird die SysML verwendet, weil sie geeignet ist, Funktionsnetze darzustellen. Gegegenüber anderen Modellierungssprachen wie der UML-RT [SGW94] und der UML 2.1 [OMG07b, OMG07a] hat sie den Vorteil, dass sie nicht eine objektorientierte Terminologie mit Klassen und Objekten verwendet, sondern den neutralen Begriff Block verwendet. Dadurch wird klargestellt, dass Blöcke nicht zwangsläufig instanzierbar sind und mehrfach in einem Kontext vorkommen können, sondern im Regelfall einmal existieren. Dieses erleichtert die Modellierung von Architekturen. Die SysML fordert dabei nicht die strikte zweistufige Modellierung, so dass Kommunikation über Blockgrenzen möglich ist und eine Portdelegation nicht notwendig wird. Somit lässt sich das Kommunikationsparadigma der Funktionsnetze entsprechend modellieren. Die folgenden Erweiterungen der Funktionsnetze werden wiederum als SysML-Interne-Blockdiagramme (IBD) dargestellt. Hier ermöglicht die SysML zwischen der Art des Diagramms, was für alle hier verwendeten Diagramme IBD ist, und der Verwendung des Diagramms zu unterscheiden.

Die Abbildung 10.4 zeigt die textuelle Version des Funktionsnetzes aus Abbildung 10.3. Zur Realisierung eines prototypischen Werkzeugs, das auf dem DSLTool-Framework basiert, wurde diese Syntax verwendet.

Die Blöcke in den Blockdiagrammen der SysML können als stromverarbeitende Funktionen aufgefasst werden. Somit kann die Notation mit Focus [BS01] semantisch fundiert werden.

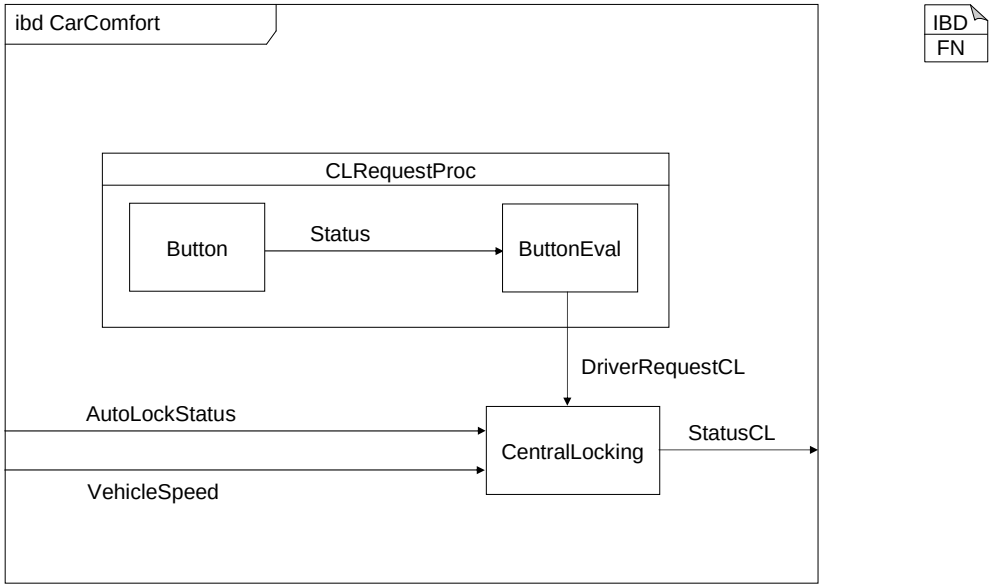


Abbildung 10.3: Funktionsnetz

Funktionsnetz

```
1 idb CarComfort {
2
3   CLRequestProc {
4           Button;
5           ButtonEval;
6           Button - Status -> ButtonEval;
7   }
8
9   ButtonEval - DriverRequestCL -> CentralLocking;
10  extern - AutoLockStatus -> CentralLocking;
11  extern - VehicleSpeed -> CentralLocking;
12  CentralLocking - StatusCL -> extern;
13 }
```

Abbildung 10.4: Textuelle Version des Funktionsnetzes aus Abbildung 10.3

Innerhalb einer Industriekooperation des Instituts mit einem Automobilhersteller wurden die folgenden Verbesserungsmöglichkeiten für die Verwendung von Funktionsnetzen identifiziert [GHK+08b]:

- Die mehrfache Verwendung derselben Funktion innerhalb eines automotiven Systems ist nicht möglich, weil keine Modularität innerhalb der Notation existiert. Diese mehrfache Verwendung dient dazu, wiederkehrende Spezifikationen wie Diagnose oder mehrfach auftretende Elemente wie Sensoren auf strukturierte Art wiederholt zu verwenden.

- Die funktionale Architektur eines Systems lässt sich entlang zweier sich überlappender Dimensionen strukturieren: Erstens die hierarchische Dekomposition und zweitens die Aufteilung in verschiedene Kundenfunktionen. Die Darstellung als Funktionsnetze erlaubt nicht die Darstellung beider Dimensionen.

- Funktionsnetze umfassen meistens nur den durch Software zu realisierenden Teil und nicht Rückkopplungen über die reale Welt. Dies ist nahe liegend, weil nur dieser Teil realisiert werden muss, aber dadurch wird das Systemverständnis der einzelnen Funktionen erschwert.

- Die Darstellung als Funktionsnetze erlaubt nicht die Modellierung der Modi eines automotiven Systems. Unter einem Modus wird ein Systemzustand verstanden, den das System oder ein Subsystem annehmen kann, in dem es ein signifikant anderes Verhalten zeigt. Ein typisches Beispiel hierfür sind Rückfallebenen bei gestörter Sensorik oder Aktorik.

- Funktionsnetze erlauben nicht die Darstellung von Varianten. Unter einer Variante wird eine alternative Funktion verstanden, wobei innerhalb eines automotiven Systems immer nur eine vorkommt.

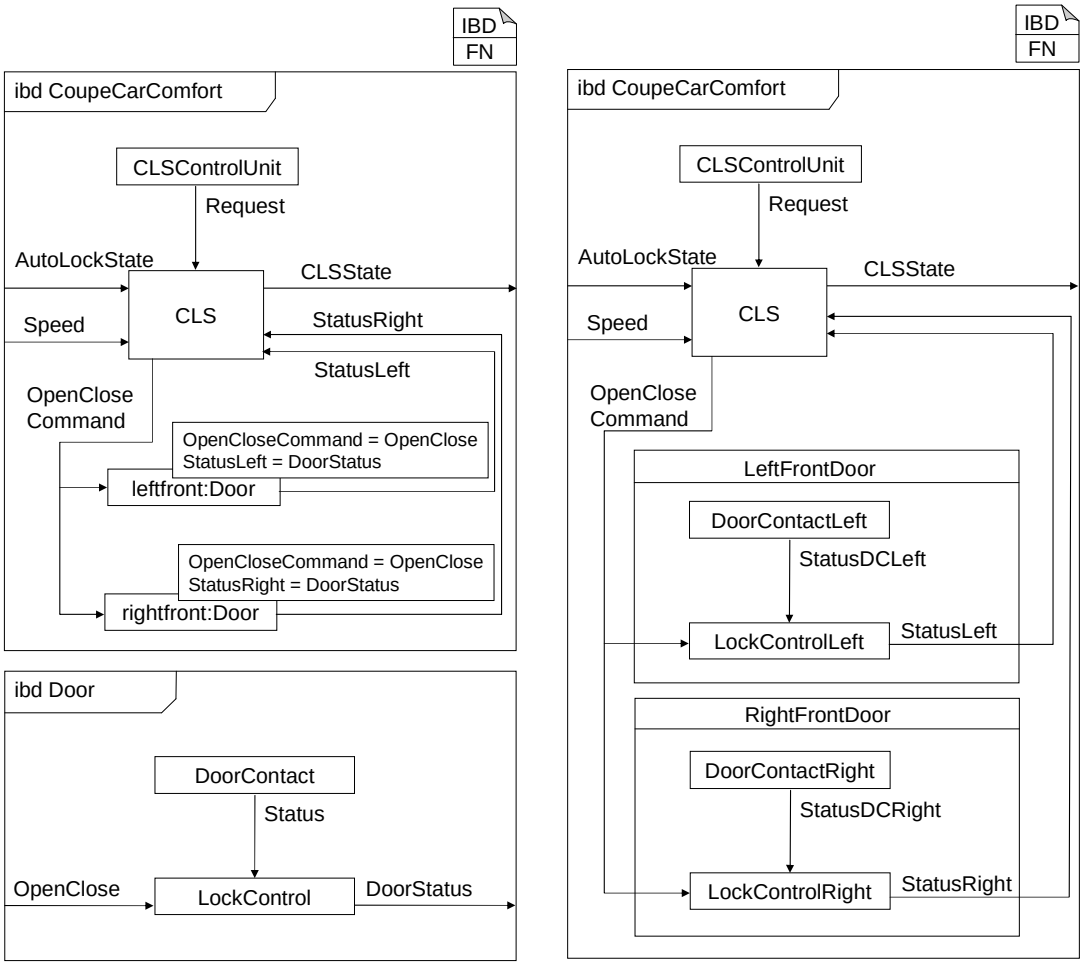
Daher wurden die folgenden Erweiterungen der Funktionsnetznotation definiert, die hier übersichtsartig dargestellt werden. Weitere Details zu den einzelnen Aspekten finden sich in den Veröffentlichungen [GHK+07, GKPR08, GHK+08b].

Instanzierung

In einer objektorientierten Methodik ist es üblich, dass Klassen in einem Kontext instanziert werden können und als Schablone für die Objekte fungieren. Innerhalb der logischen Architektur eines automotiven Softwaresystems, die durch ein Funktionsnetz beschrieben wird, können Funktionen jedoch nicht mehrfach verwendet werden. Dieses konzeptuelle Defizit von Funktionsnetzen führt dazu, dass keine feingranulare Wiederverwendung stattfinden kann, indem Teile einer Spezifikation in einem anderen Zusammenhang genutzt werden können.

Funktionsnetze können derart erweitert werden, dass ein Instanzierungskonzept zur Verfügung steht. Dazu wird eine Funktion durch ein eigenes IBD beschrieben, um dann innerhalb anderer IBDs verwendet zu werden. Wendet man dieses Vorgehen direkt an, ergibt sich jedoch das Problem, dass ein gesendetes Signal des mehrfach verwendeten IBDs nun auch mehrere Sender hat. Dieses widerspricht der zentralen Konsistenzbedingung von Funktionsnetzen und ist missverständlich, weil meistens nur bestimmte Funktionen die validen Empfänger bestimmter Signale sind und keine Broadcastkommunikation gemeint ist. Daher ist bei der mehrfachen Verwendung einer Funktion eine genaue Definition der Schnittstelle notwendig. Grundsätzlich bildet die Grenze zwischen den Funktionen eine Sichtbarkeitsgrenze, über die Signale nicht verfügbar sind. Daher muss die zentrale Konsistenzbedingung, dass jedes Signal genau einen Sender hat, insofern modifiziert werden, als dieses für jede abgeschlossene Einheit von Funktionen ohne Instanzierung gilt. Damit eine Kommunikation über die Sichtbarkeitsgrenze dennoch möglich ist, wird an der Schnittstelle definiert, wie ein Signal innerhalb des IBDs in ein Signal außerhalb des IBDs umgesetzt wird. Nicht aufgeführte Signale sind nicht über die Grenzen verfügbar.

Zur Verdeutlichung des Sachverhalts zeigt die Abbildung 10.5(a) zwei IBDs, die zusammen eine logische Architektur bilden. Die Abbildung 10.5(b) zeigt ein ähnliches Funktionsnetz, das auf die Instanzierung verzichtet. Hier wird jedoch nicht deutlich, dass für die Blöcke `LeftFrontDoor` und `RightFrontDoor` dieselbe Spezifikation hinterlegt ist. Eine textuelle version findet sich in Abbildung 10.6.



(a) Logische Architektur mit Instanzierung (b) Logische Architektur ohne Instanzierung

Abbildung 10.5: Modellierung der logischen Architektur mit Funktionsnetzen

Sichten

Ein zentraler Mechanismus für die Verwendung von Funktionsnetzen ist die Sichtenbildung. Eine Sicht bezieht sich immer auf die logische Architektur, die durch ein oder mehrere Funktionsnetze beschrieben wird. Eine Sicht enthält grundsätzlich eine Auswahl der Blöcke und Signale der logischen Architektur. Dabei können beliebige Elemente ausgelassen werden, um eine zugängliche Darstellung zu erreichen. Ergänzt werden können Elemente, die keinen elektronischen Anteil haben, sondern physische Elemente oder die Umgebung darstellen. Diese werden durch den Stereotypen `«env»` gekennzeichnet und können zusätzlich über so genannte `H-`, `E-` oder `M-`Ports verfügen, die eine hydraulische, elektrische oder mechanische Verbindung darstellen. Somit lassen sich Rückkopplungen, die außerhalb des Systems stattfinden, explizit modellieren.

Eine Sicht ist konsistent zum vollständigen Funktionsnetz, wenn die folgenden Bedingungen erfüllt sind:

1. Jeder Block, der nicht mit dem Stereotypen `«env»` gekennzeichnet ist, muss Teil der logischen Architektur sein.

Funktionsnetz

```
idb CoupeCarComfort {

  CLSControlUnit;

  CLS;
  CLSControlUnit - Request -> CLS;
  extern - AutoLockState -> CLS;
  extern - Speed -> CLS;
  CLS - CLSState -> extern;


  Door leftfront with {
    OpenCloseCommand = OpenClose;
    StatusLeft = DoorStatus;
  }

  leftfront - StatusLeft -> CLS;

  Door rightfront with {
    OpenCloseCommand = OpenClose;
    StatusRight = DoorStatus;
  }

  rightfront - StatusRight -> CLS;
}
```

Funktionsnetz

```
idb Door {

  DoorContact;

  DoorContact - Status -> LockControl;
  extern - OpenClose -> LockControl;

  LockControl - DoorStatus -> extern;
}
```

Abbildung 10.6: Textuelle Version des Funktionsnetzes aus Abbildung 10.5(a)

2. Sichten müssen die Teil-Ganze-Beziehungen der logischen Architektur respektieren: Wenn eine solche Beziehung in einer Sicht existiert, dann muss sie auch in der logischen Architektur existieren. Zwischenschichten dürfen dabei ausgelassen werden.

3. Umgekehrt: Wenn zwei Funktionen in einer (möglicherweise transitiven) Teil-Ganze-Beziehung in der logischen Architektur stehen, und beide werden in einer Sicht dargestellt, dann muss diese auch in der Sicht dargestellt werden.

4. Normale Kommunikationsbeziehungen (ausgenommen solche, die von `M-`, `E-` oder `H-`Ports ausgehen), die in einer Sicht gezeigt werden, müssen in der logischen Architektur existieren. Wenn dabei in der Sicht Signale erwähnt werden, müssen die auch in der logischen Architektur erwähnt werden. Wenn keine Signale benannt werden, dann muss es mindestens einen Kommunikationslink in der logischen Architektur geben.

5. Eine Kommunikationsbeziehung in einer Sicht muss nicht von der genauen Quelle zum genauen Ziel gezeichnet werden, sondern jeder Oberblock ist ausreichend, wenn das genaue Ziel nicht Teil der Sicht ist.

Eine Sicht ist eine Spezialisierung einer anderen Sicht, wenn beide konsistent zur logischen Architektur sind und zusätzlich die folgende Bedingung gilt:

6. Die Blöcke und Konnektoren in einer spezialisierenden Sicht müssen eine Teilmenge der Elemente der spezialisierten Sicht sein.

Sichten zur Darstellung von Features

Unter einem Feature wird eine Kunden- oder Systemfunktion verstanden, die an der Systemgrenze beobachtbare Funktionalität darstellt. Die Hierarchie innerhalb der Funktionsnetze kann nur sehr bedingt genutzt werden, um solche Funktionalitäten umfassend darzustellen. Der Hauptgrund hierfür ist, dass zentrale Eingaben wie die Fahrzeuggeschwindigkeit an einer Stelle ermittelt werden, aber von vielen Funktionen benötigt werden. Des Weiteren werden einige Funktionen wie die Betriebsbremsfunktion innerhalb verschiedener Systeme wie Komfortfunktionen oder auch innerhalb von elektronischen Handbremsen genutzt. Somit lässt sich bei einer rein hierarchischen komponentenorientierten Modellierung nur der Kernbereich beschreiben. An den Randbereichen müssen dann davon getrennte Funktionalitäten genutzt werden. Von diesen Funktionen steht immer die gesamte Funktionalität zur Verfügung, auch wenn nur ein kleiner Teil benötigt wird. Diese Form der Modellierung erschwert es daher, eine Funktion in ihrer Gesamtheit zu beschreiben. Damit wird sie schwerer zu verstehen und komplizierter für eine zukünftige Baureihe wieder zu verwenden, da sie nur in einem Kontext von anderen Funktionen zu verstehen ist.

Aus diesen Gründen können Sichten zur Modellierung von Features verwendet werden. Dadurch können nur die relevanten Teile dargestellt werden, ohne dass die verwendeten Funktionen vollständig modelliert werden müssen. Es wird eine in sich geschlossene Form der Modellierung erreicht, die zur Erstellung wieder verwendbarer Modelle geeignet ist.

Sichten zur Darstellung von Varianten

Moderne Fahrzeuge sind durch den Endkunden hochkonfigurabel, indem zwischen verschiedenen Ausführungen gewählt werden kann. Dabei kann zusätzliche Funktionalität hinzugekauft werden, die schließlich zu einer anderen Parametrisierung und Programmierung der Steuergeräte führt. Die verschiedenen Varianten für die Programmierung eines Steuergeräts werden jedoch meistens durch dasselbe Softwarelastenheft beschrieben, so dass die Modellierung mit Funktionsnetzen diesen Sachverhalt beachten sollte.

In diesem vorgestellten Ansatz wird von einem so genannten „150%-Modell“ gesprochen. Das bedeutet, dass die logische Architektur die Gesamtheit aller Varianten umfasst. Die validen Kombinationen der Varianten werden dabei durch so genannte Feature-Bäume eingeschränkt [CE00]. Zu jeder Variante kann dann eine Sicht formuliert werden, die die Feature-Sicht spezialisiert und nur die in der Variante aktiven Blöcke enthält. Das genaue Verhältnis von Feature-Bäumen zu den Sichten wird in [GKPR08] detailliert beschrieben.

Sichten zur Darstellung von Modi

Im Gegensatz zu einer Variante, die einen unterschiedlichen Softwareumfang bei der Konfiguration der Software bewirkt, verändern sich Modi innerhalb eines Automobils während der Laufzeit. Unter einem Modus versteht man den Zustand eines Subsystems, der

sich von einem anderen Modus dadurch unterscheidet, dass das System ein fundamental unterschiedliches Ein-Ausgabeverhalten zeigt. Ein typisches Beispiel für einen Modus ist die Funktionsdegradation bei erkannten Hardwarefehlern, die zu einer Basisfunktionalität führt.

Die Zusammenhänge und Bedingungen für den Übergang zwischen den verschiedenen Modi lassen sich als Zustandsautomat darstellen. Den Zustandsnamen kommt dabei die besondere Bedeutung zu, dass sie die einzelnen Modi eines Features beschreiben. Die Transitionen zwischen den Zuständen dienen zur Annotation der Bedingungen, die einen Moduswechsel herbeiführen. Mit jedem Modus kann eine Sicht verbunden sein, die die Feature-Sicht verfeinert. Dort werden nur noch die Blöcke dargestellt, die in diesem Modus aktiv sind. Alle anderen Teile der Spezifikation sind in diesem Modus nicht relevant und gelten nicht.

Sichten zur Darstellung von Szenarien

Die Sichten können ebenfalls zur Darstellung von Szenarien verwendet werden, die exemplarische Abläufe des spezifizierten Systems veranschaulichen. Dabei werden gezielt bestimmte Signalwerte oder deren Änderungen notiert. Die Ereignisse werden dabei ähnlich wie in einem UML-Kommunikationsdiagramm nummeriert, um die Reihenfolge anzugeben. Details zur genauen Bedeutung der Notation finden sich in [GHK+08b]. Die Abbildung 10.7 zeigt eine Sicht auf die logische Architektur aus Abbildung 10.5(a), wobei eine exemplarische Kommunikation dargestellt wird. Zunächst muss die Fahrzeuggeschwindigkeit 10 km/h übersteigen, dann muss die Nachricht zum Schließen der Türen ausgelöst werden. Schließlich soll der Status der linken Tür auf geschlossen gesetzt sein. Die Textversion findet sich in Abbildung 10.8.

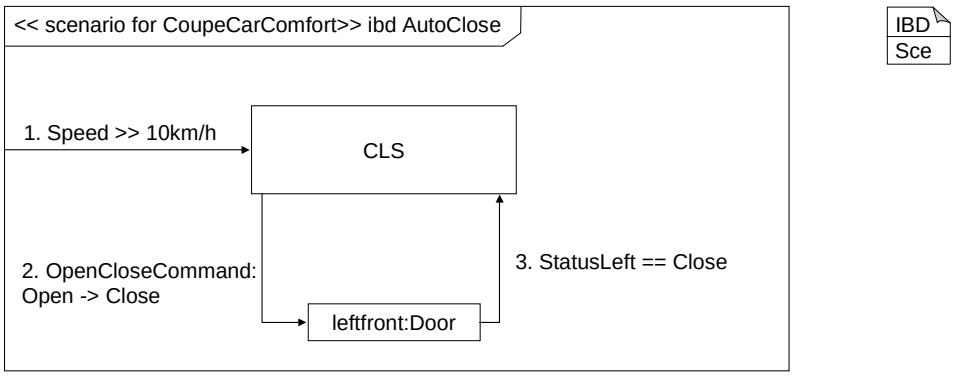


Abbildung 10.7: Funktionsnetz als Szenario

Funktionsnetz

```
1 scenario AutoClose for CoupeCarComfort {
2
3   extern - Speed -> CLS   >> 10 km/h;
4
5   CLS - OpenCloseCommand -> leftfront:Door   Open -> Close;
6
7   leftfront:Door - StatusLeft -> CLS   == Close;
8 }
```

Abbildung 10.8: Textuelle Version des Szenarios aus Abbildung 10.7

10.3.3 Funktionsnetz-DSL

Zur Evaluierung des Ansatzes wurde die beschriebene Erweiterung der Funktionsnetze mit einer DSL innerhalb einer prototypischen Implementierung mit MontiCore umgesetzt. Dabei wurden die folgenden Anwendungsfälle angenommen:

- Unterstützung der Entwickler des automotiven Funktionsnetzes durch die Prüfung von Kontextbedingungen, zum Beispiel dass empfangene Signale auch gesendet werden, und die Sicherstellung, dass jedes Signal nur einmal gesendet wird.
- Verwendung von Instanzierung zur effizienten Wiederverwendung von Funktionen.
- Überprüfung der Sichten bezüglich der Konsistenz mit dem automotiven Gesamtfunktionsnetz.

Das Werkzeug wurde mit Hilfe des DSLTool-Frameworks entwickelt. Dabei werden die folgenden textuellen Modelle als Eingabe für das Werkzeug akzeptiert:

- Mehrere Funktionsnetzdateien, die zusammen die logische Architektur eines Automobils bilden.
- Unabhängig zu der hierarchischen Dekomposition der Funktionen existiert eine Menge an Sichten, die Features im Zusammenhang beschreiben und Elemente der Umgebung enthalten können.
- Eine Auflistung von Varianten pro Feature. Diese kann durch eine Sicht pro Variante ergänzt werden, die die Feature-Sicht spezialisiert.
- Für jedes Feature oder jede Variante kann ein Modidiagramm existieren, das die verschiedenen Modi der Funktion und deren Übergänge beschreibt. Zu jedem Modus kann eine Sicht auf die entsprechende Funktion existieren, die die aktiven Subfunktionen beschreibt.

In [Hab08] wurde weitergehend die Simulation des Funktionsnetzes untersucht, wobei für die einzelnen Basisfunktionen eine Implementierung in der Programmiersprache Java existieren muss. Dadurch ist es möglich, die Spezifikation auszuführen und Szenarien automatisiert zu prüfen.

10.3.4 Verwandte Arbeiten

Funktionsnetzmodellierung mit der UML-RT wird in [Bee04] beschrieben. Der Ansatz wird hier insofern erweitert, als die SysML als Grundlage für die Modellierung verwendet wird. Der Ansatz wird dadurch verbessert, dass der Übergang von den Anforderungen zur logischen Architektur, wie er in frühen Entwurfsphasen passiert, durch die Modellierung von Features und deren Varianten, Modi und Szenarien unterstützt wird.

In [RFH+05, WFH+06] wird die dienstorientierte Modellierung automotiver Systeme erklärt. Die Dienstschicht des Ansatzes besitzt eine konzeptuelle Ähnlichkeit zur Modellierung von Features. Eine Architekturbeschreibungssprache, die den Entwicklungsprozess von den initialen Anforderungen bis zur Realisierung unterstützt, wurde innerhalb des ATESST-Projekts [ATE] basierend auf der EAST-ADL [EAS] definiert. Beide Ansätze unterstützen nicht direkt die Modellierung von Szenarien und die explizite Betrachtung der Umwelt innerhalb der Funktionsnetzmodelle.

AML [BBRS02, BBFR03] betrachtet verschiedene Abstraktionsebenen zwischen Szenarien und der Implementierung. Anders als unsere Szenarien werden diese in [BBRS02] in einem Top-Down-Prozess eingesetzt, so dass aus ihnen später die logische Architektur entwickelt wird.

In [DVM+05] wird die Verwendung von Rich Components erklärt, die eine komplexe Schnittstelle besitzen, die auch nicht-funktionale Anforderungen mit einschließt. Szenarien werden dabei als Sichten auf Komponenten definiert. Im Gegensatz zu den Darstellungen in diesem Abschnitt ist das Ziel von Rich Components weniger eine reibungsloser Übergang von den Anforderungen zum Funktionsnetz, sondern es wird eine vordefinierte Aufteilung in Komponenten angenommen.

Das Autosar-Konsortium [AUT] standardisiert die Software-Architektur automotiver Systeme und erlaubt die Entwicklung austauschbarer Softwarekomponenten. Ein Hauptproblem dieses Ansatzes ist, dass Softwarearchitekturbeschreibungen für frühe Entwicklungsschritte zu detailliert sind, in denen Funktionsnetze eine von den Entwicklern akzeptierte Notation darstellen. Zusätzlich hat, wie in [Gie08] argumentiert, der komponentenbasierte Ansatz Probleme, wenn er auf automotive Features angewendet wird, die stark von der Zusammenarbeit verschiedener Funktionen abhängen, wie dieses bei Automobilsoftware typischerweise der Fall ist.

In [SE06] wird beschrieben, wie sich Sichten zu einem vollständigen System zusammenfügen lassen. Die Vereinigung der Sichten ist in einem strukturierten Top-Down-Entwicklungsprozesse einsetzbar. Nichtsdestotrotz ist dieses Vorgehen problematisch, wenn sich Anforderungen ändern. Die Überprüfung der Sichten und der logischen Architektur auf Konsistenz hingegen erlaubt das Finden von Inkonsistenzen und deren manuelle Auflösung.

In [JKS06, JS08] wird die Sichtenbildung von Metamodellen mittels Triple-Graphgrammatiken beschrieben. Die Techniken sind hier nicht direkt anwendbar, weil die Art der Sichten nicht nicht auf den Metamodellen allgemeingültig definiert wird, sondern individuell auf den einzelnen Modellen festgelegt wird.

10.3.5 Zusammenfassung

Die Fallstudie hat gezeigt, dass mit MontiCore DSLs entwickelt werden können, die nicht primär zur Codeerzeugung und Implementierung von Softwaresystemen gedacht sind. Die verschiedenen Teilsprachen konnten realisiert werden und verschiedene Kontextbedingungen erfolgreich implementiert werden, die sich aus dem Zusammenspiel verschiedener Modelle ergeben.

Zusätzlich wurde klar, dass für die optimale Realisierung analytischer Werkzeuge weitergehende Unterstützung sinnvoll wäre. Das DSLTool-Framework ist so organisiert, dass die unabhängige, modulare Codeerzeugung aus Einzelmodelle unterstützt wird und so inkrementelle Generatoren erstellt werden können. Für ein analystisches Werkzeug wäre es vorteilhaft zwischen den einzelnen Durchläufen Daten sichern zu können, so dass Analysen großen Datenbasen bei Modifikationen nicht neu aubauen müssen.

10.4 JavaTF - Eine Transformationssprache

Die modellbasierte Entwicklung spielt eine zunehmend wichtigere Rolle bei der Erstellung komplexer Softwaresysteme. Dabei sind Modelle die zentralen Abstraktionen, die von den Entwicklern und anderen Beteiligten am Entwicklungsprozess betrachtet werden und auf deren Grundlage Systeme und deren Interaktion beurteilt werden [KKS07, FR08]. Diese Form der modellbasierten Entwicklung kann durch Metamodellierungstechniken wie MOF [OMG06a] oder EMF [BSM+03] formalisiert werden. Die Verwendung einer MontiCore-Grammatik definiert neben der Struktur der Modelle auch eine konkrete textuelle Syntax und ermöglicht so eine kompakte Sprachdefinition.

Bei der modellbasierten Entwicklung können Transformationen zum Beispiel für folgende Anwendungen eingesetzt werden:

- Die Integration verschiedener Softwaresysteme kann durch die explizite Modellierung der Schnittstellen erleichtert werden. Dadurch lässt sich in service-orientierten Architekturen eine Orchestrierung von Diensten erreichen. Die Kommunikation der Systeme untereinander lässt sich dann als Modelltransformation der ausgetauschten Daten darstellen.

- Die Model-Driven-Architecture (MDA) [OMG03] der OMG beschreibt ein Vorgehen zur Verwendung von verschiedenen Abstraktionsebenen zur Erstellung von Software-Systemen. Zwischen den Abstraktionsebenen können Transformationen eingesetzt werden.

- Codegeneratoren können auf zwei Arten von Transformationen profitieren. Erstens können die Schnittstellen durch Modelle beschrieben werden und so existierende Generatoren verwendet werden, indem die Daten auf das Schnittstellenmodell abgebildet werden. Zweitens können komplexe Generatoren so gestaltet werden, dass viele Modellelemente durch Transformationen auf wenige Basiselemente zurückgeführt werden, die sich dann leicht umsetzen lassen.

- Zur Extraktion von abstrakten Modellen, die nur die wesentlichen Informationen für einen Sachverhalt enthalten, aus konkreteren Modellen oder Quellcode.

Aus diesem Grund soll in diesem Abschnitt die Entwicklung einer auf Java basierten Transformationssprache beschrieben werden. Die Kernidee dabei ist, dass sich Java prinzipiell gut zur Implementierung von Algorithmen eignet, aber um bestimmte Elemente erweitert werden sollte, damit sich Transformationen gut formulieren lassen. Dabei besteht eine Transformation aus Regeln, die durch eine spezielle Form der Java-Klasse beschrieben werden.

Diese Fallstudie wurde ausgewählt, weil sie ein Beispiel für die Entwicklung einer eingebetteten DSL mit MontiCore und dem DSLTool-Framework ist, also die Erweiterung einer GPL um domänenspezifische Anteile. Diese zusätzlichen Anteile werden dann direkt auf GPL-Elemente umgesetzt, wobei die GPL in diesem Zusammenhang als Host-Sprache bezeichnet wird. Das Ziel der Fallstudie ist gleichzeitig nicht die Realisierung einer möglichst komfortablen und ausdrucksmächtigen Transformationssprache. Hierfür wäre eine verstärkte Verwendung der Sprache für verschiedene Transformationen und der intensive Vergleich mit existieren Ansätzen notwendig.

Der Abschnitt 10.4.1 beschreibt JavaTF übersichtsartig, wobei in Abschnitt 10.4.2 ein Vergleich zu anderen Transformationssprachen gezogen wird. Der Abschnitt 10.4.3 beschreibt die schrittweise Realisierung. Der Abschnitt 10.4.4 zeigt ein kurzes Beispiel. Der

Abschnitt 10.4.5 beschreibt die technische Realisierung. Der Abschnitt 10.4.6 fasst die Darstellungen kurz zusammen.

10.4.1 Konzeptueller Überblick

Die Transformationssprache JavaTF erlaubt die Realisierung von unidirektionalen Transformationen, die die Objekte eines Eingabemodells in Objekte eines Ausgabemodells übersetzen. Dabei ist die Realisierung von Transformationen möglich, die ein neues Modell erzeugen oder das existierende Modell verändern (Inplace-Transformation). Die Struktur der Modelle wird dabei durch Java-Klassen oder eine MontiCore-Grammatik festgelegt.

Eine Transformation besteht aus einer Menge an Regeln, die auf das Quellmodell angewendet werden. Eine Regel besteht dabei grundsätzlich aus zwei Elementen: Erstens einem Matching-Teil, der den für eine Regel relevanten Ausschnitt eines Modells selektiert und Objekte an Bezeichner bindet, um darauf aufbauend eine Transformation programmieren zu können. Der Ausschnitt des Quellmodells mit gebundenen Bezeichnern wird als Match bezeichnet. Zweitens einen Ausführungsteil, der die Regel ausführt, indem die Objekte des Quellmodells in Objekte des Zielmodells transformiert werden. Dabei können die einmal gefundenen Matches gezielt invalidiert werden. Dadurch kann die doppelte Implementierung von Programmcode in Matching-Teil und Ausführungsteil vermieden werden.

Die Abbildung 10.9 zeigt ein Beispiel einer Transformation mit JavaTF. Dabei werden in den Zeilen 3-5 alle Objekte vom Typ `UMLAttribute` bestimmt, deren Attribut `name` den Wert „`private`“ hat. Für diese Attribute werden nacheinander die Anweisung im Transform-Block (Zeile 8) ausgeführt und so der Name in „`r_private`“ geändert.

JavaTF Transformation

```
1 aspdriven rule Example {
2
3   match {
4    (UmlAttribute u) [ name == "private" ];
5   }
6
7   transform {
8     u.setName("r_private");
9   }
10 }
```

Abbildung 10.9: Ersetzen des Namens eines UML-Attributs

Die Form der Anwendung der Regeln auf das Quellmodell wird dabei durch eine Regelanwendungsstrategie bestimmt. Für die Transformationssprache JavaTF wurden zwei Regelanwendungsstrategien definiert, wobei die Implementierung von weiteren Strategien möglich ist.

- Die quellgetriebene Regelanwendungsstrategie versucht für jedes Objekt im Quellmodell die Regeln der Transformation nacheinander anzuwenden. Dabei wird nur die erste Regel ausgeführt, für die ein Match erzeugt werden kann, und dann dasselbe Verfahren für das nächste Objekt im Quellmodell angewendet. Jede Regel kann im Ausführungsteil eine erweiterte Menge an anderen Regeln festlegen, die dann auf dem Modell mit der quellgetriebenen Regelanwendungsstrategie ausgeführt werden.
- Die aspektgetriebene Regelanwendungsstrategie verwendet eine Liste von Regeln, die in der angegebenen Reihenfolge auf dem Quellmodell ausgeführt werden, solange

Matches existieren. Die einzelnen Regeln können dabei weitere Regeln starten, die zunächst für das gesamte Modell oder eine übergebene Teilmenge des Modells ausgeführt werden. Es kann auch eine Menge von bereits gebundenen Bezeichnern übergeben werden, um die Matching-Teile der abhängigen Regeln zu vereinfachen, weil so nicht bereits erkannte Pattern nochmals formuliert werden müssen. Die aspektgetriebene Ausführungsstrategie kann durch eine explizite Ausführungsreihenfolge in Form einer Strategie parametrisiert werden. Dabei kann neben der alternativen und sequentiellen Anwendung der Regeln auch eine iterative Anwendung erfolgen, bis das Modell einen stationären Zustand erreicht.

10.4.2 Verwandte Arbeiten

Es existiert eine große Anzahl an publizierten Transformationssprachen, die sich durch unterschiedliche Eigenschaften auszeichnen. Eine gute Übersicht über die verschiedenen Ansätze findet sich in [CH06]. Im Folgenden werden die Sprachen dargestellt, die die Realisierung von JavaTF am meisten beeinflusst haben.

SiTra [ABE+06, BHES07] bezeichnet eine Java-API, die zur Implementierung von unidirektionalen Transformationen geeignet ist. Die dabei verwendete Strategie ähnelt der quellgetriebenen Stratgie von JavaTF, wobei eine Ersetzung der Strategie bei SiTra nicht möglich ist.

MT [Tra06] bezeichnet eine Modelltransformationssprache, die mit der Programmiersprache Converge [Tra08] durch Compile-Time-Meta-Programming realisiert wurde. Die Patternsprache von JavaTF ist an Converge angelehnt, wobei Java und nicht Converge in den Pattern verwendet werden kann. Converge verwendet eine quellgetriebene Ausführungsstrategie, die im Gegensatz zu JavaTF nicht austauschbar ist.

Die Atlas Transformation Language (ATL) [JK05] ist eine Transformationsprache, die sowohl quellgetriebene als auch aspektgetriebene Transformationselemente enthält. Die Ausführungsteile werden in einer eigenen Sprache formuliert, die dem Matching-Teil ähnelt. ATL erlaubt die Definition von Hilfsfunktionen, was in JavaTF durch die Verwendung von Java als Host-Sprache ebenfalls möglich ist.

Das auf Graphgrammatiken basierende Werkzeug Fujaba [KNNZ99] erlaubt die Formulierung von Transformationen. Dabei wird der Kontrollfluss einer Transformation explizit auf einer an Aktivitätsdiagrammen angelehnten Notation formuliert. Diese Form der Transformation erlaubt eine ähnliche Strukturierung der Regeln wie die aspektorientierte Regelanwendungsstrategie in JavaTF. Die Notation erlaubt im Gegensatz zu JavaTF die explizite Verwendung von Kontrollflussoperationen wie Bedingungen, aber keine Hüllenbildung. Das Werkzeug Moflon [AKRS06] basiert auf Fujaba, beinhaltet mit den Triple-Graphgrammatiken [Sch94] jedoch eine Form der bidirektionalen Transformation, die durch zwei unidirektionale Transformationen ausgeführt wird.

10.4.3 Entwicklungshistorie

Die Transformationssprache wurde zunächst so entwickelt, dass eine Java-API implementiert wurde, die den vollständigen Funktionsumfang des Transformationsframeworks realisiert. Die Transformationen werden durch die Regelanwendungsstrategie und die Regeln durch spezielle Objekte parametrisiert und auf den Modellen ausgeführt. Es steht der vollständige Funktionsumfang zur Verfügung, wobei die Programmierung einer Transformation jedoch umständlich sein kann, weil die Syntax von Java mit Objekten und Methodenaufrufen keine kompakte Formulierung erlaubt. Daher wurde für ausgewählte Teile der Transformationssprache jeweils eine DSL konzipiert:

1. Der Matching-Teil einer Regel kann durch eine Patternsprache beschrieben werden, die an die Patternsprache von MT [Tra06] angelehnt ist. Details zur Patternsprache finden sich in [The06].

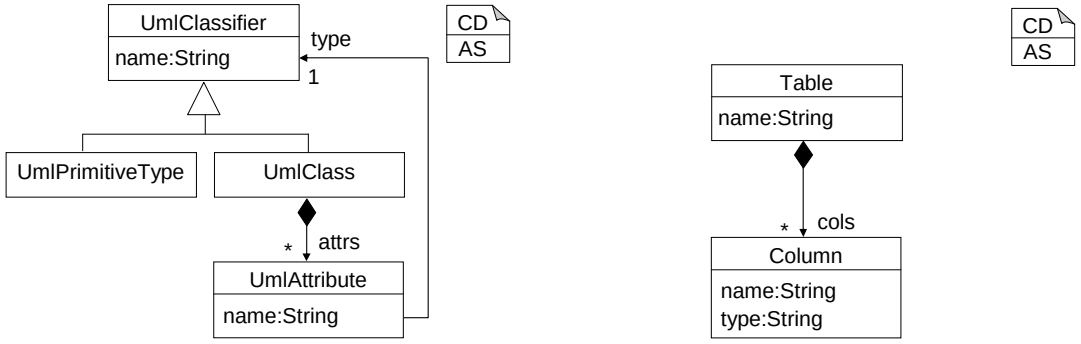
2. Die aspektorientierte Regelanwendungsstrategie lässt sich durch eine selbstentwickelte DSL beschreiben. So lässt sich aufgrund der kompakten Notation ein guter Überblick über die Reihenfolge der Regelanwendung gewinnen. Dabei können die Transformationen sequentiell ausgeführt oder als Alternativen kombiniert werden, so dass die erste Regel ausgeführt wird, die eine Match produziert. Zusätzlich sind Hüllenbildungen möglich, die Regeln so lange ausführen, bis das Modell einen stabilen Zustand erreicht. Details zur aspektorientierten Regelanwendungsstrategie finden sich in [Mah06].

3. Die Regeln lassen sich kompakter formulieren, wenn die Patternsprache verwendet wird, weil gebundene Bezeichner nicht mehr als Attribute der Regeln definiert werden müssen, sondern automatisiert angelegt werden. Daher können die Regeln mit einer speziellen DSL formuliert werden, die die Patternsprache und Java einbettet [Pau09].

10.4.4 Beispiel zum Einsatz von JavaTF

Dieser Abschnitt greift ein Beispiel aus [The06] auf, verwendet jedoch die konkrete Syntax JavaTF und nicht die Java-API. Das Beispiel ist stark vereinfacht und stellt kein reales Beispiel dar, sondern soll die Verwendung der Transformationssprache illustrieren.

Die im Folgenden dargestellte Transformation überführt eine Klasse aus einem Klassendiagramm in eine Tabelle in einem relationalen Modell. Die Abbildung 10.10 zeigt die abstrakte Syntax der Quell- und Zielmodelle. Die Quellmodelle bestehen aus UML-Classifiern, die einen Namen haben. Dabei kann es sich um primitive Datentypen oder Klassen handeln, die ihrerseits eine Menge an Attributen besitzen. Die Attribute haben wiederum einen Classifier als Datentyp. Das Zielmodell besteht aus Tabellen, die ihrerseits aus einer Menge an Spalten bestehen.



(a) Vereinfachtes Metamodell für UML-Klassendiagramme

(b) Relationales Datenmodell

Abbildung 10.10: Abstrakte Syntax der Quell- und Zielmodelle

Die Abbildungen 10.11- 10.13 zeigen drei quellgetriebene Regeln, die zusammen eine Transformation bilden. Dabei formt die Transformation eine Klasse mit ihren Attributen in eine Tabelle um. Besitzt diese Klasse nicht-primitive Attribute, werden diese nicht durch eine zusätzliche Tabelle realisiert, sondern durch die rekursive Anwendung der Transformation, so dass diese in zusätzlichen Spalten derselben Tabelle resultieren. Diese einfache Form

der Transformation funktioniert nur, wenn keine Rekursion innerhalb der Datenstruktur existiert.

Die Transformation besteht aus drei Regeln. Für jedes Objekt des Quellmodells werden die Regeln durchsucht und die erste Regel angewendet, für die ein Match existiert:

- Die Regel `ClassToTable` erzeugt aus einem Classifier eine Tabelle. Die Regel matcht ein Klasse, deren Name an den Bezeichner `n` gebunden wird und daher im Ausführungsteil zur Verfügung steht. Im Anwendungsteil wird ein Objekt vom Typ `Table` mit dem Namen der gematchten Klasse erstellt. Die Attribute der Klasse werden transformiert, indem die Funktion `transformAll()` aufgerufen wird. Dadurch wird die Transformation rekursiv für andere Objekte angewendet. Die so erzeugten Spalten werden der Tabelle hinzugefügt, indem der Rückgabewert der Transformation an die passende Stelle im Zielmodell eingefügt wird.

- Die Regel `PrimitiveTypeAttrToColumn` matcht ein Attribut, dessen Typ durch ein `UmlPrimitiveType` repräsentiert wird. Der Name des Attributs wird an den Bezeichner `n`, der Typ an den Bezeichner `type` und der Name des Typs an `pn` gebunden. Darauf aufbauend erzeugt die Regel aus einem Attribut vom Typ `PrimitiveDataType` eine Spalte in der Tabelle. Der Name der Spalte ergibt sich aus einem eventuellen Prefix und dem Namen des Attributs im Quellmodell.

- Die Regel `UserTypeAttrToColumn` matcht ein Attribut, dessen Typ eine `UmlClass` ist. Der Name des Attributs wird an den Bezeichner `n`, der Typ an den Bezeichner `type` und dessen Attribute an `atts` gebunden. Die Regel verändert das aktuelle Präfix, indem es den Attributnamen anfügt, und ruft die Transformation mit der Methode `transformAll()` rekursiv auf.

10.4.5 Technische Realisierung

Die beiden beschriebenen DSLs wurden mit Hilfe des MontiCore-Frameworks als Grammatiken spezifiziert und zusammen mit der existierenden Java-Grammatik zum Java-Dialekt JavaTF zusammengestellt. Neben den normalen Java-Klassen und Schnittstellen existieren auch gleichberechtigt durch das Schlüsselwort `srcdriven` oder `aspdriven` eingeleitete Regeln. Die Abbildung 10.14 zeigt Ausschnitte der Grammatiken und der daraus resultierenden abstrakten Syntax.

Dabei wird die Grammatik für die Sprache Java als Grundlage verwendet. Beim Parsen wird als Einstiegspunkt die Produktion `CompilationUnit` verwendet, die eine Java-Quelldatei beschreibt. Nach dem Dateikopf aus Paketbezeichnung und Importen folgen möglicherweise mehrere `TypeDeclaration`. Der übersichtlichen Darstellung wegen sind hier nur `ClassDeclaration` und `InterfaceDeclaration` auszugsweise dargestellt. Die Schnittstelle `MemberDeclaration` fasst dabei alle Elemente zusammen, die sich innerhab einer Klasse oder Schnittstelle verwendet werden können. Die Schnittstelle `Expression` wird von allen Java-Ausdrücken implementiert und stellt den Einstiegspunkt beim Parsen dar.

Die Sprache TF erweitert Java, indem eine Unterschnittstelle `RuleMemberDeclaration` eingeführt wird, die alle zusätzlichen `MemberDeclaration` zusammenfasst, die in einer Regel auftreten können.

JavaTF-Quelldateien werden durch einen Generator verarbeitet, der auf dem DSLTool-Framework basiert. Die normalen Java-Elemente werden dabei unverändert übernommen und die erweiterten Elemente auf normale Java-Klassen mit Elementen der JavaTF-API zurückgeführt.

JavaTF Transformation

```
srcdriven rule ClassToTable {

  match {
    (UmlClass) [ n = name, atts = attrs ];
  }

  transform {
    Table table = new Table(n);
    table.getCols().addAll(transformAll(atts));

    return Collections.singletonList(table);
  }
}
```

Abbildung 10.11: Transformation einer Klasse in eine Tabelle

JavaTF Transformation

```
srcdriven rule PrimitiveTypeAttrToColumn {

  match {
    (UmlAttribute) [ n = name, type == (UmlPrimitiveDataType) [pn = name] ];
  }

  transform {
    String colName = prefix + "_" + n;
    String colType = pn;
    Column c = new Column(colName, colType);

    return Collections.singletonList(c);
  }
}
```

Abbildung 10.12: Transformation von primitiven Attributen

JavaTF Transformation

```
srcdriven rule UserTypeAttrToColumn {

  @Inject String prefix = "";

  match {
   (UmlAttribute) [ n = name, type == (UmlClass)[atts = attrs] ];
  }

  transform {
    VarTable vars = new VarTable("prefix", prefix +"_"+ n);
    List transformedAttrs = transformAll(vars, atts);

    return transformedAttrs;
  }
}
```

Abbildung 10.13: Transformation von komplexen Attributen

MontiCore-Grammatik

```
1  grammar Java {
2
3    CompilationUnit =
4      PackageDeclaration? ImportDeclaration*
5      (TypeDeclaration | ";" )*
6
7    interface TypeDeclaration;
8
9    ClassDeclaration
10     implements TypeDeclaration =  ...
11     "{" MemberDeclaration* "}" ;
12
13   InterfaceDeclaration
14     implements TypeDeclaration =  ...
15     "{" MemberDeclaration* "}" ;
16
17   interface MemberDeclaration;
18
19   interface Expression;
20 }
```

MontiCore-Grammatik

```
1  grammar TF {
2
3    RuleDeclaration
4      implements TypeDeclaration = ...
5        RT "rule" Name:IDENT
6        ("extends" ExtendedRule:RuleType)?
7        "{" RuleMemberDeclaration*  "}"
8
9    enum RT = "srcdriven" | "aspdriven" ;
10
11   interface RuleMemberDeclaration
12     extends MemberDeclaration;
13
14   MatchDeclaration
15     implements RuleMemberDeclaration =
16       "match" "{" Pattern* "}";
17
18   TransformDeclaration
19     implements RuleMemberDeclaration =
20       "transform" BlockStatement;
21
22   external Pattern;
23 }
```

MontiCore-Grammatik

```
1  grammar Pattern {
2
3    StartPattern = ...
4
5    external Expression;
6  }
```

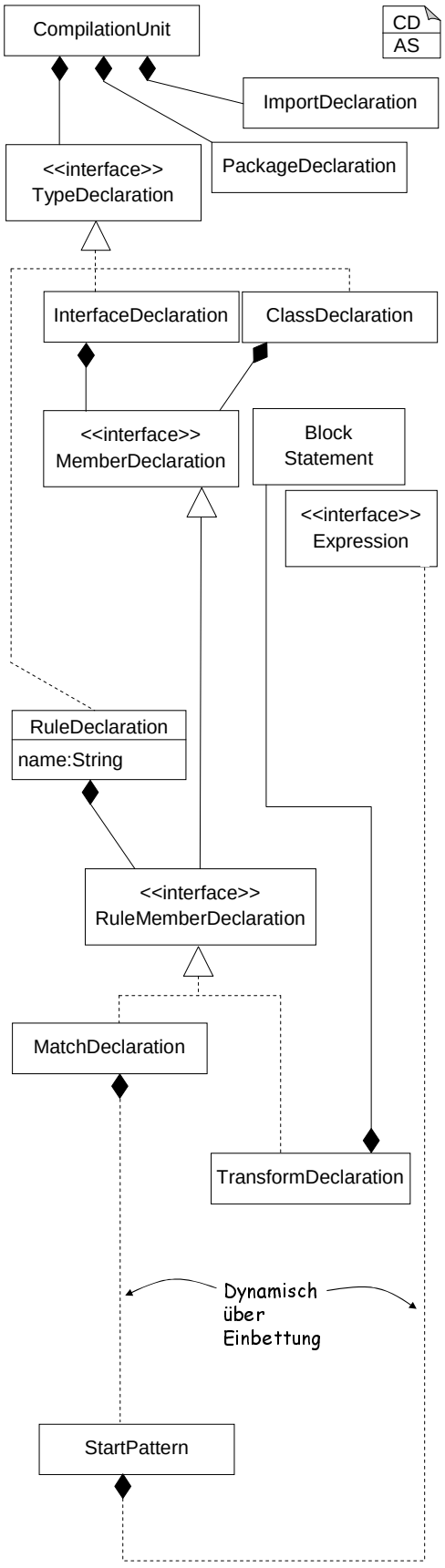


Abbildung 10.14: Grammatiken und abstrakte Syntax von JavaTF

10.4.6 Zusammenfassung

Diese Fallstudie hat gezeigt, dass die Realisierung einer eingebetteten DSL mit dem MontiCore-Framework möglich ist und so die Programmiersprache Java sinnvoll erweitert werden kann. Dabei wird Sprachvererbung und Einbettung eingesetzt, um die Sprache modular zu erweitern. Die Verarbeitung von JavaTF basiert auf dem DSLTool-Framework wodurch die Implementierung in verschiedene Workflows strukturiert werden konnte.

Die derzeitige Realisierung der Transformationssprache prüft die syntaktische Korrektheit der Eingabedateien. Zusätzlich existiert eine Überprüfung grundlegender Kontextbedingungen. In Zukunft sollte die vollständige Überprüfung der Kontextbedingungen der Java-Anteile eingebunden werden, sobald diese im MontiCore-Framework auf eine modulare Art und Weise verfügbar ist.

Teil IV

Epilog

Kapitel 11

Zusammenfassung und Ausblick

In diesem Kapitel werden kurz die Ergebnisse dieser Arbeit zusammenfassend dargestellt und durch einen Ausblick auf mögliche Erweiterungen und weitere Forschungsideen ergänzt.

Zusammenfassung der Ergebnisse

In dieser Arbeit wurde die agile Softwareentwicklung unter Nutzung von DSLs beschrieben. DSLs können in allen Phasen der Softwareerstellung eingesetzt werden und die Entwickler in ihrer Arbeit unterstützen. Sie können dabei zeitgleich und verschränkt zum Produkt innerhalb einer agilen iterativen Entwicklung entstehen. Sie werden dabei in einem eigenständigen Prozess entworfen und anschließend in der Produktentwicklung eingesetzt.

Die DSL-Definition erfolgt mittels eines integrierten Grammatikformats, das sowohl die konkrete als auch die abstrakte Syntax einer Sprache definiert. Dabei wurde ein auf der EBNF basiertes Format so erweitert, dass sich die abstrakte Syntax der DSL ableiten lässt, die verschiedene Klassen, getypte Attribute, Vererbung und Assoziationen enthält und so vergleichbar mit Metamodellierungstechniken ist. Das integrierte Format enthält Elemente, die eine optimierte Abbildung auf einen Parsergenerator erlauben und so die Erzeugung von effizienten Parsern ermöglicht. Durch die Integration beider Dimensionen zu einem Format kann eine Grammatik als zentrale Sprachreferenz verwendet werden, die sowohl für die Verwender der DSL als auch für Werkzeugentwickler nützlich ist.

In dieser Arbeit wurden zwei Formen der Modularität identifiziert, die die Definition einer Sprache in voneinander getrennten Einheiten erlauben. Die Einbettung erlaubt die Definition von unabhängigen Sprachmodulen, die über so genannte Sprachdateien miteinander kombiniert werden können. Die einzelnen Module können unabhängig voneinander compiliert und zur Laufzeit miteinander kombiniert werden. Die Sprachvererbung kann zur Spezialisierung und Erweiterung einer Sprache eingesetzt werden. Dabei wurden Mechanismen, die für die abstrakte und konkrete Syntax existieren, auf das integrierte Format übertragen. Aufbauend auf dieser Modularisierung wurde eine Traversierung der Modelle implementiert, die sowohl die Einbettung als auch die Sprachvererbung beachtet und die modulare Implementierung von Analysealgorithmen und Codegenerierungen ermöglicht.

Zusätzlich wurde ein Erweiterungsmechanismus des Grammatikformats entwickelt: Konzepte ermöglichen die Spezifikation von zusätzlichen Informationen innerhalb einer Sprachdefinition. Der Entwickler eines Konzepts kann die Symboltabelle des Grammatikformats nutzen, so dass die Sprachvererbung und die Ableitung der abstrakten und konkreten Syntax bereits geprüft wurden, und so nur die zusätzlich zu überprüfenden Kontextbedin-

gungen in bestimmten Phasen eingefügt werden müssen. Für die Codeerzeugung stehen ebenfalls bestimmte Erweiterungspunkte zur Verfügung, damit notwenige Funktionalität zu den Parsern und AST-Klassen hinzugefügt werden oder davon unabhängig passend generiert werden kann.

Die Essenz des MontiCore-Grammatikformats wurde formalisiert, indem die abstrakte Syntax als algebraischer Datentyp dargestellt wurde. Erstens wird die Ableitung der abstrakten Syntax definiert, indem aus einer Grammatik ein UML/P-Klassendiagramm abgeleitet wird. Zweitens wird die konkrete Syntax definiert, indem eine MontiCore-Grammatik auf ein spezielles kompakteres Grammatikformat abgebildet wird. Die spezielle Form der Grammatik ist notwendig, weil die Ausdrucksmächtigkeit der MontiCore-Grammatik über die Kontextfreiheit hinausgeht. Die Formalisierung legt die beiden Ableitungsprozesse unabhängig von der Implementierung im MontiCore-Framework fest und definiert sie so eindeutig.

Ein zentraler Aspekt der modellgetrieben Entwicklung ist die automatisierte Übersetzung von DSL-Modellen in Quellcode, damit sie als primäre Artefakte der Softwareentwicklung die Struktur und das Verhalten eines Softwaresystems direkt beeinflussen. Daher ist die Realisierung von Codegeneratoren und analytischen Werkzeugen ein zentraler Bestandteil der Etablierung von DSLs. Die Implementierung der Generatoren wird durch das DSLTool-Framework unterstützt, das eine Referenzarchitektur festlegt und Querschnittsfunktionalitäten zusammenfasst, die bei der Realisierung nützlich sind. Hierdurch ermöglicht das Framework eine effiziente und qualitativ hochwertige Erstellung von generativen, DSL verarbeitenden Werkzeugen. Die Abbildung 11.1 zeigt das Zusammenspiel von MontiCore bei der Erzeugung von Komponenten und Algorithmen mit dem DSLTool-Framework und die Wiederverwendung von Komponenten aus einer Bibliothek.

Das DSLTool-Framework und das MontiCore-Grammatikformat wurden innerhalb eines Bootstrapping-Prozesses entwickelt. Dabei wurde ein Werkzeug erstellt, das Grammatiken und Sprachdateien zur Generierung von Parsern und stark getypten AST-Klassen nutzt. Die beschriebenen Modularitätsmechanismen stehen zur Verfügung und der Generator ist so implementiert, dass eine inkrementelle Codeerzeugung möglich ist. Das Werkzeug ist dabei als Kommandozeilenwerkzeug, als Eclipse-Plugin und als Online-Service verfügbar, wobei eine gemeinsame Codebasis verwendet wird.

Das eingeführte Grammatikformat kann von Sprachentwicklern intuitiv zur Entwicklung von DSLs genutzt werden, weil seine Syntax an verbreitete grammatikbasierten No-

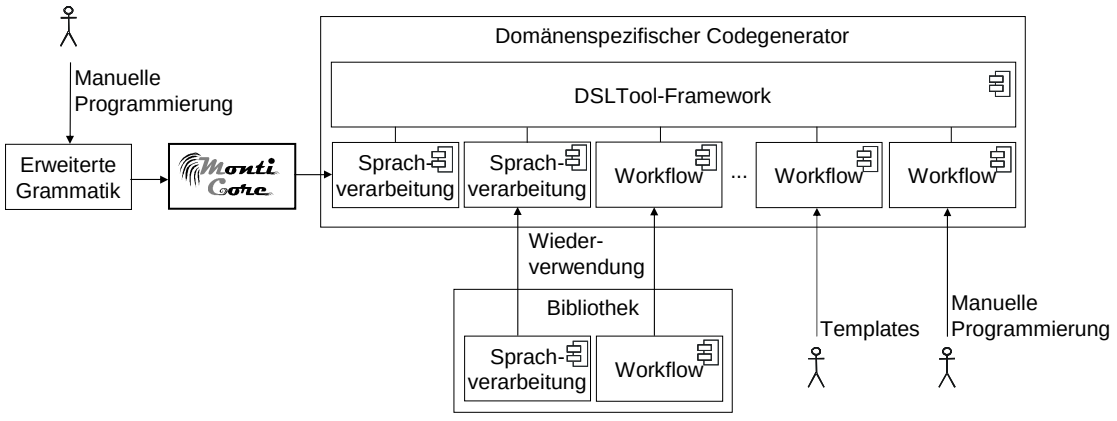


Abbildung 11.1: Übersicht über die Erstellung von domänenspezifischen Codegeneratoren mit MontiCore

tationen angelehnt ist. Zusätzlich werden drei weitere Methoden dargestellt, die die Erstellung einer Grammatik erleichtern. Dabei werden Modellen der Sprachen, die nur die abstrakte oder nur die konkrete Syntax definieren, gezielt auch die andere Syntaxform hinzugefügt und so eine MontiCore-Grammatik erstellt. Des Weiteren wird eine Methode vorgestellt, die von exemplarischem Quellcode ausgeht, der in Zukunft generiert werden soll. Der Entwickler kennzeichnet die variablen Elemente und kann so einen Generator erstellen, dessen Datenmodell automatisch abgleitet werden kann.

Neben der Verwendung des Grammatikformats und des DSLTool-Frameworks innerhalb des Bootstrapping-Prozesses wird MontiCore in verschiedenen Projekten zur Realisierung von DSLs eingesetzt. Darunter finden sich mit Java und C++ auch zwei GPLs, die demonstrieren, dass MontiCore auch komplexe Sprachen realisieren kann. In dieser Arbeit wurden zwei Fallstudien detaillierter dargestellt: Erstens die Ausgestaltung einer DSL zur Funktionsnetzmodellierung innerhalb einer modellbasierten Anforderungsspezifikation von Softwarelastenheften. Zweitens die Implementierung einer auf Java basierten Transformationssprache.

Ausblick

Das MontiCore-Grammatikformat erlaubt die spezifische Codeerzeugung für verschiedene Plattformen. Bei der Realisierung wurden erste Erfahrungen gesammelt, wie eine solche Codeerzeugung strukturiert sein muss, damit ein möglichst allgemeingültiger Generator entwickelt werden kann, der aber flexibel für verschiedene Plattformen angepasst werden kann. Erweiterungen sind hier in zwei Richtungen denkbar: Erstens sollten mehr Plattformen realisiert werden, um Monticore und textuelle DSLs innerhalb verschiedener Umgebungen verfügbar zu machen. Zweitens sollten die Mechanismen grundlegender untersucht werden, um sie auf weitere DSLs übertragen zu können und am besten ins DSLTool-Framework zu integrieren.

Die Codegenerierung innerhalb des MontiCore-Frameworks wurde so gestaltet, dass aus den Grammatiken, den darin enthaltenen Konzepten und den Sprachdateien unabhängig voneinander Code erzeugt werden kann. Weil die Codegeneratoren technische Details zur Realisierung auf verschiedene Plattformen kapseln, die voneinander unabhängig generierten Artefakte aber dennoch kompatibel sein müssen, wurden Schnittstellen zum Informationsaustausch zwischen den Generatoren entwickelt. Dieses ist immer bei der modellgetriebenen Entwicklung notwenig, wenn verschiedene DSLs verwendet werden, die miteinander interagieren sollen. Daher sollte hier eine allgemeingültige Lösung etabliert werden, damit verschiedene, mit dem MontiCore-Framework entwickelte DSLs und Generatoren ohne Änderungen flexibel miteinander kombiniert werden können. Die bei den Grammatiken, Konzepten und Sprachdateien verwendeten Schnittstellen sollten den Ausgangspunkt für eine allgemeingültige Lösung darstellen.

Für die Ausgestaltung des MontiCore-Werkzeugs sind zahlreiche Erweiterungen denkbar. Insbesondere die Ausgestaltung von weiteren Refactoring-Operationen für Grammatiken und deren Integration in die Eclipse-Umgebung erhöht die Entwicklerproduktivität und unterstützt die agile Entwicklung. Die Parsererzeugung sollte auf die Antlr Version 3 umgestellt werden, um die automatische Erzeugung von Prädikaten zu ermöglichen, wodurch eine Grammatik weniger technische Details enthalten muss.

Über die in dieser Arbeit dargestellten technischen Möglichkeiten zur Modularisierung hinaus sollte die Modularisierung von spezifischen Algorithmen, die bei Sprachdefinitionen immer wieder benötigt werden, in Zukunft genauer betrachtet werden. Am häufigsten benötigt wird dabei die Überprüfung von Kontextbedingungen, für die eine wieder verwendbare

Symboltabelleninfrastruktur sinnvoll ist. In diese Kategorie fällt auch die Modularisierung von Templates, wie sie bei einigen anderen Ansätzen ähnlich bereits möglich ist, die jedoch spezifisch auf Einbettung und Sprachvererbung in MontiCore abgestimmt sein muss.

In dieser Arbeit wurde gezeigt, wie sich eine bestimmte DSL, das MontiCore-Grammatikformat, modularisieren lässt. Bei den Fallstudien, der Funktionsnetzmodellierung und der Transformationssprache, wurde ebenfalls eine Modularisierung der Modelle durchgeführt. Ähnlich wie bei Programmiersprachen ist die Modularisierung bei modellgetriebener Entwicklung unersetzlich. Diese Form der Modularisierung erlaubt es, dass mehrere Entwickler gleichzeitig an einem Projekt arbeiten können und sichert somit, dass die modellgetriebene Entwicklung auch für größere Projekte einsetzbar ist. Die Experimente haben gezeigt, dass die Ausgestaltung der Modularität auf der Sprachdefinition festgelegt werden sollte und nicht nur auf der Modellebene. Dadurch lässt sich ein Werkzeug speziell darauf abstimmen, Modelle inkrementell zu verarbeiten und so die Verarbeitungsgeschwindigkeit der Entwicklung zu steigern. Die bei den betrachteten DSLs eingeführten und bei Programmiersprachen verwendeten Modularitätsmechanismen sollten der Ausgangspunkt für eine generelle Untersuchung sein.

Literaturverzeichnis

[ABE+06] David H. Akehurst, Behzad Bordbar, Michael J. Evans, W. Gareth J. Howells, Klaus D. McDonald-Maier. SiTra: Simple Transformations in Java. In *Proc. of International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2006*, LNCS 4199. Springer, 2006.

[Ada91] Stephen R. Adams. *Modular Grammars for Programming Language Prototyping*. Doktorarbeit, University of Southhampton, 1991.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Proc. of European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA) 2006*, LNCS 4066. Springer, 2006.

[Ant] Apache Ant Website `http://ant.apache.org/`. (25.04.2009).

[AP03] Marcus Alanen, Ivan Porres. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Technischer Bericht TUCS 606, Turku Centre for Computer Science, 2003.

[ATE] ATESST Project Website `http://www.atesst.org/`. (25.04.2009).

[AUT] Automotive Open System Architecture (AUTOSAR) Website `http://www.autosar.org`. (25.04.2009).

[BA04] Kent Beck, Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

[BBB+] Kent Beck, Mike Beedle, Arie van Bennekum, et al. Agile Manifesto `http://agilemanifesto.org/`. (25.04.2009).

[BBFR03] Michael von der Beeck, Peter Braun, Ulrich Freund, Martin Rappl. Architecture Centric Modeling of Automotive Control Software. In *Proc. of World Congress of Automotive Engineers 2003*, SAE Technical Paper Series 2003-01-0856, 2003.

[BBRS02] Michael von der Beeck, Peter Braun, Martin Rappl, Christian Schröder. Automotive Software Development: A Model Based Approach. In *Proc. of World Congress of Automotive Engineers 2002*, SAE Technical Paper Series 2002-01-0875, 2002.

[BCD+89] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, Victor Pascual. Centaur: The System. *SIGPLAN Notices*, 24:14–24, 1989.

[BCK03] Len Bass, Paul Clements, Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[BDV07] Martin Bravenboer, Eelco Dolstra, Eelco Visser. Preventing Injection Attacks with Syntax Embeddings. In *Proc. of International Conference on Generative Programming and Component Engineering (GPCE) 2007*. ACM, 2007.

[Bee04] Michael von der Beeck. Function Net Modeling with UML-RT: Experiences from an Automotive Project at BMW Group. In *Proc. of UML Modeling Languages and Applications Satellite Activities*, LNCS 3297. Springer, 2004.

[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 *Proc. of Compiler Construction (CC) 2001*, LNCS 2102. Springer, 2001.

[BHES07] Behzad Bordbar, Gareth Howells, Michael Evans, Athanasios Staikopoulos. Model Transformation from OWL-S to BPEL Via SiTra. In *Proc. of European Conference Model-Driven Architecture - Foundations and Applications (ECMDA-FA) 2007*, LNCS 4530. Springer, 2007.

[Bin00] Robert V. Binder. *Testing Object-Oriented Systems - Models, Patterns, and Tools*. Addison-Wesley, 2000.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72:52–70, 2008.

[BLS98] Don Batory, Bernie Lofaso, Yannis Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proc. of Conference on Software Reuse 1998*. IEEE Computer Society Press, 1998.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21:61–72, 1988.

[Bos97] Jan Bosch. Delegating Compiler Objects: Modularity and Reusability in Language Engineering. *Nordic J. of Computing*, 4:66–92, 1997.

[Bra61] Harvey Bratman. An Alternate Form of the "UNCOL diagram". *Communications of the ACM*, 4:142, 1961.

[BRLG04] Fabian Büttner, Oliver Radfelder, Arne Lindow, Martin Gogolla. Digging into the Visitor Pattern. In *Proc. of International Conference on Software Engineering & Knowledge Engineering (SEKE) 2004*. IEEE Computer Society Press, 2004.

[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.

[BS01] Manfred Broy, Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement.* Springer, 2001.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose. *Eclipse Modeling Framework.* Addison-Wesley, 2003.

[Bug] Bugzilla Website `www.bugzilla.org`. (25.04.2009).

[BV04] Martin Bravenboer, Eelco Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *Proc. of International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA) 2004.* ACM, 2004.

[BV07] Martin Bravenboer, Eelco Visser. Designing Syntax Embeddings and Assimilations for Language Libraries. In *Models in Software Engineering: Workshops and Symposia at MoDELS 2007*, LNCS 5002, 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.

[CE00] Krzysztof Czarnecki, Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[CGP00] Edmund M. Clarke, Orna Grumberg, Doron A. Peled. *Model Checking.* The MIT Press, 2000.

[CGR08] María V. Cengarle, Hans Grönniger, Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, Technische Universität Braunschweig, 2008.

[CH06] Krzysztof Czarnecki, Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CJKW07] Steve Cook, Gareth Jones, Stuart Kent, Alan C. Wills. *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley, 2007.

[Cle07] Thomas Cleenewerck. *Modularizing Language Constructs: A Reflective Approach.* Doktorarbeit, Vrije Universteit Brussel, 2007.

[CN02] Paul Clements, Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[CSW08] Tony Clark, Paul Sammut, James Willans. *Superlanguages: Developing Languages and Applications with XMF.* Ceteva, 2008. `http://www.ceteva.com/docs/Superlanguages.pdf` (25.04.2009).

[Cza05] Krzysztof Czarnecki. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. of International Conference on Generative Programming and Component Engineering (GPCE) 2005*, LNCS 3676. Springer, 2005.

[Des84] Thierry Despeyroux. Executable Specification of Static Semantics. In *Proc. of International Symposium on Semantics of Data Types 1984*, LNCS 173. Springer, 1984.

[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.

[DK98] Arie van Deursen, Paul Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice*, 10:75–92, 1998.

[DKLM84] Veronique Donzeau-Gouge, Gilles Kahn, Bernard Lang, Bertrand Mélèse. Document Structure and Modularity in Mentor. In *Proc. of Software Engineering Symposium on Practical Software Development Environments 1984*. ACM, 1984.

[DKV00] Arie van Deursen, Paul Klint, Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[DS90] Peter DeGrace, Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions*. Yourdon Press, 1990.

[DVM+05] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, Eckard Böde. Boosting Re-use of Embedded Automotive Applications Through Rich Components. In *Proc. of International Workshop on Foundations of Interface Technologies (FIT) 2005*, 2005.

[Ear70] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13:94–102, 1970.

[EAS] EAST-EEA Architecture Description Language, `http://www.east-eea.net`. (25.04.2009).

[EH07] Torbjörn Ekman, Görel Hedin. The JastAdd System - Modular Extensible Compiler Construction. *Science of Computer Programming*, 69:14–26, 2007.

[EMGR+01] Hartmut Ehrig, Bernd Mahr, Martin Große-Rhode, Felix Cornelius, Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer, 2. Auflage, 2001.

[ES70] Jay Earley, Howard Sturgis. A Formalism for Translator Interactions. *Communications of the ACM*, 13(10):607–617, 1970.

[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.

[For02] Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *Proc. of the International Conference on Functional Programming (ICFP) 2002*. ACM, 2002.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[Fow05] Martin Fowler. Generating Code for DSLs, 2005. `http://martinfowler.com/articles/codeGenDsl.html` (25.04.2009).

[FPF95] Bill Frakes, Rubén Príeto-Diaz, Christopher Fox. DARE: Domain Analysis and Reuse Environment. In *Proc. of Workshop on Software Reuse 1995*. ACM, 1995.

[FPR00] Marcus Fontoura, Wolfgang Pree, Bernhard Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley, Boston, MA, USA, 2000.

[FR07] Robert France, Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Proc. of Future of Software Engineering (FOSA) 2007*. IEEE Computer Society, 2007.

[FR08] Robert B. France, Bernhard Rumpe. Model-based Development. *Software and System Modeling*, 7(1):1–2, 2008.

[Gao07] Jimin Gao. *Building an Extensible Modeling Language Framework with Enhanced Attribute Grammars*. Doktorarbeit, University of Minnesota, 2007.

[GBU08] Thomas Goldschmidt, Steffen Becker, Axel Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *Proc. of European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA) 2008*, LNCS 5095. Springer, 2008.

[GC03] Joseph D. Gradecki, Jim Cole. *Mastering Apache Velocity*. Wiley, 2003.

[Gen] Gentleware Apollo `http://www.gentleware.com/apollo.html` (25.04.2009).

[GH98] Etienne Gagnon, Laurie Hendren. SableCC - an Object-Oriented Compiler Framework. In *Proc. of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS) 1998*. IEEE Computer Society, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Bernhard Rumpe. View-Based Modeling of Function Nets. In *Proc. of the Object-Oriented Modeling of Embedded Real-Time Systems Workshop (OMER4) 2007*, October 2007.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshops Modellbasierte Entwicklung Eingebetteter Systeme (MBEES) 2008*, 2008.

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proc. of Embedded Real Time Software (ERTS) 2008*, 2008.

[GHR+03] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, Colin Willcock. An Introduction to the Testing and Test Control Notation (TTCN-3). *Computer Networks*, 42:375–403, 2003.

[Gie08] Holger Giese. Reuse of Innovative Functions in Automotive Software: Are Components enough or do we need Services? In *Tagungsband Modellierungs-Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF) 2008*, 2008.

[GJS05] James Gosling, Bill Joy, Guy L. Steele. *The Java Language Specification*. Addison-Wesley, 3. Auflage, 2005.

[GK03] Jeff Gray, Gabor Karsai. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. In *Proc. of Hawaii International Conference on System Sciences (HICSS) 2003*, 2003.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Tagungsband Modellierungs-Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF) 2008*, 2008.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. 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. Text-based Modeling. In *Models in Software Engineering: Workshops and Symposia at MoDELS 2007*, LNCS 5002, 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 *Companion Volume of International Conference on Software Engineering (ICSE) 2008*. ACM, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Proc. of Modellierung 2006*, LNI P-82. Springer, 2006.

[GME] GME Website `http://www.isis.vanderbilt.edu/projects/gme/`. (25.04.2009).

[GMF] Graphical Modeling Framework Website. `http://www.eclipse.org/gmf/`. (25.04.2009).

[GMW97] David Garlan, Robert T. Monroe, David Wile. ACME: An Architecture Description Interchange Language. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON) 1997*, 1997.

[Gou01] John Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall, 2001.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[GV] Graphviz Website, `www.graphviz.org`. (25.04.2009).

[GZ04] Sabine Glesner, Wolf Zimmermann. Natural Semantics as a Static Program Analysis Framework. *ACM Trans. Program. Lang. Syst.*, 26(3):510–577, 2004.

[Hab08] Arne Haber. Steigerung der Entwicklungseffizienz automotiver Systeme durch Modellierung der Systemarchitektur. Studienarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, Daniel Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proc. of the IEEE*, 79:1305–1320, 1991.

[Hed89] Görel Hedin. An Object-Oriented Notation for Attribute Grammars. In *Proc. of European Conference on Object-Oriented Programming (ECOOP) 1989*, 1989.

[Hed00] Görel Hedin. Reference Attributed Grammars. *Informatica*, 24(3), 2000.

[Her06] Christoph Herrmann. Online Software Transformation Platform. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[HN05] Angel Herranz, Pablo Nogueira. More Than Parsing. In *Proc. of Spanish Conference on Programming and Languages (PROLE) 2005*, 2005.

[Hoa73] Charles A. R. Hoare. Hints on Programming Language Design. Technischer Bericht, Stanford University, 1973.

[Hof08] Conrad Hoffmann. Vergleich der Ausdrucksmächtigkeiten von EMF und dem grammatikbasierten Modellierungstool MontiCore. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

[Höw07] Frank Höwing. Effiziente Entwicklung von AUTOSAR-Komponenten mit domänenspezifischen Programmiersprachen. In *Proc. of Workshop Automotive Software Engineering*, LNI P-110. Springer, 2007.

[HR04] David Harel, Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[HT86] Susan Horwitz, Tim Teitelbaum. Generating Editing Environments Based on Relations and Attributes. *ACM Trans. Program. Lang. Syst.*, 8:577–608, 1986.

[HTML] HTML Spezifikation, `http://www.w3.org/html/wg/html5/`. (25.04.2009).

[Hud98] Paul Hudak. Modular Domain Specific Languages and Tools. In *Proc. of International Conference on Software Reuse 1998*. IEEE Computer Society, 1998.

[Ivy] Apache Ant, Subproject Ivy Website `http://ant.apache.org/ivy/`. (25.04.2009).

[Jac02] Daniel Jackson. Alloy: a Lightweight Object Modelling Notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[JB06] Frederic Jouault, Jean Bezivin. KM3: A DSL for Metamodel Specification. In *Proc. of IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems 2006*, LNCS 4037. Springer, 2006.

[JBK06] Frédéric Jouault, Jean Bezivin, Ivan Kurtev. TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proc. of the International Conference on Generative Programming and Component Engineering (GPCE) 2006*. ACM, 2006.

[JK05] Frédéric Jouault, Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, LNCS 3844, 2005.

[JKS06] Johannes Jakob, Alexander Königs, Andy Schürr. Non-materialized Model View Specification with Triple Graph Grammars. In *Proc. of International Conference on Graph Transformation 2006*, LNCS 4178. Springer, 2006.

[JS08] Johannes Jakob, Andy Schürr. View Creation of Meta Models by Using Modified Triple Graph Grammars. *Electron. Notes Theor. Comput. Sci.*, 211:181–190, 2008.

[JV00] Merijn de Jonge, Joost Visser. Grammars as Contracts. In *Proc. of International Symposium on Generative and Component-Based Software Engineering (GCSE) 2000*, LNCS 2177. Springer, 2000.

[Kah87] G. Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Sciences (STACS) 1987*, LNCS 247. Springer, 1987.

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Nowak, Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technischer Bericht, Software Engineering Institute, Carnegie Mellon University, 1990.

[KKS07] Felix Klar, Alexander Königs, Andy Schürr. Model Transformation in the Large. In *Proc. of European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE) 2007*. ACM, 2007.

[Kli93] Paul Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions Software Engineering Methodology*, 2:176–201, 1993.

[KLV05] Paul Klint, Ralf Lämmel, Chris Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology*, 14:331–380, 2005.

[KMB+96] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, Lisa Walton. A Software Engineering Experiment in Software Component. In *Proc. of International Conference on Software Engineering (ICSE) 1996*. IEEE Computer Society, 1996.

[KNNZ99] Thomas Klein, Ulrich A. Nickel, Jörg Niere, Albert Zündorf. From UML to Java And Back Again. Technischer Bericht tr-ri-00-216, Universität Paderborn, 1999.

[Knu68] Donald F. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 12:127–145, 1968.

[KPKP06] Dimitrios Kolovos, Richard Paige, Tim Kelly, Fiona Polack. Requirements for Domain-Specific Languages. In *Proc. of ECOOP Workshop on Domain-Specific Program Development (DSPD) 2006*, 2006.

[KR05] Holger Krahn, Bernhard Rumpe. Evolution von Software-Architekturen. Informatik-Bericht 2005-04, Technische Universität Braunschweig, 2005.

[KR06a] Holger Krahn, Bernhard Rumpe. *Handbuch der Software-Architektur*, Kapitel Grundlagen der Evolution von Software-Architekturen, Seiten 133–151. dpunkt-Verlag, 2006.

[KR06b] Holger Krahn, Bernhard Rumpe. Techniques For Lightweight Generator Refactoring. In *Proc. of Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, LNCS 4143. Springer, 2006.

[Kru95] Philippe Kruchten. Architectural Blueprints—The "4+1" View Model of Software Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Professional, 3. Auflage, 2003.

[KRV06] Holger Krahn, Bernhard Rumpe, Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Proc. of OOPSLA Workshop on Domain-Specific Modeling 2006*. University of Jyväskylä, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proc. of OOPSLA Workshop on Domain-Specific Modeling 2007*. University of Jyväskylä, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proc. of International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2007*, LNCS 4735. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proc. of International Conference Objects, Models, Components, Patterns (TOOLS-EUROPE) 2008*, LNBIP 11. Springer, 2008.

[KS97] Peter Klein, Andy Schürr. Constructing SDEs with the IPSEN Meta Environment. In *Proc. of International Conference on Software Engineering Environments (SEE) 1997*. IEEE Computer Society, 1997.

[KT08] Steven Kelly, Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley, 2008.

[KW94] Uwe Kastens, William M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31(7):601–627, 1994.

[KW96] Basim M. Kadhim, William M. Waite. Maptool - Supporting Modular Syntax Development. In *Proc. of International Conference on Compiler Construction (CC) 1996*. Springer, 1996.

[Läm01a] Ralf Lämmel. Grammar Adaptation. In *Proc. of Formal Methods Europe (FME) 2001*, LNCS 2021. Springer, 2001.

[Läm01b] Ralf Lämmel. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, LNCS 2029. Springer, 2001.

[Lan66] Peter J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.

[LJ03] Ralf Lämmel, Simon P. Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proc. of Workshop on Types in Language Design and Implementation (TLDI) 2003*, LNCS 4719. ACM, 2003.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Trans. Softw. Eng.*, 21(4):336–355, 1995.

[LM07] Ralf Lämmel, Erik Meijer. Revealing the X/O impedance mismatch (Changing lead into gold). In *Datatype-Generic Programming*. Springer, 2007.

[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) 2001*. IEEE, 2001.

[LV01] Ralf Lämmel, Chris Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31:1395–1438, December 2001.

[LZ09] Ralf Lämmel, Vadim Zaytsev. An Introduction to Grammar Convergence. In *Proc. of Conference on integrated Formal Methods (iFM) 2009*, LNCS 5423. Springer, 2009.

[Mah06] Björn Mahler. Aspektgetriebene Modelltransformationen. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[Mat05] Bernd Matzke. *Ant: Eine praktische Einführung in das Java-Build-Tool*. dpunkt-Verlag, 2. Auflage, 2005.

[Mav] Apache Maven Website `http://maven.apache.org/`. (25.04.2009).

[MBB06] Erik Meijer, Brian Beckman, Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. of the 2006 International Conference on Management of Data (SIGMOD) 2006:*. ACM, 2006.

[MCN] MetaCase. Nokia case study `http://www.metacase.com/papers/metaedit_in_nokia.pdf`. (26.04.2009).

[Meta] MetaCase Website `http://www.metacase.com`. (26.04.2009).

[MFF+06] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, Jean-Marc Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In *Proc. of International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2006*, LNCS 4199, 2006.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, Jean M. Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *Proc. of International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2005*, LNCS 3713. Springer, 2005.

[MHS03] Marjan Mernik, Jan Heering, Anthony M. Sloane. When and How to Develop Domain-Specific Languages. Technischer Bericht SEN-E0309, Centrum voor Wiskunde en Informatica, Amsterdam, 2003.

[MHS05] Marjan Mernik, Jan Heering, Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[MO07] Holt Mebane, Joni T. Ohta. Dynamic Complexity and the Owen Firmware Product Line Program. In *Proc. of International Software Product Line Conference (SPLC) 2007*. IEEE Computer Society, 2007.

[MP99] Vijay Menon, Keshav Pingali. A Case for Source-Level Transformations in MATLAB. In *Conference on Domain-Specific Languages 1999*. ACM, 1999.

[MPS] Meta Programming System Website. `http://www.jetbrains.com/mps/`. (26.04.2009).

[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.

[Nag79] Manfred Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, 1979.

[Nag96] Manfred Nagl (Editor). *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, LNCS 1170. Springer, 1996.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. of International Conference on Compiler Construction (CC) 2003*, LNCS 2622. Springer, 2003.

[Nei80] James M. Neighbors. *Software Construction Using Components*. Doktorarbeit, Department Information and Computer Science, University of California, 1980.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[NR69] Peter Naur, Brian Randell (Editoren). *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969.

[OAW] OpenArchitectureWare Website `http://www.openarchitectureware.com/`. (26.04.2009).

[OMG03] Object Management Group. MDA Guide Version 1.0.1 (2003-06-01), 2003. http://www.omg.org/docs/omg/03-06-01.pdf (26.04.2009).

[OMG04] Object Management Group. Human-Usable Textual Notation V1.0 (04-08-01), 2004. http://www.omg.org/docs/formal/04-08-01.pdf (26.04.2009).

[OMG06a] Object Management Group. MOF Specification Version 2.0 (2006-01-01), January 2006. http://www.omg.org/docs/ptc/06-05-04.pdf (26.04.2009).

[OMG06b] Object Management Group. Object Constraint Language Version 2.0 (OMG Standard 2006-05-01), 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf (26.04.2009).

[OMG06c] Object Management Group. SysML Specification Version 1.0 (2006-05-03), 2006. http://www.omg.org/docs/ptc/06-05-04.pdf (26.04.2009).

[OMG07a] Object Management Group. Unified Modeling Language: Infrastructure Version 2.1.2 (07-11-05), 2007. http://www.omg.org/docs/formal/07-11-04.pdf (26.04.2009).

[OMG07b] Object Management Group. Unified Modeling Language: Superstructure Version 2.1.2 (07-11-02), 2007. http://www.omg.org/docs/formal/07-11-02.pdf (26.04.2009).

[OMG08] Object Management Group. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (2008-06-08), 2008. http://www.omg.org/cgi-bin/apps/doc?ptc/08-06-08.pdf (26.04.2009).

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Doktorarbeit, University of Illinois at Urbana-Champaign, 1992.

[OSG07] OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.1*. IOS Press, 2007.

[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 Bookshelf, 2007.

[Pau09] Miguel Paulos Nunes. MDA Transformationstechniken. Masterarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2009.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Krüger, Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. In *Future of Software Engineering (FOSA) 2007*. IEEE Computer Society, 2007.

[PBL05] Klaus Pohl, Günter Böckle, Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[PJ98] Jens Palsberg, C. Barry Jay. The Essence of the Visitor Pattern. In *Proc. of International Conference Computer Software and Applications (COMPSAC) 1998*. IEEE, 1998.

[PJ07a] Markus Pizka, Elmar Jürgens. Automating Language Evolution. In *Proc. of Symposium on Theoretical Aspects of Software Engineering (TASE) 2007*. IEEE Computer Society, 2007.

[PJ07b] Markus Pizka, Elmar Jürgens. Tool-Supported Multi-Level Language Evolution. In *Proc. of Software and Services Variability Management Workshop - Concepts, Models and Tools 2007*, 2007.

[Plo81] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technischer Bericht DAIMI FN-19, University of Aarhus, 1981.

[POB99] Richard Paige, Jonathan Ostroff, Phillip Brooke. Principles for Modeling Language Design. Technischer Bericht CS-1999-08, York University, 1999.

[PQ95] Terence Parr, Russell Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Journal of Software Practice and Experience*, 25(7):789–810, 1995.

[Pur72] Paul Purdom. A Sentence Generator For Testing Parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[RB08] Andreas Rausch, Manfred Broy. *Das V-Modell XT : Grundlagen, Erfahrungen und Werkzeuge*. dpunkt-Verlag, 2008.

[Ren07] Holger Rendel. Generierung von Beispielinstanzen unter Berücksichtigung von Kontextbedingungen. Seminararbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2007.

[RFH+05] Sabine Rittmann, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl, Doris Wild. Integrating Service Specifications at Different Levels of Abstraction. In *Proc. of the IEEE International Workshop on Service Oriented Software Engineering (SOSE) 2005*. IEEE Computer Society, 2005.

[RMHV06] Damijan Rebernak, Marjan Mernik, Pedro Rangel Henriques, Maria João Varanda. AspectLISA: An Aspect-Oriented Compiler Construction System Based On Attribute Grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):37–53, 2006.

[Roy70] Winston Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Technical Papers of Western Electronic Show and Convention (WesCon) 1970*, 1970.

[RPKP08] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, Fiona Polack. Constructing Models with the Human-Usable Textual Notation. In *Proc. of International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2008*, LNCS 5301. Springer, 2008.

[RRSS08] Dirk Reiss, Bernhard Rumpe, Marvin Schulze-Quester, Mark Stein. Evolving and Implanting Web-Based E-Government-Systems in Universities. In *Proc. of International United Information Systems Conference (UNISCON) 2008*, LNBIP 5. Springer, 2008.

[RSA] IBM Rational Software Architect
`http://www-01.ibm.com/software/awdtools/architect/swarchitect/`.
(26.04.2009).

[RT84] Thomas Reps, Tim Teitelbaum. The Synthesizer Generator. *SIGSOFT Softw. Eng. Notes*, 9:42–48, 1984.

[RT89] Thomas W. Reps, Tim Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer, 3. Auflage, 1989.

[Rum04a] Bernhard Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer, 2004.

[Rum04b] Bernhard Rumpe. *Modellierung mit UML: Sprache, Konzepte und Methodik*. Springer, 2004.

[RVWL03] Grigore Rosu, Ram Venkatesan, Jon Whittle, Laurentiu Leustean. Certifying Optimality of State Estimation Programs. In *Proc. of Computer Aided Verification (CAV) 2003*, LNCS 2725. Springer, 2003.

[SAE06] Society of Automobile Engineers. SAE Architecture Analysis and Design Language (AADL), SAE Standard AS5506/1, 2006.

[Sah06] Henning Sahlbach. Testfallentwicklung für generative Software. Studienarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[SC89] Daniel J. Salomon, Gordon V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. *SIGPLAN Not.*, 24(7):170–178, 1989.

[Sca] Scala Website `http://www.scala-lang.org/`. (26.04.2009).

[SCC06] Charles Simonyi, Magnus Christerson, Shane Clifford. Intentional Software. In *Proc. of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2006*. ACM, 2006.

[Sch90] Andy Schürr. PROGRES: A VHL-Language Based on Graph Grammars. In *Proc. of Graph-Grammars and Their Application to Computer Science 1990*, LNCS 532, 1990.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *Proc. of International Workshop on Graph-Theoretic Concepts in Computer Science (WG) 1994*, LNCS 903. Springer, 1994.

[Sch02] August-Wilhelm Scheer. *ARIS. Vom Geschäftsprozeß zum Anwendungssystem*. Springer, 4. Auflage, 2002.

[SE06] Mehrdad Sabetzadeh, Steve Easterbrook. View Merging in the Presence of Incompleteness and Inconsistency. *Requir. Eng.*, 11(3):174–193, 2006.

[SGW94] Bran Selic, Garth Gullekson, Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, April 1994.

[Sim] Mathworks Matlab Simulink Website,
`http://www.mathworks.com/products/simulink/`. (26.04.2009).

[Sim05] Anthony J. H. Simons. The Theory of Classification, Part 17: Multiple Inheritance and the Resolution of Inheritance Conflicts. *Journal of Object Technology*, 4(2):15–26, 2005.

[Sne91] Gregor Snelting. The Calculus of Context Relations. *Acta Informatica*, 28(5):411–445, 1991.

[SQL96] International Organization for Standardization. ISO/IEC 9075-14:2006, 2006.

[SS71] Dana Scott, Christopher Strachey. Toward a Mathematical Semantics for Computer Languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[Stü06] Ingo Stürmer. *Systematic Testing of Code Generation Tools - A Test Suite-oriented Approach for Safeguarding Model-based Code Generation*. Doktorarbeit, Technische Universität Berlin, 2006.

[Ste08] Jens Stephani. Erweiterung von MontiCore zur Verwendung von Attributen. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

[The06] Jens Theeß. MDA Transformationstechniken. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[Tho97] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1997.

[TLA06] Niels Taatgen, Christian Lebiere, John Anderson. Modeling Paradigms in ACT-R. In *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*. Cambridge University Press, 2006.

[TOP] Topcased Website `http://www.topcased.org/`. (26.04.2009).

[Tor06] Christoph Torens. Konstruktion von Grammatiken aus exemplarischen Dokumenten. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[TR81] Tim Teitelbaum, Thomas Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM*, 24:563–573, 1981.

[Tra06] Laurence Tratt. The MT Model Transformation Language. In *Proc. of Symposium on Applied Computing (SAC) 2006*. ACM, 2006.

[Tra08] Laurence Tratt. Domain Specific Language Implementation via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):1–40, 2008.

[Trac] Trac Website `trac.edgewall.org/`. (26.04.2009).

[VBKG07] Eric Van Wyk, Derek Bodin, Lijesh Krishnan, Jimin Gao. Silver: An Extensible Attribute Grammar System. In *Proc. of Workshop on Language Descriptions, Tools, and Analysis (LDTA) 2007*, ENTCS 203. Elsevier, 2007.

[Vel] Velocity Website `http://velocity.apache.org/`. (26.04.2009).

[Ver01] IEEE Verilog Std 1364-2001, 2001.

[Vis97] Eelco Visser. Scannerless Generalized-LR Parsing. Technischer Bericht P9707, University of Amsterdam, 1997.

[Vis01] Joost Visser. Visitor Combination and Traversal Control. In *Proc. of Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2001*. ACM, 2001.

[Vis03] Joost Visser. *Generic Traversal over Typed Source Code Representations*. Doktorarbeit, University of Amsterdam, February 2003.

[VMo97] Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell, 1997. Allgemeiner Umdruck Nr. 250/1, BWB IT I5.

[Vog93] Harald H. Vogt. *Higher Order Attribute Grammars*. Doktorarbeit, University of Utrecht, 1993.

[VSK89] Harald H. Vogt, S. Doaitse Swierstra, Matthijs F. Kuiper. Higher Order Attribute Grammars. In *Proc. of Conference on Programming Language Design and Implementation (PLDI) 1989*. ACM, 1989.

[Wat00] David Watt. *Programming Language Processors in Java*. Prentice Hall, 2000.

[WFH+06] Doris Wild, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl, Sabine Rittmann. An Architecture-Centric Approach Towards the Construction of Dependable Automotive Software. In *Proc. of the SAE World Congress 2006*, 2006.

[WH95] Luke Wildman, Ian Hayes. Composing Grammar Transformations to Construct a Specification of a Parser. Technischer Bericht TR95-3, Software Verification Research Centre, University of Queensland, 1995.

[Wil94] David S. Wile. POPART: Producer of Parsers and Related Tools. Technischer Bericht, USC Information Sciences Institute, 1994.

[Wil97a] David S. Wile. Abstract Syntax from Concrete Syntax. In *Proc. of International Conference on Software Engineering (ICSE) 1997*. ACM, 1997.

[Wil97b] David S. Wile. Toward a Calculus for Abstract Syntax Trees. In *Proc. of the IFIP TC 2 WG 2.1 International Workshop on Algorithmic Languages and Calculi*. Chapman & Hall, Ltd., 1997.

[Wil01] David S. Wile. Supporting the DSL Spectrum. *Computing and Information Technology*, 4:263–287, 2001.

[Wil03] David S. Wile. Lessons Learned from Real DSL Experiments. In *Proc. of the Hawaii International Conference on System Sciences (HICSS) 2003*. IEEE Computer Society, 2003.

[Wil04] David S. Wile. Lessons Learned from Real DSL Experiments. *Science of Computer Programming*, 51:265–290, 2004.

[Win02] Shuly Wintner. Modular Context-Free Grammars. *Grammars*, 5(1):41–63, 2002.

[Wir74] Niklaus Wirth. On the Design of Programming Languages. In *Proc. of IFIP Congress*, 1974.

[Wit07] Steffen Witt. Entwurf und Implementierung einer erweiterbaren Qualitätssicherungsschnittstelle für Codierungsrichtlinien im automobilen Forschungsumfeld für das Framework MontiCore am Beispiel der Sprache C++. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2007.

[WK05] Manuel Wimmer, Gerhard Kramler. Bridging Grammarware and Modelware. In *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005*, LNCS 3844. Springer, 2005.

[WK06] Jos Warmer, Anneke Kleppe. Building a Flexible Software Factory Using Partial Domain Specific Models. In *Proc. of OOPSLA Workshop on Domain-Specific Modeling 2006*. University of Jyväskylä, 2006.

[WL99] David M. Weiss, Chi T. R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[WMBK02] Eric van Wyk, Oege de Moor, Kevin Backhouse, Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *Proc. of International Conference on Compiler Construction (CC) 2002*, LNCS 2304. Springer, 2002.

[WS07] Eric R. Van Wyk, August C. Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. In *Proc. of International Conference on Generative Programming and Component Engineering (GPCE) 2007*. ACM, 2007.

[WS08] Ingo Weisemöller, Andy Schürr. Formal Definition of MOF 2.0 Metamodel Components and Composition. In *Proc. of International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2008*, LNCS 5301. LNCS, 2008.

[WW93] Pei-Chi Wu, Feng-Jian Wang. Applying Classification and Inheritance into Compiling. *SIGPLAN OOPS Mess.*, 4(4):33–43, 1993.

[XML] Document Object Model (DOM) Level 1 Specification `http://www.w3.org/TR/REC-DOM-Level-1/`. (26.04.2009).

Teil V

Anhänge

Anhang A

Glossar

Die folgenden Definitionen sind bei der Arbeit am MontiCore-Framework entstanden und werden so in der Arbeitsgruppe verwendet. Soweit diese nicht spezifisch für MontiCore sind, wurde sich an gängigen Definitionen in der Literatur orientiert und auf diese verwiesen.

Agile Entwicklung
: Unter dem Begriff agile Entwicklung werden verschiedene Softwareentwicklungsmethoden zusammengefasst, die als Ziel haben, auf eine effiziente Art und Weise eine qualitativ hochwertige Software zu erstellen. Die Kernprinzipien der Methoden sind im agilen Manifest [BBB+] wie folgt zusammengefasst: „Individuals and interactions over processes and tools, Working software over comprehensive documentation, Customer collaboration over contract negotiation, Responding to change over following a plan“. Zentral ist dabei die Forderung, dass das →Produkt mit seinen Eigenschaften im Zentrum stehen soll und dass der Erfolg eines Projekts nicht durch die Einhaltung des Prozesses beurteilt werden kann. Wichtig ist ebenso das Erkennen der Notwendigkeit, auf sich ändernde Anforderungen flexibel zu reagieren.

Abstrakter Syntaxbaum (AST)
: Der abstrakte Syntaxbaum (engl. abstract syntax tree - AST) bezeichnet eine hierarchische Objektstruktur, die das Ergebnis der Syntaxanalyse eines →Satzes ist. Innerhalb des AST können weitere Links existieren, die den aufspannenden Baum zu einem Graphen erweitern. Dann ist auch die Bezeichnung abstrakter Syntaxgraph (engl. abstract syntax graph - ASG) gebräuchlich.

AST-Klassen
: Die AST-Klassen bezeichnen eine objektorientierte Klassenstruktur, deren Klassen zum →AST instanziert werden.

Artefakt
: Der Begriff Artefakt fasst alle Dokumente zusammen, die bei der Softwareerstellung eine Rolle spielen. Beispiele hierfür sind →Modelle, Quellcode und Anforderungsdokumente.

Abstrakte Syntax
: Die abstrakte Syntax einer Sprache beschreibt die strukturelle Essenz einer Sprache [DKLM84, Wil97a]. Sie besteht aus der zentralen Datenstruktur (den →AST-Klassen), auf der weitere Verarbeitungsschritte wie Codegenerierungen, Analysen und Transformationen aufbauen, und wird im Zusammenhang mit →DSLs auch Domänenmodell genannt.

Bibliotheksentwickler
Ein Bibliotheksentwickler entwirft Software-Komponenten oder Bibliotheken innerhalb der Implementierung der →DSL und vereinfacht daher den →Codegenerator, weil konstante wieder verwendbare Softwareteile nicht generiert werden müssen. Daher ist diese Rolle eng verwandt mit dem →Werkzeugentwickler, erfordert aber primär Wissen über das Anwendungsgebiet. Ein Ziel solcher Bibliotheken ist die Kapselung detaillierten Domänenwissens und die Konzeption von Schnittstellen, die ausreichend für die Bedürfnisse einer →Codegenerierung sind.

Bootstrapping
Bootstrapping bezeichnet den Prozess, einen →Generator mit einer früheren Version desselben →Generators teilweise oder ganz zu erzeugen.

Codegenerator
Ein Codegenerator bezeichnet eine spezielle Form eines →Generators, der →Modelle in Elemente einer →Hostsprache übersetzt.

Compiler
Ein Compiler bezeichnet eine spezielle Form eines →Generators, der Quellcode in eine ausführbare Form übersetzt.

Domäne
Der Begriff Domäne bezeichnet ein Anwendungsgebiet der Informatik, das sich durch spezifische →Modelle beschreiben lässt. Dabei kann zwischen technischen Domänen, wie der Struktur eines Softwaresystems oder der Datenbankanbindung, und fachlichen Domänen, wie Software für Automobile oder die Finanzwirtschaft, unterschieden werden.

Domänenexperte
Ein Domänenexperte legt innerhalb der Analyse der →DSL-Entwicklung die Ausgestaltung der →DSL fest. Dabei leistet er wichtige Vorarbeit für die weitere →DSL-Entwicklung, indem er vorhandene Notationen der Domäne zusammenträgt und existierende Frameworks und Bibliotheken der →Domäne auf ihre Eignung prüft.

Domänenspezifische Sprache (DSL)
Eine domänenspezifische Sprache (engl. domain specific language - DSL) bezeichnet eine Programmier- oder Modellierungssprache, die die folgenden Bedingungen erfüllt:

- Die Sprache orientiert sich an einer Anwendungsdomäne, für die sie hauptsächlich verwendet wird, und weniger an der technischen Realisierung.
- Falls die DSL ausführbar ist, ist die Ausdrucksmächtigkeit der Sprache soweit eingeschränkt, wie es für die →Domäne ausreichend ist, was meistens weniger als die universelle Berechenbarkeit ist.
- Eine DSL erlaubt, die Problemstellungen einer →Domäne kompakt zu lösen.

Domänenspezifische Modellierungssprache
Eine domänenspezifische Modellierungssprache (engl. domain specific modeling language - DSML) ist →Modellierungssprache, die spezifisch für eine bestimmte →Domäne ist. Da sich diese Arbeit vor allem auf DSMLs fokussiert und kaum domänenspezifische Programmiersprachen betrachtet, werden die beiden Begriffe →DSL und DSML synonym verwendet und die →Sätze einheitlich als →Modell bezeichnet.

DSL-Entwicklung
Die Entwicklung einer →DSL umfasst neben der Definition der Sprache auch die Realisierung darauf basierender →Codegenerierungen. Die →Codegeneratoren können innerhalb einer →Produktentwicklung zur Übersetzung von Modellen in →Produktcode verwendet werden.

Einbettung
Einbettung bezeichnet eine Form der →modularen Komposition von →Grammatikfragmenten. Dabei werden →externe Produktionen eines Fragments mit →Produktionen eines anderen Fragments oder einer →Sprache verbunden.

Eingebettete DSL
Eingebettete DSLs [BV04, BV07] bezeichnen Spracherweiterungen von GPLs (die in diesem Zusammenhang als →Hostsprache bezeichnet werden), die zusätzliche kompakte Formulierungen bieten, jedoch die Grundstruktur der GPL erhalten. Die Elemente der →DSL werden dabei auf die Konstrukte der GPL wie zum Beispiel API-Aufrufe abgebildet.

Entwicklungszeit
Als Entwicklungszeit wird die Zeitspanne bezeichnet, in der Entwickler →Artefakte in einem Softwareentwicklungsprozess formulieren und durch →Codegeneratoren und →Compiler in eine ausführbare Form übersetzen.

Externe Produktionen
Eine externe Produktion ist ein spezielles →Modellelement der →MontiCore-Grammatik, das einen Erweiterungspunkt darstellt. Die externen Produktionen werden in →Sprachdateien zur Verbindung verschiedener →Grammatikfragmente benutzt.

Generator
Ein Generator überführt eine Sprache in eine andere Repräsentation. →Codegeneratoren und →Compiler sind spezifische Generatoren.

Grammatikfragment
Unter einem Grammatikfragment versteht man eine Grammatik, die unvollständig ist, da sie →externe Produktionen enthalten kann und erst durch →Einbettung zu einer vollständigen →Sprachdefinition wird.

Grammatikvererbung
Die Grammatikvererbung bezeichnet einen Mechanismus zur Spezialisierung einer →MontiCore-Grammatik, bei dem in der spezialisierenden Grammatik nur die Produktionen angegeben werden, die gegenüber der spezialisierten Grammatik verändert oder hinzugefügt werden sollen.

Hostsprache
Unter einer Hostsprache wird eine GPL verstanden, auf die →Codegenerierung aus einer →DSL abgebildet wird (vgl. [BV07]). Oftmals können Teile der Hostsprache in →Modellen einer →DSL verwendet werden, da dieses bei der Generierung unverändert bleiben können.

Komposition
Die Komposition bezeichnet die Operation, die zwei oder mehrere →Artefakte zu einem neuen Artefakt zusammenfasst. Dieses neue Artefakt kann dabei nur konzeptuell vorhanden sein.

Kompositionalität
Das Kompositionalitätsprinzip geht auf Gottlob Frege zurück und besagt, dass die →Semantik eines Kompositums sich nur aus seinen Teilen und der →Komposition ergibt.

Konkrete Syntax
Die konkrete Syntax einer →Sprache definiert, wie →Sätze einer →Sprache zu formulieren sind.

Kontextbedingung
Eine Kontextbedingung bezeichnet eine prüfbare Bedingung, die die Menge der wohldefinierten Modelle einschränkt. Die Kontextbedingungen ergänzen den Formalismus zur Spezifikation der →abstrakten Syntax und der →konkreten Syntax um Einschränkungen, die sich in diesen Formalismen nicht oder nur schwer formulieren lassen. Die Kontextbedingungen sollen ausschließen, dass →Modelle verarbeitet werden, für die die →Semantik keine Bedeutung definiert.

Kontextfreie Syntax
Die kontextfreie Syntax bezeichnet den Teil einer Sprachdefinition, der die Inhalte der →Syntaxanalyse festlegt.

Konzept
Ein Konzept bezeichnet eine Erweiterung der →MontiCore-Grammatik, die die Grundlage für spezifische Codegenerierungen bildet. Die notwendigen Informationen werden innerhalb einer →MontiCore-Grammatik nach dem Schlüsselwort concept und dem Namen des Konzepts aufgeführt.

Laufzeit
Als Laufzeit wird die Zeitspanne bezeichnet, in der ein →Produkt ausgeführt wird.

Laufzeitumgebung
Wird aus einer →DSL →Produktcode erzeugt, dann muss er nicht zwangsläufig eine komplette in sich lauffähige Software sein. Vielmehr ist üblicherweise ein fixer Teil vorgegeben, der den für alle Modelle gleichen und damit vom Modell unabhängigen Code enthält, und ein variabler generierter Teil. Der fixe Teil der Software kann eine Sammlung von Klassen sein, auf die sich der variable Teil durch Unterklassenbildung oder Methodenaufruf bezieht, und wird als →Laufzeitumgebung bezeichnet. Sie wird zur →Laufzeit der →Produkte ausgeführt, gehört jedoch konzeptuell zur →DSL.

Lexer
Ein Lexer bezeichnet einen Softwareteil, der die →lexikalische Analyse innerhalb eines →Generators ausführt.

Lexikalische Analyse
Die lexikalische Analyse ist eine Arbeitsphase innerhalb eines Generators, der die Zeichen eines →Satzes zu Wörtern gruppiert und die Grundlage für die →Syntaxanalyse bildet.

Lexikalische Syntax
Die lexikalische Syntax bezeichnet den Teil einer →Sprachdefinition, die die Inhalte der →lexikalischen Analyse festlegt.

Modell
Ein Modell bezeichnet einen →Satz einer →Modellierungssprache oder →DSL.

Modellelement
Ein Modellelement ist ein Teil eines Modells, das eine in sich konzeptuelle geschlossene Einheit darstellt. Ein Beispiel hierfür ist eine →Produktion innerhalb einer →MontiCore-Grammatik.

Modellgetriebene Entwicklung
Die modellgetriebene Entwicklung bezeichnet eine Entwicklungsmethodik, die Modelle als →primäre Artefakte der Softwareentwicklung verwendet. Dabei werden →Modelle, die in einer →Modellierungssprache beschrieben sind, automatisiert in →Produktcode umgesetzt.

Modellierungssprache
Eine Modellierungssprache zur Modellierung von Software dient zur abstrahierten Beschreibung eines Softwaresystems und erlaubt oftmals die frühzeitige Analyse wichtiger Systemeigenschaften.

Modellklasse
Eine Modellklasse ist eine alternative Bezeichnung für eine →AST-Klasse bei →Modellierungssprachen.

Modelltransformation
Eine Modelltransformation bezeichnet eine Software, die →Modelle einer →Modellierungssprache in →Modelle einer möglicherweise anderen →Modellierungssprache überführt.

Modulare Komposition
Die modulare Komposition bezeichnet eine →Komposition, die zur Definition der →Semantik des Kompositums nicht die gesamten →Modelle benötigt, sondern deren explizit exportierter Anteil (Schnittstelle) ausreichend ist.

Modularitätsmechanismus
Ein Modularitätsmechanismus bezeichnet Elemente einer →Sprache, die dazu dienen, →Sätze der →Sprache auf verschiedene →Artefakte aufteilen zu können. Diese Artefakte können dann durch →Komposition konzeptuell (und nicht zwangsläufig physisch) zusammengeführt werden.

MontiCore-Grammatik
Die MontiCore-Grammatik ist eine →DSL, die die Definition der →konkreten Syntax und der →abstrakten Syntax in einem integrierten Format erlaubt. Die Modelle stellen dabei →Grammatikfragmente dar, wenn →externe Produktionen verwendet werden, und können durch →Sprachdateien zu einer →Sprache kombiniert werden.

Nichtterminal
Eine Nichtterminal ist ein spezielles Element einer Grammatik, das durch eine →Produktion definiert wird und in der →Syntaxanalyse weiter expandiert werden kann (vgl. →Terminal).

Parsefähigkeit
Parsefähigkeit bezeichnet eine Eigenschaft einer →MontiCore-Grammatik, dass sich aus ihr ein →Parser ableiten lässt, ohne dass dabei Mehrdeutigkeiten aufgelöst werden müssen.

Parser
Ein Parser bezeichnet einen Softwareteil, der die →Syntaxanalyse innerhalb eines →Generators ausführt.

Plattform
Eine Plattform bezeichnet eine spezifische Technologie, die als Ziel für eine Codegenerierung aus →DSLs genutzt wird. Innerhalb der →modellgetriebenen Entwicklung können →Modelle ohne spezifische Verweise auf die Plattform erstellt und durch entsprechend parametrisierte →Modelltransformationen automatisiert in Quellcode überführt werden.

Produktion
Eine Produktion bezeichnet die →Modellelemente einer →MontiCore-Grammatik. Dabei gibt es drei Arten von Parserproduktionen: Klassenproduktionen, Schnittstellenproduktionen und abstrakte Produktionen. Diese legen die →kontextfreie Syntax einer →Sprache fest. Zusätzlich gibt es lexikalische Produktionen und Enumerationsproduktionen, die die →lexikalische Syntax bestimmen. Ergänzend gibt es →externe Produktionen.

Primäres Artefakt
Ein primäres Artefakt der Softwareentwicklung beeinflusst direkt die Struktur oder das Verhalten einer Software. Darunter fallen neben Quellcode bei der →modellgetriebenen Entwicklung vor allem auch →Modelle, die durch eine →Codegenerierung in eine ausführbare Form überführt werden (vgl. →Sekundäres Artefakt).

Produkt
Innerhalb der →modellgetriebenen Softwareentwickung wird eine Software, die mit Hilfe von Modellen entwickelt wird, als Produkt bezeichnet. Das Produkt besteht dabei aus dem →Produktcode und möglicherweise einer oder mehreren →Laufzeitumgebungen.

Produktcode
Produktcode bezeichnet den Quellcodeanteil eines →Produkts, der spezifisch für das →Produkt aus Modellen generiert oder manuell programmiert wurde.

Produktentwicklung
Die Produktentwicklung bezeichnet den Prozess zur Entwicklung eines →Produkts. Sie nutzt die in der →DSL-Entwicklung entstandenen →DSLs und darauf basierten →Generatoren.

Programm
Ein Programm bezeichnet einen →Satz einer →Programmiersprache.

Programmiersprache
Eine Programmiersprache ist eine turing-mächtige Sprache, deren →Programme automatisiert durch einen →Compiler in ein ausführbares Softwaresystem überführt werden können.

Semantik
Die Semantik einer Sprache bezeichnet deren Bedeutung. Dabei können verschiedene Techniken wie denotationale Semantik [SS71] und operationale Semantik [Plo81] genutzt werden, die jeweils auf der →abstrakten Syntax aufbauen, ohne direkt Bezug

auf die →konkrete Syntax zu nehmen. Die Semantik besteht dabei aus einer wohlverstandenen semantischen Domäne und einer semantischen Abbildung, die →Sätze auf deren Bedeutung abbildet.

Satz
Ein Satz ist ein Element einer →Sprache.

Sekundäres Artefakt
Ein sekundäres Artefakt der Softwareentwicklung beeinflusst die Struktur oder das Verhalten einer Software nur indirekt. Beispiele hierfür sind textuelle Anforderungen oder →Modelle, die durch einen Entwickler manuell in Quellcode umgesetzt werden (vgl. →Primäres Artefakt).

Sprachdatei
Eine Sprachdatei ist eine →DSL, die mehrere →Grammatikfragmente zu einer vollständigen Sprachdefinition komponiert.

Sprache
Eine Sprache bezeichnet eine Menge von wohldefinierten →Sätzen, die durch eine →Sprachdefinition festgelegt werden.

Sprachdefinition
Eine Sprachdefinition bezeichnet ein oder mehrere →Artefakte, die die →konkrete Syntax und →abstrakte Syntax einer Sprache und die dazu gehörigen →Kontextbedingungen definieren. Ergänzt werden die →Artefakte durch eine Definition der →Semantik.

Sprachentwickler
Ein Sprachentwickler definiert oder erweitert eine →DSL bezüglich der Bedürfnisse und Anforderungen des →Produktentwicklers und den Festlegungen des →Domänenexperten im Entwurf der →DSL.

Symboltabelle
Eine Symboltabelle ist eine meist hierarchisch gegliederte Datenstruktur zum Speichern und Auflösen von Bezeichnern in einer Sprache. Wesentliche Aufgaben bestehen im Auflösen von Namen für Variablen und deren Typen.

Syntaxanalyse
Die Syntaxanalyse ist eine Arbeitsphase innerhalb eines →Generators, in der ein →Parser einen →Satz in einen →AST überführt.

Terminal
Ein Terminal ist ein spezielles Element einer Grammatik, das in der →Syntaxanalyse nicht weiter expandiert werden kann (vgl. →Nichtterminal).

Tokenklasse
Eine Tokenklasse bezeichnet eine Menge an Wörtern, die durch eine lexikalische Produktion definiert sind und durch einen →Parser auf dieselbe Art behandelt werden.

Visitor
Ein Visitor bezeichnet einen Algorithmus, der auf den →AST-Klassen einer Sprache entkoppelt von der Traversierungstrategie definiert wird. Die technische Realisierung ist durch das gleichnamige Entwurfsmuster möglich [GHJV95].

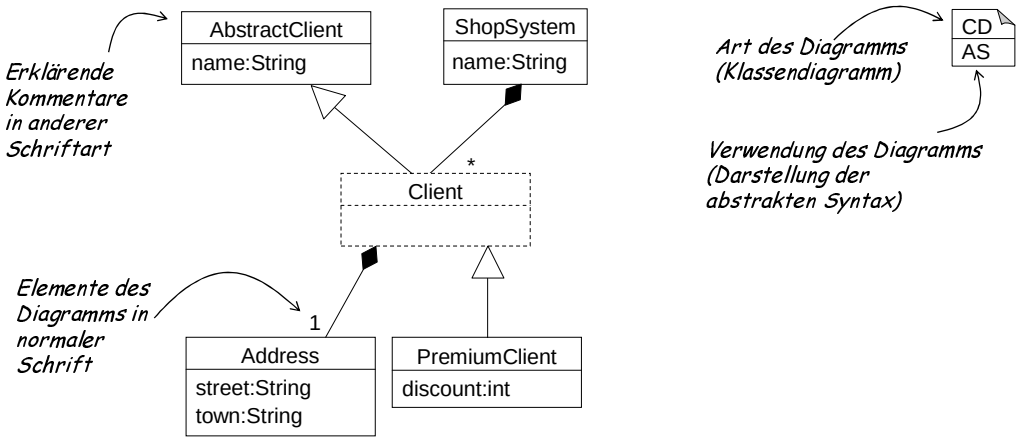
Werkzeugentwickler

Ein Werkzeugentwickler schreibt einen →Codegenerator für die →DSL, der sowohl Produktions- und Testcode als auch Analysen des Inhalts und der Qualität erzeugen kann. Er testet seine Implementierung in der Validierung/Verfikation der →DSL-Entwicklung. Zusätzlich integriert er neu entwickelte oder wieder verwendete Sprachverarbeitungskomponenten und →Generatoren zu neuen Werkzeugen, die bei der Entwicklung der Software eingesetzt werden.

Anhang B

Zeichenerklärungen

In dieser Arbeit sind einige Diagramme durch spezielle Kommentare gekennzeichnet. Dabei wird zwischen der Diagrammart, die die verfügbaren Notationselemente und deren grundsätzliche Bedeutung festlegt, und der Diagrammverwendung unterschieden, falls eine Diagrammart für verschiedene Zwecke verwendet werden kann. Die folgende Abbildung zeigt die verwendete Notation anhand eines Beispiels:



In dieser Arbeit wurden die folgenden Diagrammarten und -verwendung eingesetzt:

Klassendiagramme [OMG07a, OMG07b] (CD)

Abstrakte Syntax einer DSL (AS)
Die Verwendung zeigt die AST-Klassen einer Sprache, ohne die genaue Umsetzung für eine Plattform darzustellen. Dabei werden Pakete gegebenenfalls dazu verwendet, um verschiedene Sprachen darzustellen.

Implementierung (Impl)
Die Verwendung zeigt die implementierten Klassen und Schnittstellen innerhalb des MontiCore-Frameworks. Bei Attributen wird eine Realisierung durch get/set-Methoden angenommen.

Kompositionsstrukturdiagramme [OMG07a, OMG07b] (CSD)

Implementierung (Impl)
Die Verwendung zeigt die implementierten Klassen und Schnittstellen sowie deren hierarchische Dekomposition innerhalb einer Softwarearchitektur.

Komponentendiagramme [OMG07a, OMG07b] (CpD)

MontiCore (MC)
Diese Diagramme zeigen MontiCore-Teilprojekte als ausführbare Komponenten. Die Teilprojekte und deren Verzeichnisorganisation werden als Pakete dargestellt.

Verteilungsdiagramme [OMG07a, OMG07b] (DD)

MontiCore (MC)
Diese Diagramme sind eine Erweiterung der Komponentendiagramme, weil zusätzlich Hardwaresysteme dargestellt werden, auf denen die verschiedenen Komponenten ausgeführt werden.

Objektdiagramme [OMG07a, OMG07b] (OD)

AST einer DSL (AST)
Die Verwendung zeigt den AST eines Modells, wobei die Klassen und Schnittstellen aus den plattformunabhängigen Klassendiagrammen zur Darstellung der abstrakten Syntax verwendet werden.

Implementierung (Impl)
Die Verwendung zeigt den AST eines Modells, wobei direkt die Klassen und Schnittstellen einer Plattform verwendet werden.

Aktivitätsdiagramme [OMG07a, OMG07b] (AD)

Methodik (Meth)
Es werden (teilweise automatisierte) Arbeitsabläufe als Aktivitäten zur Illustration einer Methodik dargestellt.

Tombstone-Diagramme [Bra61, ES70] (T)

Die Diagramme werden verwendet, um die Entwicklung von Generatoren gemäß den Konventionen des Compilerbaus darzustellen.

Anhang C

Abkürzungen

AADL	Architecture Analysis and Design Language
ACT-R	Adaptive Control of Thought-Rational
AML	Automotive Modeling Language
API	Application Programming Interface
ARMS	Architektur-Referenzmodellsprache
ASF	Algebraic Specification Formalism
ASG	Abstract Syntax Graph
AST	Abstract Syntax Tree
ATESST	Advancing Traffic Efficiency and Safety through Software Technology
ATL	Atlas Transformation Language
AUTOSAR	Automotive Open System Architecture
BNF	Backus-Naur-Form
CIP	Computer-Aided, Intuition-Guided Programming
COM	Component Object Model
CPU	Central Processing Unit
CTS	Common Type System
DARE	Domain Analysis and Reuse Environment
DCO	Delegating Compiler Objects
DOM	Document Object Model
DSL	Domain Specific Language (Domänenspezifische Sprache)
DSML	Domain Specific Modeling Language
EAST-ADL	Electronics Architecture and Software Technologies - Architecture Description Language
EAST-EEA	Electronics Architecture and Software Technologies - Embedded Electronic Architecture
EBNF	Extended Backus-Naur-Form
EMF	Eclispe Modeling Framework
EOF	End Of File
FAST	Family-Oriented Abstractions, Specifications, and Translation
FODA	Feature-Oriented Domain Analysis
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
GONF	Generalized Object Normal Form

GPL	General Public License
GUI	Graphical User Interface
HTML	HyperText Markup Language
HUTN	Human-Usable Textual Notation
IBD	Internes Blockdiagramm
IDE	Integrated Development Environment
IPSEN	Integrated Project Support ENvironment
IT	Informationstechnologie
ITEA	Information Technology for European Advancement
JavaEE	Java Enterprise Edition
JDK	Java Development Kit
MC	MontiCore
MIP	ModelInfrastructureProvider
MOF	Meta Object Facility
MOP	Meta Object Protocol
MPS	Meta Programming System
MT	Model Transformation
OAW	OpenArchitectureWare
OCL	Object Constraint Language
OEM	Original Equipment Manufacturer
OMG	Object Management Group
ONF	Object-Oriented Normal Form
OO	Object Oriented
OSTP	Online Software Transformation Platform
PC	Personal Computer
POJO	Plain Old Java Objects
PROGRES	PROgrammed Graph REwriting Systems
RAG	Reference Attributed Grammars
RE	Requirements Engineering
SAE	Society of Automobile Engineers
SDF	Syntax Definition Formalism
SLOC	Source Lines Of Code
SQL	Structured Query Language
SW	Software
TTCN-3	Testing and Test Control Notation Version 3
UML	Unified Modeling Language
UML/P	Unified Modeling Language / Programmier-geeignet
UML-RT	Unified Modeling Language - RealTime
VM	Virtual Machine
XML	Extensible Markup Language

Anhang D

Grammatiken

Dieser Anhang enthält jeweils für die MontiCore-Grammatik und die Sprachdateien die in den Kapiteln 3 und 4 verwendeten vollständige Definition der Sprachen als MontiCore-Grammatik. Zusätzlich wird die in Kapitel 5 definierte Essenz als MontiCore-Grammatik angegeben und die Unterschiede zur vollständigen Version erklärt.

D.1 MontiCore-Grammatik

D.1.1 Vollständige MontiCore-Grammatik

Die Abbildungen D.1-D.8 zeigen die MontiCore-Grammatik in ihrem eigenem Format. Zu Gunsten einer kompakten Notation wurden die Kommentare und die Berechung der Attribute entfernt.

MontiCore-Grammatik

```
1  package mc.grammar;
2
3  grammar Grammar  {
4
5    options {
6      parser lookahead=2   lexer lookahead=3
7      nostring
8      compilationunit Grammar
9    }
10
11   external Action;
12
13   external MCConcept;
14
15   Grammar =
16     "grammar" name:IDENT
17       ("extends" supergrammar:GrammarReference ("," supergrammar:GrammarReference)* )?
18       "{"
19          ( GrammarOption | LexProd | ClassProd | EnumProd | ExternalProd |
20            InterfaceProd | AbstractProd | ASTRule | AssociationBlock |
21            Association | Concept )*
22       "}" ;
```

Abbildung D.1: MontiCore-Grammatik (1 / 9)

MontiCore-Grammatik

```
23  ast Grammar =
24    grammarOption:GrammarOption max=1 ;
25
26  GrammarReference =
27    name:IDENT& ("." name:IDENT& )*  ;
28
29  GrammarOption=
30    "options" "{"
31      ( ParserOption | LexerOption | HeaderOption | VariousOption |
32        PrettyPrinterOption | CompilationUnitOption | KeywordOption |
33        LexerRuleOption | FollowOption | GenericOption
34      )*
35    "}" ;
36
37  ast GrammarOption =
38    parserOption:ParserOption max = 1
39    lexerOption:LexerOption max = 1
40    headerOption:HeaderOption max = 1
41    prettyPrinterOption:PrettyPrinterOption max = 1
42    compilationUnitOption:CompilationUnitOption max = 1
43    keywordOption:KeywordOption max = 1
44    lexerRuleOption:LexerRuleOption max = 1 ;
45
46  ParserOption =
47    "parser" ("lookahead"  "=" Lookahead:NUMBER)? (parserOptions:STRING)?;
48
49  LexerOption=
50    "lexer" ("lookahead" "=" Lookahead:NUMBER)? (lexerOptions:STRING)?;
51
52  HeaderOption=
53    "header" headerOptions:STRING;
54
55  VariousOption=
56    (options {greedy=true;} :
57      ["nows"] | ["noslcomments"] | ["nomlcomments"] | ["noanything"]
58      | ["noident"] | ["dotident"] | ["nostring"] | ["nocharvocabulary"]
59    )+ ;
60
61  PrettyPrinterOption=
62    "prettyprinter" namePrettyPrinter:IDENT ("." namePrettyPrinter:IDENT)*;
63
64  CompilationUnitOption =
65    "compilationunit" type:IDENT ;
66
67  KeywordOption =
68    "keywords" "{" Keywords:STRING ("," Keywords:STRING)* "}";
69
70  LexerRuleOption =
71    "lexerrules" "{" lexerRules:STRING ("," lexerRules:STRING)*  "}";
72
73  FollowOption =
74    "follow" prodName:IDENT Alt ";";
75
76  GenericOption =
77    name:IDENT  ("=" value:IDENT |"=" value:STRING)? ;
```

Abbildung D.2: MontiCore-Grammatik (2 / 9)

MontiCore-Grammatik

```
78  interface Prod;
79
80  ast Prod =
81    Name:IDENT ;
82
83  interface ParserProd extends Prod;
84
85  ast ParserProd =
86    external:/boolean;
87
88  LexProd implements Prod =
89    (Protected:[Protected:"protected"] | Comment:[Comment:"comment"])*
90    "token" name:IDENT
91    (
92      LexOption ("{" InitAction:Action "}")?
93    |
94      "{" InitAction:Action "}"
95    )?
96    "=" Alts:LexAlt ("|" Alts:LexAlt)*
97    (":" Variable:IDENT
98      ("->" Type:IDENT ("." Type:IDENT )* (":" "{" Block:Action "}" )? )?
99    )?
100   ";";
101
102 EnumProd implements Prod =
103   "enum" (External:["/"])? name:IDENT "=" Constant ("|" Constant)* ";";
104
105 ExternalProd implements Prod =
106   "external" name:IDENT ExternalType? ";";
107
108 InterfaceProd =
109   "interface" (External:["/"])? name:IDENT
110   (
111     "[" Parameter ("," Parameter)* "]"
112   |
113     "extends" superInterfaceRule:RuleReferenceWithPredicates
114     ("," superInterfaceRule:RuleReferenceWithPredicates)*
115   |
116     "astextends" aSTSuperInterface:TypeReference
117     ("," aSTSuperInterface:TypeReference)*
118   )*
119   ( overrideStandard:["="]
120     ( interfaces:RuleReferenceWithPredicates
121       ("|" interfaces:RuleReferenceWithPredicates)*
122     )?
123   )?
124   ";";
125
126 AbstractProd implements Prod =
127   "abstract" (External:["/"])? name:IDENT
128   (
129     "[" Parameter ("," Parameter)* "]"
130   |
131     "extends" superRule:RuleReferenceWithPredicates
132     ("," superRule:RuleReferenceWithPredicates)*
```

Abbildung D.3: MontiCore-Grammatik (3 / 9)

MontiCore-Grammatik

```
133     |
134       "implements" superInterfaceRule:RuleReferenceWithPredicates
135       ("," superInterfaceRule:RuleReferenceWithPredicates)*
136     |
137       "astextends" aSTSuperClass:TypeReference
138       ("," aSTSuperClass:RuleReferenceWithPredicates)*
139     |
140       "astimplements" aSTSuperInterface:TypeReference
141       ("," aSTSuperInterface:TypeReference)*
142     )*
143     ( overrideStandard:["="]
144       ( interfaces:RuleReferenceWithPredicates
145         ("|" interfaces:RuleReferenceWithPredicates)*
146       )?
147     )? ";";
148
149   ClassProd implements ParserProd =
150     (External:["/"])? name:IDENT (":" type:IDENT)?
151     (
152        "extends" superRule:RuleReferenceWithPredicates
153        ("," superRule:RuleReferenceWithPredicates)*
154     |
155        "implements" superInterfaceRule:RuleReferenceWithPredicates
156        ("," superInterfaceRule:RuleReferenceWithPredicates)*
157     |
158        "astextends" aSTSuperClass:TypeReference ("," aSTSuperClass:TypeReference)*
159     |
160        "astimplements" aSTSuperInterface:TypeReference
161        ("," aSTSuperInterface:TypeReference)*
162     |
163        ("[|") Parameter ("," Parameter)* ("|]")
164     )*
165     ("returns" returnType:IDENT)?
166     ("{" Action "}")?
167     ("=" alts:Alt ("|" alts:Alt)* )? ";";
168
169   ast ClassProd =
170     method public String toString() { return type; } ;
171
172   Parameter =
173     (
174       name:IDENT prodElementType:["="|":"] reference:IDENT
175     |
176       reference:IDENT prodElementType:["!"]
177     )
178     Card? ;
179
180   Card =
181    (
182      Unbounded:["*"]
183    |
184      "min" "=" Min:NUMBER ("max" "=" ( Max:NUMBER | Max:"*"))?
185    |
186      "max" "=" ( Max:NUMBER | Max:"*")
187    );
```

Abbildung D.4: MontiCore-Grammatik (4 / 9)

MontiCore-Grammatik

```
188 interface TypeReference;
189
190 ast TypeReference =
191   method public String getTypeName() {}
192   method public boolean isExternal() {};
193
194 ExternalType implements ("/")=> TypeReference =
195   "/" GenericType ;
196
197 ast ExternalType =
198   method public String getTypeName() {
199     return "/"+ getGenericType().toString();
200   }
201   method public boolean isExternal() {
202     return true;
203   };
204
205 RuleReference implements TypeReference =
206   Name:IDENT ;
207
208 RuleReferenceWithPredicates:RuleReference =
209   ("{" Action "}" | Predicate:Block)? Name:IDENT ;
210
211 ast RuleReference =
212   method public String getTypeName() {
213     return getName();
214   }
215   method public boolean isExternal() {
216       return false;
217   };
218
219 Alt =
220   Components:RuleComponent* ;
221
222 interface RuleComponent;
223
224 Block  implements ("(")=>RuleComponent=
225   "(" (
226     (
227       Option  ( "init" "{" InitAction:Action "}" )?
228     |
229       "init" "{" InitAction:Action "}"
230     )
231     ":" )?
232   Alts:Alt ("|" Alts:Alt)* ")"
233   (Iteration:["?"|"*"|"+"|Predicate:"=>"])?;
234
235 Option =
236   "options" "{" OptionValue+ "}";
237
238 OptionValue =
239   key:IDENT "=" value:IDENT ";" ;
240
241 NonTerminal implements ( (IDENT ("="|":"))? IDENT )=> RuleComponent =
242   (variableName:IDENT "=" | usageName:IDENT  ":")?
```

Abbildung D.5: MontiCore-Grammatik (5 / 9)

MontiCore-Grammatik

```
243     name:IDENT
244     "<" (parserParameter:[PARSER:"global"])? Parameter:IDENT ">"
245     (
246      "[|" ruleParameters:IDENT ("," ruleParameters:IDENT )* "|]"
247     )?
248     (ignore:["!"] | plusKeywords:["&"] | iteration:["?"|"*"|"+"] )*;
249
250   Terminal  implements ( (IDENT ("="|":"))? STRING )=> RuleComponent,
251                        (IDENT ("="|":"))=> ITerminal =
252     (variableName:IDENT "=" | usageName:IDENT  ":")?
253     name:STRING
254     (ignore:["!"] | iteration:["?"|"*"|"+"])*;
255
256   Constant implements ITerminal=
257     (humanName:IDENT ":")? name:STRING ;
258
259   interface ITerminal;
260
261   ast ITerminal =
262       Name:IDENT;
263
264   ConstantGroup implements ( (IDENT ("="|":"))? "[" )=> RuleComponent =
265      (variableName:IDENT "=" | usageName:IDENT  ":")?
266      "[" Constant ("|" Constant )* "]"
267      (Iteration:["?"|"*"|"+"])? ;
268
269   Eof implements RuleComponent =
270     "EOF";
271
272   MCAnything implements RuleComponent =
273     "MCA";
274
275   Anything implements RuleComponent=
276     ".";
277
278   Sematicprecicateoraction implements RuleComponent=
279     (Name:"astscript" {tmpemb_name="astscript";} )?
280     "{" Text:Action(parameter Name) "}" (Predicate:["?"])?;
281
282   Concept=
283     "concept" name:IDENT Concept:MCConcept<Name>;
284
285   ASTRule =
286      "ast" type:IDENT
287     (
288       "astextends" aSTSuperClass:TypeReference
289       ("," aSTSuperClass:TypeReference)*
290     |
291       "astimplements" aSTSuperInterface:TypeReference
292       ("," aSTSuperInterface:TypeReference)*
293     )*
294     (
295       "="
296       ( Method | AttributeInAST)*
297     )? ";";
```

Abbildung D.6: MontiCore-Grammatik (6 / 9)

MontiCore-Grammatik

```
298  Method=
299      "method"
300      (options {greedy=true;}: ["public"] | ["private"] | ["protected"])?
301      (options {greedy=true;}: ["final"])?
302      (options {greedy=true;}: ["static"])?
303      return:GenericType name:IDENT  "(" ( MethodParameter ("," MethodParameter)*)?  ")"
304      ("throws" exceptions:GenericType ("," exceptions:GenericType)*)?
305      "{" body:Action "}" ;
306
307  MethodParameter =
308    type:GenericType name:IDENT;
309
310  AttributeInAST =
311    (name:IDENT  ":")?
312    TypeReference ( options {greedy=true;}: Iterated:["*"])?
313    Card?
314    (Derived:["derived"] "{" Body:Action "}")? ;
315
316  GenericType  =
317    Name:IDENT& (options {greedy=true;}: "." Name:IDENT& )*
318    ("<" GenericType ("," GenericType)*  ">")?
319    {a.setDimension(0);}
320    (options {greedy=true;}: "[" "]" {a.setDimension(a.getDimension()+1);} )*;
321
322  ast GenericType =
323    Dimension:/int
324    method public String toString() {
325      return  mc.mcgrammar.HelperGrammar.printGenericType(this);
326    };
327
328  AssociationBlock =
329    "associations" "{" (Association:AssociationInBlock)* "}";
330
331  AssociationInBlock:Association =
332     (name:IDENT)?
333     left:LeftAssociationEnd
334     direction:["<-"|"->"|"<->"|"--"]
335     right:RightAssociationEnd ";" ;
336
337  Association =
338     "association"
339     (name:IDENT)?
340     left:LeftAssociationEnd
341     direction:["<-"|"->"|"<->"|"--"]
342     right:RightAssociationEnd ";" ;
343
344  RightAssociationEnd:AssociationEnd =
345   (cardMin:NUMBER "..")?
346   (cardMaxOrMaxAndMin:NUMBER | cardMaxOrMaxAndMin:"*")?
347   typeAndName:IDENT ("." typeAndName:IDENT  )* ;
348
349  LeftAssociationEnd:AssociationEnd =
350   typeAndName:IDENT ("." typeAndName:IDENT  )*
351   (cardMin:NUMBER "..")?
352   (cardMaxOrMaxAndMin:NUMBER | cardMaxOrMaxAndMin:"*")?  ;
```

Abbildung D.7: MontiCore-Grammatik (7 / 9)

MontiCore-Grammatik

```
353  LexAlt =
354    (Ignore:["!"])? (LexComponents:LexComponent)*;
355
356  interface LexComponent;
357
358  LexBlock implements (("~")? "(" )=> LexComponent =
359    (Negate:["~"])? "("
360    (
361      (
362        Option:LexOption ("init" "{" InitAction:Action "}")?
363      |
364        "init" "{" InitAction:Action "}"
365      )
366      ":"
367     )?
368     LexAlts:LexAlt ( "|" LexAlts:LexAlt )* ")"
369     (Iteration:["?"|"*"|"+"|Predicate:"=>"])?;
370
371  LexCharRange implements (("~")? CHAR "..")=> LexComponent =
372     (Negate:["~"])? LowerChar:CHAR ".." UpperChar:CHAR;
373
374  LexChar implements (("~")? CHAR )=> LexComponent =
375    (Negate:["~"])? Char:CHAR (Ignore:["!"])? ;
376
377  LexString implements LexComponent =
378    String:STRING (Ignore:["!"])? ;
379
380  LexActionOrPredicate implements LexComponent =
381    (Name:"astscript")? "{" Text:Action(parameter Name) "}" (Predicate:["?"])?;
382
383  LexNonTerminal implements LexComponent =
384    (Variable:IDENT "=")? Name:IDENT ;
385
386  LexOption =
387    "options" "{" ID:IDENT "=" Value:IDENT ";" "}";
388
389  token CHAR =
390    '\''! (ESC|~'\'') '\''!;
391
392  token STRING =
393    '"'! (ESC|~'"')* '"'!;
394
395  protected token  ESC  =
396    '\\'
397    (  'n' | 'r' | 't'  | 'b' |  'f'  | 'w' |  'a'  | '\"' | '\'' | '\\'  | ('0'..'3')
398    ( options { warnWhenFollowAmbig = false; }: '0'..'7'
399    ( options { warnWhenFollowAmbig = false; }: '0'..'7')?  )?
400    |  ('4'..'7')  ( options { warnWhenFollowAmbig = false; }: '0'..'7' )?
401    |  'u' XDIGIT XDIGIT XDIGIT XDIGIT
402    )
403  ;
404
405  protected token DIGIT =
406    '0'..'9';
407
```

Abbildung D.8: MontiCore-Grammatik (8 / 9)

MontiCore-Grammatik

```
408   protected token XDIGIT =
409     '0' .. '9'  |  'a' .. 'f'  |  'A' .. 'F';
410
411   token NUMBER =
412     ( '0'..'9' )+;
413 }
```

Abbildung D.9: MontiCore-Grammatik (9 / 9)

D.1.2 Gemeinsame Obergrammatik der Essenz und der Sprachdateien

Die Abbildung D.10 zeigt die gemeinsame Obergrammatik der Essenz und der Sprachdateien, wie sie bei der Formalisierung in Abbildung 5.1 auf Seite 84 verwendet wurde.

MontiCore-Grammatik

```
1  package mc.common;
2
3  grammar Common  {
4
5    Direction =
6      direction:["<-"|"--"|"<->"|"->"];
7
8    QName =
9      name:IDENT ("." name:IDENT)*;
10
11   JType =
12     "/" name:IDENT ("." name:IDENT)*;
13
14   Card =
15     "min" "=" Min:NUMBER "max" "=" (Max:NUMBER | Max:"*") ;
16
17   token NUMBER =
18     ( '0'..'9' )+;
19 }
```

Abbildung D.10: Gemeinsame Obergrammatik der Essenz und der Sprachdateien

D.1.3 Essenz der Aktionssprache

Die Abbildung D.11 zeigt die Aktionssprache, wie sie bei der Formalisierung in Abbildung 5.2 auf Seite 85 verwendet wurde.

MontiCore-Grammatik

```
1  package mc.common;
2
3  grammar Common  {
4
5    interface Action;
6
7    Push implements Action =
8      "push" "(" stackName:IDENT "," varName:IDENT ")" ";" ;
9
10   Pop implements Action =
11     "pop" "(" stackName:IDENT ")" ";" ;
12
13   Constr implements Action =
14     Stmt* ;
15
16   interface Stmt;
17
18   Assign implements Stmt =
19     attName:IDENT "=" varName:IDENT;
20
21   Append implements Stmt =
22     attName:IDENT "+=" varName:IDENT;
23
24 }
```

Abbildung D.11: Essenz der Aktionssprache

D.1.4 Essenz der MontiCore-Grammatik

Die Abbildungen D.12-D.14 zeigen die Essenz der MontiCore-Grammatik, wie sie bei der Formalisierung in Abbildung 5.3 auf Seite 85 verwendet wurde.

MontiCore-Grammatik

```
1  package mc.grammaressence;
2
3  grammar Grammar extends mc.common.Common {
4
5    Grammar =
6      "grammar" packageName:QName name:IDENT Alphabet
7      ("extends" supergrammar:Grammar ("," supergrammar:Grammar)* )?
8      "{" ( Prod | ASTRule | Association )* "}" ;
9
10   interface Prod;
11
12   interface ParserProd extends Prod;
```

Abbildung D.12: Essenz der MontiCore Grammatik (1 / 3)

MontiCore-Grammatik

```
13  LexProd implements Prod =
14    "token" name:IDENT RegularExpression "->" JType ":" MappingJava ";" ;
15
16  EnumProd implements Prod =
17    "enum" name:IDENT "=" Constant ("|" Constant)* ";";
18
19  ExternalProd implements Prod =
20    "external" name:IDENT JType ";";
21
22  InterfaceProd =
23    "interface" name:IDENT
24    (
25      "extends" superInterfaceRule:IDENT ("," superInterfaceRule:IDENT)*
26    |
27      "[|" Parameter ("," Parameter)* "|]"
28    )*
29    ";";
30
31  AbstractProd implements Prod =
32    "abstract" (External:["/"])? name:IDENT
33    (
34      "extends" superRule:IDENT ("," superRule:IDENT)*
35    |
36      "implements" superInterfaceRule:IDENT ("," superInterfaceRule:IDENT)*
37    |
38       "[|" Parameter ("," Parameter)* "|]"
39    )* ";";
40
41  ClassProd implements ParserProd =
42    name:IDENT type:IDENT
43    (
44       "extends" superRule:IDENT ("," superRule:IDENT)*
45    |
46       "implements" superInterfaceRule:IDENT ("," superInterfaceRule:IDENT)*
47    |
48      "[|" Parameter ("," Parameter)* "|]"
49    )*
50    "returns" returnType:IDENT
51    "=" Body ";";
52
53  Parameter =
54    name:IDENT prodElementType:[":"|"!"|"="] reference:IDENT Card ;
55
56  Body =
57    alts:Alt ("|" alts:Alt)* ";" ;
58
59  Alt =
60    Components:RuleComponent* ;
61
62  interface RuleComponent;
63
64  Block implements RuleComponent =
65    "(" Alts:Alt ("|" Alts:Alt)* ")"
66    (Iteration:["?"|"*"|"+"])?;
67
```

Abbildung D.13: Essenz der MontiCore Grammatik (2 / 3)

MontiCore-Grammatik

```
68  ProdElementAction implements RuleComponent = Action ;
69
70  external Action;
71
72  ProdElement implements RuleComponent =
73   (
74     name:IDENT prodElementType:["="|":"] ProdElementRef
75   |
76     ProdElementRef prodElementType:["!"]
77   );
78
79  interface ProdElementRef;
80
81  Constant implements ProdElementRef =
82    (humanName:IDENT ":")? name:STRING ;
83
84  NonTerminal implements ProdElementRef =
85    name:IDENT ParserParameter
86    ( "[|" ruleParameters:IDENT ("," ruleParameters:IDENT )* "|]" )?;
87
88  Terminal implements ProdElementRef =
89    name:STRING ;
90
91  ParserParameter =
92    "<" (parserParameter:[PARSER:"global"])? Parameter:IDENT ">";
93
94  ASTRule =
95     "ast" type:IDENT
96    (
97      "astextends" aSTSuperClass:Type ("," aSTSuperClass:Type)*
98    |
99      "astimplements" aSTSuperInterface:Type ("," aSTSuperInterface:Type)*
100   )*
101   ( "=" AttributeInAST*  )? ";";
102
103 AttributeInAST =
104   name:IDENT ":" Type Card;
105
106 Association =
107    "association" name:IDENT left:AssociationEnd Direction right:AssociationEnd ";" ;
108
109 AssociationEnd =
110  Type name:IDENT Card;
111
112 Type =
113   name:IDENT;
114
115 Alphabet =
116   value:STRING;
117
118 RegularExpression =
119   value:STRING;
120
121 MappingJava =
122   value:STRING; }
```

Abbildung D.14: Essenz der MontiCore Grammatik (3 / 3)

D.1.5 Unterschiede zwischen Essenz und Grammatikformat

In diesem Abschnitt sind die Unterschiede der MontiCore-Grammatik aus Abschnitt D.1.1 zur Essenz aus den Abschnitten D.1.2 - D.1.4 für die einzelnen Produktionen angeben. Gibt es für einzelne Konstrukte keine Entsprechung oder handelt es sich lediglich um ein andere Strukturierung der Produktionen, bleibt die entsprechende Zelle leer. Können durch ähnliche Konstrukte dieselbe konkrete und abstrakte Syntax spezifiziert werden, wird dieses kurz beschrieben.

MontiCore-Grammatik	**Essenz**
Grammar	**Grammar**
- Das Paket ist über den einheitlichen Dateikopf des MontiCore-Frameworks verfügbar. - Das Alphabet wird über Optionen definiert. - Alle `Prod` und `ParserProd` sind einzeln als Attribute verfügbar.	- Das Paket ist ist direkt als Attribut verfügbar. - Das Alphabet wird direkt spezifiziert. - Nur `Prod` als Attribut verfügbar.

Optionen (Umfasst die Produktionen `GrammarOption`, `ParserOption`, `LexerOption`, `HeaderOption`, `VariousOption`, `PrettyPrinterOption`, `CompilationOption`, `KeywordOption`, `LexerRuleOption`, `FollowOption`, `GenericOption`)	**Optionen**
	- Optionen definieren technische Details, die in der Essenz nicht betrachtet werden.

LexProd (inklusive der Produktionen `LexComponent`, `LexBlock`, `LexCharRange`, `LexChar`, `LexString`, `LexActionOrPredicate`, `LexNonTerminal`, `LexOption`)	**LexProd**
	- Die Essenz verwendet ein vereinfachtes Format auf Basis eines regulären Ausdrucks.

EnumProd	**EnumProd**
- Das Attribut `External` zeigt an, dass die AST-Klasse extern implementiert wird.	

ExternalProd	**ExternalProd**
- Der `ExternalType` ist optional und wird beim Auslassen durch die Standardoberklasse ersetzt.	- `JType` erfüllt dieselbe Funktion wie `ExternalType`.

MontiCore-Grammatik	**Essenz**
InterfaceProd	**InterfaceProd**
- Das Attribut `External` zeigt an, dass die AST-Klasse extern implementiert wird. - Das Attribut `aSTSuperClass` beinhaltet die Oberschnittstellenproduktionen, die nur für die abstrakte Syntax gelten. - Die Attribute `overridestandard` und `interfaces` zeigen an, welche Produktionen in welcher Reihenfolge zum Parsen verwendet werden.	- Die Oberschnittstellenproduktionen können in den AST-Regeln spezifiziert werden. - Die Menge der Produktionen beim Parsen kann durch die Verwendung von AST-Regeln begrenzt werden und die Reihenfolge der Produktionen in der Grammatik bestimmt die Reihenfolge beim Parsen.

AbstractProd	**AbstractProd**
- Das Attribut `External` zeigt an, dass die AST-Klasse extern implementiert wird. - Das Attribut `aSTSuperClass` beinhaltet die Oberproduktionen, die nur für die abstrakte Syntax gelten. - Das Attribut `aSTSuperInterface` beinhaltet die Oberschnittstellenproduktionen, die nur für die abstrakte Syntax gelten. - Die Attribute `overridestandard` und `interfaces` zeigen an, welche Produktionen in welcher Reihenfolge zum Parsen verwendet werden.	- Die Oberschnittstellenproduktionen können in den AST-Regeln spezifiziert werden. - Die Oberschnittstellenproduktionen können in den AST-Regeln spezifiziert werden. - Die Menge der Produktionen beim Parsen kann durch die Verwendung von AST-Regeln begrenzt werden und die Reihenfolge der Produktionen in der Grammatik bestimmt die Reihenfolge beim Parsen.

ClassProd	**ClassProd**
- Das Attribut `External` zeigt an, dass die AST-Klasse extern implementiert wird. - Das Attribut `aSTSuperClass` beinhaltet die Oberproduktionen, die nur für die abstrakte Syntax gelten. - Das Attribut `aSTSuperInterface` beinhaltet die Oberschnittstellenproduktionen, die nur für die abstrakte Syntax gelten. - Das Attribut `Action` beinhaltet Aktionen, die vor dem Erkennen ausgeführt werden. - Das Attribut `Alt` enthält die Alternativen des Produktionskörpers.	- Die Oberschnittstellenproduktionen können in den AST-Regeln spezifiziert werden. - Die Oberschnittstellenproduktionen können in den AST-Regeln spezifiziert werden. - Die Aktionen können in den Produktionskörper integriert werden. - Das Attribut `Body` kapselt die Alternativen des Produktionskörpers.

MontiCore-Grammatik	**Essenz**
Body	**Body**
- Aktionen und Init-Aktionen können am Anfang jedes Block spezifiziert werden - Optionen steuern den Erkennungsprozess. - Blöcke können als Prädikate verwendet werden.	- Keine Unterscheidung zwischen den Aktionstypen. - Prädikate werden nicht betrachtet.

Parameter	**Parameter**
	- Veränderte konkrete Syntax.

Card	**Card**
	- Veränderte konkrete Syntax zur kompakteren Formulierung.

Referenzen (inklusive der Produktionen `TypeReference`, `ExternalType`, `RuleReference`, `RuleReferenceWithPredicates`)	**Referenzen**
- Referenzen dienen zur Formulierung von intenern Referenzen, die auf ander Produktionen verweisen und externe Java-Typen.	- Hier wird vereinfachend `Name` und `JType` verwendet.

ProdElement (inklusive der Produktionen `NonTerminal`, `Terminal` und `Constant`)	
- Realisierung als einzelene Produktionen.	- Aufteilung in `ProdElement`, `ParserParameter` und `ProdElementRef`

Constantgroup	
- Bezeichnet eine Menge an Konstanten für ein Attribut	- Durch verschiedene Konstanten kann derselben Effekt erzeugt werden.

Anything	
- Bezeichnet ein beliebiges Token	- Durch Aufzählung aller Tokenklassen nachahmbar.

MCAnything	
- Bezeichnet ein Token, das einem Parsefehler entspricht (Nur intern für die Einbettung verwendet)	

Anything	
- Zeigt das Dateiende an.	

MontiCore-Grammatik	Essenz
Semanticpredicateoraction	**ProdElementAction**
- Aktion oder semantisches Prädikat.	- Ausschließlich Aktion aus der Aktionssprache.

Concept/MCConcept	
- Erweiterungspunkt der Grammatik, der in der Essenz nicht betrachtet wird.	

ASTRule	**ASTRule**
- Das Attribut `method` beinhaltet Methoden für die AST-Klassen.	

AttributeInAST	**AttributeInAST**
- Das Attribut `derived` beschreibt die Definition von abgeleiteten Attributen.	

AssociationBlock/AssociationInBlock	**AssociationBlock/AssociationInBlock**
- Die Produktion `AssociationBlock` erlaubt die Zusammenfassung von Assoziationen.	- Die Assoziationen können einzeln spezifiziert werden.

Association	**Association**
- Der Name ist optional, er ergibt sich aus den beteiligten Klassen	

AssociationEnd	**AssociationEnd**
- `typ` und `name` sind zu einem Attribut zusammengefasst und in objektorientierter Notation aufgeschrieben. Somit ergibt sich aus der Name eines Assoziationsendes in der Essenz aus dem gegenüberliegenden `AssociationEnd` in der Grammatik.	
- Card in UML-typischer Notation	- Verwendung der Notation aus der Common-Sprache.

D.2 MontiCore-Sprachdateien

D.2.1 Vollständige Grammatik

Die Abbildungen D.15 und D.16 zeigen die Sprachdefinition der MontiCore-Sprachdateien. Zu Gunsten einer kompakten Notation wurden die Kommentare und zusätzliche Attribute entfernt.

MontiCore-Grammatik

```
1  package mc.grammar.mcdsltool;
2
3  grammar DSLToolConcept {
4
5    options{
6      parser lookahead=4   lexer lookahead=2
7      compilationunit DSLToolUnit
8    }
9
10   DSLToolUnit =
11     "language" name:IDENT ConceptDsltool;
12
13   ConceptDsltool =
14     "{"
15     ( Root | ParsingWorkflow | SymboltableWorkflow | RootFactory | Concept )*
16     "}" ;
17
18   Root=
19     "root" rootClass:IDENT "<" (DSL ".")? prod:IDENT ">" ";";
20
21   ParsingWorkflow  =
22     "parsingworkflow" (["extern"])? name:IDENT
23     "for" rootClass:IDENT "<" (DSL ".")? Rule:IDENT ">" ";";
24
25   RootFactory  =
26     "rootfactory" Name:IDENT
27     "for" RootClass:IDENT "<" (DSL ".")? Rule:IDENT ">"
28     "{"
29     (
30       ("language")=> LanguageRef
31     |
32       ("prettyprint")=> pp:["prettyprint"] "{" (prettyprinters:Type ";")+ "}"
33     |
34       FragmentProduction
35     |
36       Annotation
37     )*
38     "}";
39
40   interface Element;
41
42   FragmentProduction implements Element =
43     (DSL ".") prodname:IDENT elementname:IDENT (start:["<<start>>"])?
44     ("in" (Emb|EmbWithParameter) ("," (Emb|EmbWithParameter))* )? ";" ;
45
46   LanguageRef implements Element =
47     "language" DSL elementname:IDENT (Start:["<<start>>"])?
48     ("in" (Emb|EmbWithParameter) ("," (Emb|EmbWithParameter))* )? ";" ;
```

Abbildung D.15: MontiCore Sprachdateien (1 / 2)

MontiCore-Grammatik

```
49   Emb =
50     elementName:IDENT "." prodName:IDENT;
51
52   EmbWithParameter =
53     elementName:IDENT "." prodName:IDENT "(" Parameter:STRING ")";
54
55   DSL =
56     ( Package:IDENT& "." )* DSLName:IDENT ;
57
58   SymboltableWorkflow =
59     "symboltableworkflow" name:IDENT
60     "for" rootClass:IDENT "<" (ruleDSL:DSL "." )? rule:IDENT ">"
61     ("with" (scopeDSL:DSL ".")? scopeName:IDENT)? ";";
62
63   Type =
64     ( Package:IDENT& "." )* name:IDENT;
65
66   Concept=
67     "concept" name:IDENT Concept:MCConcept(parameter Name);
68
69   external MCConcept;
70
71   Annotation =
72     "annotation" stackname:IDENT value:IDENT ";" ;
73
74 }
```

Abbildung D.16: MontiCore Sprachdateien (2 / 2)

D.2.2 Essenz der Sprachdateien

Die Abbildung D.17 zeigt die Essenz der Sprachdateien, die zur Formalisierung in Abbildung 5.5 auf Seite 114 verwendet wurde.

MontiCore-Grammatik

```
1  package mc.languageessence;
2
3  grammar Language extends mc.common.Common, mc.grammaresssence.Grammar {
4
5    Language =
6      "language" Package:QName name:IDENT
7      "{"
8        ( SubLanguage |  FragmentProduction )*
9      "}";
10
11   interface Element;
12
13   FragmentProduction implements Element =
14     Grammar prodName:IDENT elementName:IDENT (start:["<<start>>"])?
15     ("in" (Emb|EmbWithParameter) ("," (Emb|EmbWithParameter))* )? ";" ;
16
17   SubLanguage implements Element =
18     Language elementname:IDENT (Start:["<<start>>"])?
19     ("in" (Emb|EmbWithParameter) ("," (Emb|EmbWithParameter))* )? ";" ;
20
21   Emb =
22     elementName:IDENT "." prodName:IDENT;
23
24   EmbWithParameter =
25     elementName:IDENT "." prodName:IDENT "(" Parameter:STRING ")";
26
27 }
```

Abbildung D.17: Essenz der MontiCore Sprachdateien

D.2.3 Unterschiede zwischen Essenz und den vollständigen Sprachdateien

In diesem Abschnitt sind die Unterschiede der MontiCore-Sprachdateien aus Abschnitt D.2.1 zur Essenz aus dem Abschnitt D.2.2 für die einzelnen Produktionen angeben. Gibt es für einzelne Konstrukte keine Entsprechung oder handelt es sich lediglich um ein andere Strukturierung der Produktionen, bleibt die entsprechende Zelle leer. Können durch ähnliche Konstrukte dieselben Effekte erreicht werden, wird dieses kurz beschrieben.

MontiCore-Sprachdateien	**Essenz**
DSLToolUnit / ConceptDsltool / RootFactory	**Language**
	- Vereinfachte Struktur

Root	
- Zusätzliche Generierung von Komponenten für das DSLTool-Framework, die in der Formalisierung nicht betrachtet werden.	

ParsingWorkflow	
- Zusätzliche Generierung von Komponenten für das DSLTool-Framework, die in der Formalisierung nicht betrachtet werden.	

FragmentProduction	**FragmentProduction**
- Verweis auf das Fragment über dessen Namen mittels des Attributs `DSL`	- Direkter Verweis auf das Fragment

SubLanguage	**LanguageRef**
- Verweis auf das Fragment über dessen Namen mittels des Attributs `DSL`	- Direkter Verweis auf die Sprache

SymboltableWorkflow	
- Erweiterung durch die Symboltabelleninfrastruktur CTS, die hier nicht betrachtet wird.	

Concept/MCConcept	
- Erweiterungspunkt der Sprachdateien, der in der Essenz nicht betrachtet wird.	

Annotation	
- Setzen von initialen Werten für die Einbettung	- Diesselben Effekte lassen sich durch geeignete Aktionen in den Fragmenten erreichen.

Anhang E

Index der Formalisierung

E.1 Funktionen

AN_g	*Name* $\rightarrow$ *Association* Parameter: $g \in$ *Grammar* Bildet einen Assoziationsnamen auf die Assoziation ab	Definition 5.4.27 Seite 104
AN'_g	*Name* $\rightarrow$ *Association* Parameter: $g \in$ *Grammar* Bildet einen Assoziationsnamen auf die Assoziation ab und beachtet dabei Sprachvererbung	Definition 5.4.28 Seite 104
ast_g	*Name* $\rightarrow$ *ASTRule* Parameter: $g \in$ *Grammar* Bildet einen Typnamen auf die entsprechende AST-Regeln oder die leere Menge ab	Definition 5.4.13 Seite 93
att_g	$\top \rightarrow D$ Parameter: $g \in$ *Grammar* Berechnet die Datenstruktur D aus den Produktionen einer Grammatik	Definition 5.4.19 Seite 98
att_g^T	*JType* $\rightarrow D$ Parameter: $g \in$ *Grammar* Berechnet die Datenstruktur D aus den Typen, die aus einer Grammatik hervorgehen	Definition 5.4.25 Seite 103
$\mathrm{att}_g^{T'}$	*JType* $\rightarrow D$ Parameter: $g \in$ *Grammar* Berechnet die Datenstruktur D aus den Typen, die aus einer Grammatik hervorgehen und deren Obertypen	Definition 5.4.26 Seite 103
CV_g	*ClassProd* $\rightarrow \wp($*NonTerminal* $\times$ *CurVar*$)$ Parameter: $g \in$ *Grammar* Berechnet die Nichtterminale und die Belegung der Variablen	Definition 5.4.36 Seite 109

DToAttr	$D \times Grammar \times JType \to \wp(Attr)$ Bildet die Datenstruktur D für einen Typen aus einer Grammatik auf die entsprechenden Attribute des UML/P-Klassendiagramms ab	Definition 5.6.1 Seite 111
DToComp	$D \times Grammar \times JType \to \wp(Assoc)$ Bildet die Datenstruktur D für einen Typen aus einer Grammatik auf die entsprechenden Kompositionen des UML/P-Klassendiagramms ab.	Definition 5.6.2 Seite 111
DType_g	$Name \to JType$ Parameter: $g \in Grammar$ Bildet einen Produktionsnamen auf den von der Produktion defininierten Java-Typen ab.	Definition 5.4.9 Seite 92
DType_g^{TN}	$Name \to JType$ Parameter: $g \in Grammar$ Bildet einen Typnamen auf den von der Produktion defininierten Java-Typen ab.	Definition 5.4.10 Seite 92
EN_l	$Name \to Element$ Parameter: $l \in Language$ Bildet einen Namen auf das enstprechende Element ab	Definition 5.10.1 Seite 115
PN_g	$Name \to Grammar \times Prod$ Parameter: $g \in Grammar$ Bildet den Namen einer Produktion einer Grammatik auf das Grammatik-Produktionspaar ab	Definition 5.4.1 Seite 88
PN'_g	$Name \to Grammar \times Prod$ Parameter: $g \in Grammar$ Bildet den Namen einer Produktion einer Grammatik auf das Grammatik-Produktionspaar ab und beachtet dabei die Sprachvererbung	Definition 5.4.2 Seite 89
Prods_g	$JType \to Grammar \times Prod$ Parameter: $g \in Grammar$ Die Funktion bildet einen Typ auf die Grammatik-Produktions-Paare ab, die zur Definition des Typs geführt haben	Definition 5.4.24 Seite 102
qname	$List\,\langle Name\rangle \times Name \to JType$ Bildet einen Paket und einen Namen auf eine vollqualifiizierten Namen ab	Definition 5.4.6 Seite 91

TN'_g	$Name \to \wp\{Grammar \times Prod\}$ Parameter: $g \in Grammar$ Bildet den Namen eines Typens einer Grammatik auf die definierenden Grammatik-Produktionspaare ab und beachtet dabei die Vererbung. Für Klassenproduktionen können dieses mehrere Paare sein	Definition 5.4.4 Seite 90
Type_g	$Name \to JType$ Parameter: $g \in Grammar$ Bildet einen Produktionsnamen auf den von der Produktion returnierten Java-Typen ab	Definition 5.4.11 Seite 93
Type_g^{Att}	$Attrib \times JType \to JType$ Parameter: $g \in Grammar$ Bildet ein Attribut innerhalb eines Typs auf den Typ des Attributs ab	Definition 5.4.21 Seite 99
Type_l^{El}	$Name \to JType$ Parameter: $l \in Language$ Bildet ein Element auf den returnierten Typen ab	Definition 5.10.3 Seite 115
Type_g^{Pr}	$Parameter \to JType$ Parameter: $g \in Grammar$ Bildet einen Parameter auf den Java-Typen ab	Definition 5.4.12 Seite 93
Type_g^{Var}	$Name \times ClassProd \to JType$ Parameter: $g \in Grammar$ Bildet einen Variablenname auf seinen Java-Typen ab	Definition 5.4.33 Seite 106
var	$\top \to D$ var berechnet die Datenstruktur D für die Variablen aus den Produktionen einer Grammatik	Definition 5.4.32 Seite 105
vars	$CurVar \times \wp(NonTerminal \times CurVar) \times \top \to$ $CurVar \times \wp(NonTerminal \times CurVar)$ vars berechnet die Datenstruktur $CurVar$ innerhalb Produktionen einer Grammatik	Definition 5.4.35 Seite 108

E.2 Mengen

$\succ_g$	$\succ_g \subset \wp(JType \times JType)$ Parameter: $g \in Grammar$ Untertyprelation, die sich aufgrund der Elemente der Grammatik ergibt	Definition 5.4.14 Seite 93
$\succ_g^*$	$\succ_g^* \subset \wp(JType \times JType)$ Parameter: $g \in Grammar$ Reflexiver und transitiver Abschluss von $\succ_g$	Definition 5.4.14 Seite 93

E.3 Prädikate

$\succ^{\pi}_{g,g'}$	$\succ^{\pi}_{g,g'} \subset \wp(ParserProd \times ParserProd)$ Parameter: $g, g' \in Grammar$ Das Prädikat is wahr, wenn die formalen Parameter der linken Produktion eine Verfeinerung der rechten sind	Definition 5.4.15 Seite 94

E.4 Symbole

$\top$	ASTNode (valider Obertyp aller Klassen und Schnittstellen)
$\emptyset$	leere Menge
$\mathbb{N}$	Menge der natürlichen Zahlen
$\wp$	Potenzmenge
$\tilde{\tau}$	Typfehler

E.5 Verzeichnis der Datentypen

Anhang F

Lebenslauf

Name	Krahn
Vorname	Holger
Geburtstag	27.04.1978
Geburtsort	Braunschweig
Staatsangehörigkeit	deutsch

2004 - 2009	Wissenschaftlicher Mitarbeiter am Institut für Software Systems Engineering TU Braunschweig
2004	Abschluss als Diplom-Informatiker
1998 - 2004	Studium der Informatik an der TU Braunschweig
1997 - 1998	Grundwehrdienst
1997	Abitur
1984 - 1997	Grundschule, Orientierungsstufe und Gymnasium in Braunschweig

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.