RWTH Aachen

Department of Computer Science
Technical Report

Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

Hans Grönniger

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2010-17

RWTH Aachen · Department of Computer Science · August 2010

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

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

vorgelegt von

Dipl.-Inform. Hans Grönniger
aus Meppen

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Professor Juergen Dingel, Ph.D.

Tag der mündlichen Prüfung: 16.06.2010

Die Druckfassung dieser Dissertation ist unter der ISBN 978-3-8322-9286-7 erschienen.

Kurzfassung

Eine Voraussetzung für den Erfolg einer modellbasierten Methode zur Softwareentwicklung ist eine variable und dennoch präzise Definition der verwendeten Modellierungssprachen. Hierzu gehört insbesondere auch die explizite Definition der Bedeutung der Sprachen – ihrer Semantik. Formale Semantik leistet einen wichtigen Beitrag zur unmissverständlichen Kommunikation zwischen den Beteiligten und kann einen hohen Grad der Automatisierung über interoperable Werkzeuge ermöglichen. Diese Arbeit beschäftigt sich mit der vollständigen, formalen Definition objektbasierter Modellierungssprachen und legt den Fokus auf die flexible Definition problemadäquater Semantik.

Zur Definition der Semantik objektbasierter Modellierungssprachen wird das Systemmodell als von allen Sprachen gemeinsam genutzte semantische Domäne definiert. Das Systemmodell charakterisiert abstrakt die Struktur, das Verhalten und die Interaktion von Objekten objektbasierter Systeme. Die prädikative semantische Abbildung von Elementen der Syntax einer Sprache in das Systemmodell definiert für ein Modell eine Menge von objektbasierten Systemen und erzeugt damit eine präzise Bedeutung auch für unvollständige bzw. unterspezifizierte Modelle. Die Semantik von Modellkomposition und -verfeinerung kann auf Basis des Systemmodells durch einfache mathematische Operationen erklärt werden.

Gemäß einer allgemeinen Klassifikation der möglichen Variabilität einer Modellierungssprache können die verschiedenen Sprachbestandteile wie Syntax, semantische Domäne und semantische Abbildung mit Varianten ausgestattet werden. Dies ermöglicht die Anpassung einer Sprache zum Beispiel an einen projektspezifischen Kontext. Varianten und ihre Abhängigkeiten werden in Feature-Diagrammen dokumentiert. Eine Konfiguration, also die Auswahl von Sprachvarianten gemäß der Feature-Diagramme, legt dann die Menge der gültigen Definitionen fest.

Die eingeführte Werkzeugunterstützung basiert auf MontiCore, einem Framework zur modularen Erstellung von textuellen Modellierungssprachen und dem Theorembeweiser Isabelle/HOL. Hierdurch wird eine flexible und gleichzeitig maschinenlesbare Syntax und Semantik gewonnen. Auch die Definition und Konfiguration von Semantikvarianten wird unterstützt.

Die Definition der Semantik der UML-Teilmenge UML/P dient der Erprobung des werkzeugunterstützten Vorgehens. Betrachtet werden Klassendiagrammen, Objektdiagramme, Statecharts und Sequenzdiagramme. Zudem werden als Aktions- bzw. Constraint-Sprache vereinfachte Versionen von Java und OCL verwendet.

Die angegebenen Semantikvarianten können direkt zur maschinenunterstützten Verifikation im Theorembeweiser verwendet werden, da sich zusätzlich auch konkrete Modelle automatisch in Isabelle übersetzen lassen. Die Verifikation wird an Beispielen unter Verwendung integrierter Semantiken verschiedener Modellierungssprachen der UML/P demonstriert. Dies zeigt die praktische Verwendbarkeit des Ansatzes.

Abstract

A successful model-based software development method requires a variable yet precise definition of the modeling languages used. This especially involves the explicit definition of the meaning of the language, i.e., its semantics. Formal semantics contributes to an unambiguous communication between people and may enable a high degree of automation using interoperable tools. This thesis is concerned with the complete, formal definition of object-based modeling languages. It concentrates on the flexible definition of problem-oriented semantics.

For the definition of the semantics of object-based modeling languages the system model is defined as a commonly used semantic domain. The system model abstractly characterizes the structure, behavior, and interaction of objects in an object-based system. The predicative semantic mapping of elements of the syntax of a language into the system model defines a set of object-based systems. The mapping thereby establishes a precise mapping even of incomplete or underspecified models. Based on the system model, semantics of model composition and model refinement can be defined by simple mathematical operations.

According to a general classification of possible variability within a modeling language, the different constituents of a language like syntax, semantic mapping, and semantic domain can be equipped with variants. This enables the tailoring of a language to, for example, a project-specific context. Variants and their interdependency are captured using feature diagrams. A configuration, i.e., the selection of language variants according to the feature diagrams, then denotes the set of effective definitions.

Tool support is introduced using MontiCore, a framework for the development of textual languages, and the theorem prover Isabelle/HOL. A flexible and at the same time machine readable syntax and semantics is thus obtained. Furthermore, the definition and configuration of semantics variants is supported.

Semantics of the UML subset UML/P is defined using the tool support in order to validate the approach. Semantics of class diagrams, object diagrams, statecharts, and sequence diagrams is considered. Additionally, simple versions of Java, acting as an action language, and OCL, used as a constraint language, are defined.

The given UML/P semantics variants can be used directly to conduct tool-supported verification in the theorem prover because concrete models can be translated automatically to Isabelle as well. Using the integrated semantics of several UML/P modeling languages, verification is demonstrated with the help of concrete examples. The examples substantiate the practical usability of the approach.

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr. Bernhard Rumpe für die Betreuung dieser Dissertation und seine sehr konstruktiven, weitsichtigen Anregungen und Ratschläge bedanken. Die vorliegende Arbeit ist im Rahmen des DFG-Projekts „rUML“ entstanden. Ich danke ihm auch dafür, mir diese Projekt anvertraut zu haben. Prof. Juergen Dingel danke ich für die Übernahme des zweiten Gutachtens. Die fachliche Diskussion mit ihm hat mir immer neue Perspektiven aufgezeigt. Von beiden habe ich während meiner wissenschaftlichen Tätigkeit viel gelernt.

Für hilfreiche Kommentare zu Vorversionen dieser Arbeit danke ich Dr. María Victoria Cengarle, Boris Gajanovic, Dr. Holger Krahn, Dirk Reiß und Jan Oliver Ringert. Victoria Cengarle war außerdem Mitstreiterin im DFG-Projekt „rUML“, für die erfolgreiche Zusammenarbeit möchte ich ihr ebenfalls danken. Für die Entwicklung von MontiCore und die Diskussionen hierüber danke ich besonders Holger Krahn, Martin Schindler und Steven Völkel. Martin Schindler danke ich zusätzlich für die Bereitstellung von MontiCore-Grammatiken, die ich in meiner Arbeit weiterverwenden konnte.

Mein besonderer Dank gilt meiner Frau Ivonne für ihre Unterstützung und ihr ausdauerndes Verständnis dafür, dass mich die Dissertation oft ganz in Beschlag genommen hat. Außerdem danke ich meiner Mutter für das Korrekturlesen und Aufspüren von Tippfehlern. Meiner ganzen Familie danke ich dafür, dass sie mich immer unterstützt und ermutigt hat.

Inhaltsverzeichnis

I Grundlagen **1**

1 Einführung **3**

1.1 Einleitung 3
1.2 Motivation 4
1.3 Wichtigste Ziele und Ergebnisse 8
1.4 Nahestehende Arbeiten 9
1.5 Aufbau der Arbeit 10

2 Grundlagen der Semantikbildung **11**

2.1 Bestandteile einer vollständigen Sprachdefinition 11
2.2 Charakteristische Eigenschaften des gewählten Ansatzes 15
2.3 Notation 16
2.4 Beispiel einer Sprachdefinition 17
  2.4.1 Konkrete Syntax von CDSimp 18
  2.4.2 Abstrakte Syntax von CDSimp 19
  2.4.3 Kontextbedingungen von CDSimp 20
  2.4.4 Semantische Domäne $\mathrm{SystemModel}_{simp}$ 20
  2.4.5 Semantische Abbildung von CDSimp 21
2.5 Einordnung des Ansatzes und verwandte Arbeiten 23

3 Variabilität in Modellierungssprachen **27**

3.1 Klassifikation von Variabilität in einer Sprachdefinition 28
  3.1.1 Präsentationsvariabilität 28
  3.1.2 Syntaktische Variabilität 31
  3.1.3 Semantische Variabilität 33
3.2 Erfassen und Konfigurieren von Variabilität 35
  3.2.1 Feature-Diagramme zur Dokumentation der Varianten 37
  3.2.2 Konfiguration von Variabilität 39
  3.2.3 Überprüfen der Konfiguration 40
  3.2.4 Konsistenz der Konfiguration 40
3.3 Vorgehen bei Sprachänderungen 40
3.4 Verwandte Arbeiten 42
3.5 Zusammenfassung 43

4 Systemmodell 45
4.1 Einführende Erläuterungen 45
  4.1.1 Entwurfsprinzipien und -entscheidungen 45
  4.1.2 Aufbau des Systemmodells 47
4.2 Typnamen, Werte und Variablen 49
4.3 Klassen und Objekte 52
4.4 Datenzustand und Assoziationen 57
  4.4.1 Assoziationen 59
4.5 Methoden, Threads und Kontrollzustand 63
4.6 Nachrichten und Nachrichtenzustand 71
4.7 Objekt- und Systemzustände 74
4.8 Objekt- und Systemverhalten 76
4.9 Zusammenfassung 80

II Werkzeugunterstützung 81

5 Verwendete Werkzeuge und Übersicht über das Vorgehen 83
5.1 Auswahl der Werkzeuge 83
  5.1.1 Anforderungen und Ziele 83
  5.1.2 Eingesetzte Werkzeuge 84
5.2 Isabelle/HOL 84
  5.2.1 Theorien 85
  5.2.2 HOL-Konstrukte 85
  5.2.3 Typaliasse und Datentypen 86
  5.2.4 Konstantendeklarationen 87
  5.2.5 Funktionsdefinitionen und induktive Definitionen 87
  5.2.6 Beweise 88
5.3 Das MontiCore-Framework 89
5.4 Feature-Diagramme mit MontiCore 91
5.5 Vorgehen zur werkzeugunterstützten Definition einer Modellierungssprache . . 93
  5.5.1 Organisation der Sprachdefinition 96
  5.5.2 Alternative Vorgehensweisen 98
5.6 Verwandte Arbeiten 99

6 Werkzeugunterstützte Definition einer Modellierungssprache 103
6.1 Übersetzung der MontiCore-Grammatik nach Isabelle/HOL 104
  6.1.1 Lexikalische Syntax 104
  6.1.2 Kontextfreie Syntax 105
  6.1.3 Grammatikvererbung 117
  6.1.4 Einbettung 119
  6.1.5 Zusammenfassung 122
6.2 Übersetzung konkreter Modelle 123
6.3 Kodierung von Kontextbedingungen 126

6.4 Kodierung des Systemmodells 129
6.5 Kodierung semantischer Abbildungen 132
6.6 Definition und Konfiguration von Variabilität 134
6.6.1 Präsentationsvariabilität 135
6.6.2 Syntaktische Variabilität 137
6.6.3 Variabilität des Systemmodells 140
6.6.4 Variabilität semantischer Abbildungen 141
6.6.5 Zusammenfassung 142

III Fallstudien 143

7 UML/P 145
7.1 Übersicht 145
7.2 Gemeinsam genutzte Sprachteile 147
7.2.1 Literals 147
7.2.2 Types 149
7.2.3 Common 152
7.3 Java/P und OCL/P 154
7.3.1 Java/P 154
7.3.2 OCL/P 160
7.4 Klassendiagramme 164
7.5 Objektdiagramme 178
7.6 Statecharts 183
7.7 Sequenzdiagramme 189
7.8 UML/P-Konfiguration 199

8 Systemmodell-basierte Verifikation 203
8.1 Verifikationsszenarien 203
8.2 Zirkuläre Vererbung in Klassendiagrammen 206
8.3 Verletzung einer Invariante im Klassendiagramm durch ein Objektdiagramm 208
8.4 Unerlaubte Interaktion im Sequenzdiagramm bezüglich eines Statechart 211
8.5 Vergleich von Sprachvarianten 215

IV Epilog 217

9 Zusammenfassung und Ausblick 219

Literaturverzeichnis 225

V Anhänge 239

A Mathematische Grundlagen 241

A.1 Funktionen, Logik, Mengen . . . 241
A.2 Listen, Puffer, Stacks . . . 242
A.3 Zustandsmaschinen . . . 243

B Isabelle-Theorien des Systemmodells 245
B.1 Base . . . 245
B.2 Type . . . 247
B.3 Object . . . 250
B.4 Data . . . 254
B.5 Control . . . 260
B.6 Messages . . . 267
B.7 State . . . 270
B.8 SMSTS . . . 272
B.9 SystemModel-base . . . 275
B.10 Variabilität des Systemmodells . . . 276

C UML/P-Syntax 277
C.1 Gemeinsam genutzte Sprachteile . . . 277
  C.1.1 Literals . . . 277
  C.1.2 Types . . . 284
  C.1.3 Common . . . 292
C.2 Java/P und OCL/P . . . 296
  C.2.1 Java/P . . . 296
  C.2.2 OCL/P . . . 300
C.3 Klassendiagramme . . . 304
C.4 Objektdiagramme . . . 313
C.5 Statecharts . . . 317
C.6 Sequenzdiagramme . . . 325

D Glossar 333

E Abkürzungen 339

F Index der Systemmodelldefinitionen 341

G Lebenslauf 345

Teil I

Grundlagen

Kapitel 1
Einführung

1.1 Einleitung

Der Entwicklung von Software kommt in vielen Industriezweigen eine hohe Bedeutung zu, da Software heute ein wichtiger Produktbestandteil und häufig ein entscheidender Innovationstreiber ist. Softwareentwickler sehen sich dabei mit immer komplexeren Systemen konfrontiert. Steigende Funktionsvielfalt, schnelle und flexible Umsetzung von sich ändernden Anforderungen und hohe Ansprüche an die Qualität des zu entwickelnden Produkts sind nur einige Herausforderungen.

Als Mittel zur Beherrschung der Komplexität und Steigerung der Produktivität wird der Modellierung im Entwicklungsprozess ein immer größerer Stellenwert zugemessen. Durch Modellierung verspricht man sich eine im Vergleich zur Programmierung verbesserte Abstraktion und eine damit einhergehende vereinfachte Kommunikation zwischen beteiligten Personengruppen wie Kunden oder Entwicklern. Zusätzlich wird eine erhöhte Automatisierung bei der Softwareerstellung als Argument für eine modellbasierte Entwicklung angeführt. Das Ziel ist eine modellbasierte Softwareentwicklung, bei der gut verständliche Modelle die zentralen Entwicklungsartefakte darstellen und durch weitgehende Automatisierung beispielsweise Programmcode oder Testfälle abgeleitet sowie wichtige Systemeigenschaften analysiert werden können.

Modellierung spielt schon immer eine wichtige Rolle in der Informatik. Im Sinne einer industriell einsetzbaren modellbasierten Softwareentwicklung ist sie aber erst mit der Konsolidierung verschiedener Methoden und objektorientierter Notationen zur *Unified Modeling Language* (UML) [OMG09a, OMG09b] in den Fokus des allgemeinen Interesses gerückt. Die UML ist mittlerweile als objektorientierte Modellierungssprache zu einem Industriestandard avanciert. Neben der UML ist auch ein Trend zur Entwicklung und Verwendung von domänenspezifischen Sprachen (DSLs) zur Softwareentwicklung zu erkennen. DSLs sollen helfen, die Produktivität weiter zu steigern, indem die Sprache genau auf den Problembereich zugeschnitten wird. Erfolgreiche DSLs sind beispielsweise Matlab Simulink [Sim10] zur Beschreibung regeltechnischer Algorithmen, SQL [Int06] als Datenbanksprache oder HTML [HTM10] zur Beschreibung und Verlinkung von Webseiten.

Eine Grundvoraussetzung für einen Erfolg eines modellbasierten Entwicklungsprozesses auf Basis von UML, verwandten Modellierungssprachen und DSLs ist, dass die verwendeten Modellierungssprachen präzise und verständlich definiert sind. Hierzu gehört insbesondere auch die explizite, formale Definition der Bedeutung der Sprache – ihrer Semantik. Semantik leistet einen wichtigen Beitrag für unmissverständliche Kommunikation zwischen den Beteiligten und kann einen hohen Grad an Automatisierung über interoperable Werkzeuge ermöglichen.

Es ist allerdings festzustellen, dass für die gängigste objektorientierte Modellierungssprache UML keine allgemein akzeptierte, formale Semantik existiert. Formale Semantik wird im UML-Standard zwar als nützlich, ihre Definition aber unter anderem aufgrund von semantischen Variationspunkten als zu schwierig erachtet [OMG09b, Abschnitte 2.3, 6.3.2]. Zudem findet sich eine Vielzahl von zum Teil konkurrierenden Ansätzen zur Semantik von Teilsprachen der UML, siehe z.B. [CD07, Bee94, HHRS05, HM08, CK04]. Allgemein gilt, dass für neu entwickelte allgemeine Modellierungssprachen und DSLs ebenfalls meist keine formale Semantik angegeben wird. Damit können modellbasierte Methoden ihr Potential häufig nicht voll ausspielen und ihr Erfolg ist gefährdet.

Das Hauptziel dieser Arbeit ist es, die *vollständige, formale Definition objektbasierter Modellierungssprache mit Variabilität* zu ermöglichen und mit Werkzeugen geeignet zu unterstützen. Der Fokus liegt hierbei auf der Definition der formalen Semantik der Sprache und ihrer semantischen Varianten. Daher werden die Gründe für die Schwierigkeiten bei der Definition einer formalen Semantik analysiert und ein Vorschlag für das Vorgehen zur Semantikdefinition gemacht, der den Herausforderungen explizit begegnet. Die Hauptaufmerksamkeit gilt der Semantik der UML/P [Rum04b], wobei das vorgestellte Vorgehen allgemein auf objektbasierte Modellierungssprachen und DSLs anwendbar ist.

1.2 Motivation

Die Motivation der Arbeit ergibt sich aus den Herausforderungen bei der Definition einer formalen Semantik. Diese Herausforderungen werden diskutiert und daraus Anforderungen abgeleitet, die ein Ansatz zur Semantikdefinition erfüllen sollte. Zusätzlich wird kurz umrissen, wie der in dieser Arbeit vorgestellte Ansatz diese Anforderungen erfüllt.

Effektivität modellbasierter Entwicklung

Der Nutzen einer modellbasierten Entwicklung konnte bislang nicht überzeugend nachgewiesen werden [MD08, HKR+09, FRR09]. Im Gegenteil, Studien in konkreten Projekten zeigen zum Teil, dass der erhoffte Produktivitätsgewinn sogar ganz ausbleiben kann [MOD06]. Die Gründe hierfür sind vielfältig. Eine Herausforderung ist die Verwendung visueller oder graphischer Modellierungssprachen. Graphische Modelle skalieren häufig nicht gut und überzeugende Bedienkonzepte für Werkzeuge zum Umgang mit großen Modellen sind aufwendig zu realisieren. Zu einer vollständigen Sprachdefinition gehört aber auch die Definition einer konkreten Syntax für die Modelle. Der in dieser Arbeit gewählte Ansatz ist zwar unabhängig von der Art der Beschreibung dieser konkreten Syntax, nichtsdestotrotz wird der Ansatz im Kontext textueller Modellierungssprachen präsentiert. Ausschlaggebend hierfür ist die Entwicklung des Frameworks MontiCore [Kra10], mit dem textuelle Modellierungssprachen einfach und modular erstellt werden können, so dass dieser Teil der Sprachdefinition bereits abgedeckt ist und in dieser Arbeit verwendet werden kann. Die Festlegung auf textuelle Sprachen stellt keine Einschränkung dar, da die Konstrukte einer Modellierungssprache sowohl graphisch als auch textuell repräsentiert werden können [HR04]. Wir können jedoch von den in [GKR+07] beschriebenen Vorteilen einer textuellen Modellierungssprache wie beispielsweise einer verhältnismäßig einfachen Erstellung von Parsern, Editoren [KRV07a] oder Generatoren profitieren.

Ein weiterer Grund für die Probleme bei der Umsetzung modellbasierter Vorgehensweisen ist das Fehlen gut funktionierender Werkzeuge zur Erstellung und Transformation von Modellen. Hinweise darauf, dass bei der Werkzeugerstellung die fehlende Semantik von Sprachen ein Problem darstellt, sind beispielsweise in [SDL$^+$08] zu finden. Die Forderung nach durchgängigen, flexiblen Werkzeugketten im modellbasierten Entwicklungsprozess rechtfertigt also ebenfalls die Fragestellung dieser Arbeit, ein systematisches, handhabbares Vorgehen zur Sprach- und insbesondere Semantikdefinition anzubieten.

Verständlichkeit der Semantik

Formale Semantik leidet unter anderem häufig unter der Unverständlichkeit und schlechten Zugänglichkeit ihrer Definitionen [Mos01]. Die Semantik einer objektbasierten Modellierungssprache soll so gestaltet werden, dass sie als Referenz für Werkzeugentwickler dienen kann. Hierbei ist eine explizite Darstellung der Semantik und eine Trennung der syntaktischen von der semantischen Ebene der Sprachdefinition von großem Nutzen für die Verständlichkeit. Nutzer von Modellierungssprachen erlernen die Sprache häufig eher an Beispielen. Aber auch sie sollen eine Semantikdefinition in Auszügen als definierende Referenz zu Rate ziehen können.

Der gewählte Ansatz erlaubt einen relativ einfachen Zugang zur Semantikdefinition in zweierlei Hinsicht.

- Die Semantik eines Modells ist definiert als Menge aller möglichen Realisierungen. Hierzu wird als semantische Domäne das Systemmodell (siehe auch [BCGR08]) als Grundlage eingeführt. Dabei handelt es sich um eine Charakterisierung wesentlicher Konzepte der Objektorientierung. Das Systemmodell beschreibt also objektorientierte Systeme, ihre Struktur, ihr Verhalten und die Interaktionen ihrer Objekte. Dabei kommen einfache mathematische Theorien basierend auf Mengen, Funktionen und Relationen zum Einsatz. Entscheidend ist, dass die Semantik eines Modells die Menge der intendierten Systeme beschreibt und so einen direkten Bezug zu Eigenschaften einer möglichen Implementierung herstellt, wohingegen andere Ansätze häufig deutlich einfachere Domänen verwenden, die diesen Bezug jedoch nicht erlauben.

- Durch die eigenschaftsorientierte semantische Abbildung ist eine gute Modularität der Definitionen möglich. Insbesondere das Nachschlagen der Bedeutung einzelner Konstrukte, ohne die Gesamtheit aller Definitionen erfassen zu müssen, ist hier als wichtiger Vorteil zu nennen.

Die in unserem Ansatz verwendete eigenschaftsorientierte semantische Abbildung ist nicht direkt zur Ausführung geeignet. Eine ausführbare Semantik kann jedoch ebenfalls als Kriterium für eine verständliche Semantik gesehen werden, da konkrete Systemabläufe nachvollziehbar werden. Allerdings müssen auch unvollständige und unterspezifizierte Modelle berücksichtigt werden, denen nicht eindeutig eine Ausführung zugeordnet werden kann [Rum02].

Unvollständige Modelle

Im Gegensatz zu Programmiersprachen werden Modellierungssprachen zur Spezifikation eingesetzt. Insbesondere in frühen Projektphasen oder wenn vornehmlich zur Kommunikation zwischen Entwicklern eingesetzt, sind Modelle häufig unvollständig (Sachverhalte sind nicht dargestellt) oder unterspezifiziert (ein Sachverhalt ist abstrahiert dargestellt, Details sind bewusst offen gelassen und bedeuten Implementierungsfreiheit für den Entwickler).

Diesen abstrakten Modellen, die nicht notwendigerweise ausführbar sind, muss eine Semantik sinnvoll zugeordnet werden können. Der in dieser Arbeit verwendete Ansatz sieht eine mengenwertige, denotationale semantische Abbildung vor. Eigenschaftsorientiert werden Mengen von Realisierungen mit Hilfe des Systemmodells charakterisiert, die zu einem Modell passen. Je vollständiger das Modell ist, desto weniger Systeme werden durch die Semantik charakterisiert.

Komposition der Semantikdefinition

Ein Softwaresystem wird typischerweise mit Hilfe verschiedener Modellierungssprachen beschrieben, die fundamental andere Paradigmen zur Beschreibung von Struktur, Verhalten und Interaktion verwenden. Dabei finden mehrere Modelle Verwendung, die unterschiedliche, sich womöglich überlappende Sichten auf das System beschreiben.

Das Systemmodell ist so detailliert und reich an Konzepten, dass es gängige Paradigmen in der Modellierung wie zustands- oder interaktionsbasierte Verhaltensbeschreibungen adäquat abdecken kann. Es wird als einzige semantische Domäne herangezogen, um eine integrierte Semantik für unterschiedliche Modellierungssprachen zu bilden. Solange die Konzepte einer Modellierungssprache im Systemmodell ausdrückbar sind, solange ist eine einfache Integration mit beliebigen anderen Systemmodell-basierten Semantiken möglich. Dies vereinfacht die Semantikintegration stark im Vergleich zu Ansätzen, die auf unterschiedliche semantische Domänen setzen, die dann meist aufwendig im Nachhinein komponiert werden müssen. In diesem Ansatz kann die Komposition der mengenwertigen Semantiken von Modellen durch Schnittmengenbildung erfolgen.

Variabilität des Einsatzkontexts

Die Erfahrung zeigt, dass universell einsetzbare Modellierungssprachen in verschiedenen Kontexten verwendet werden, so dass häufig eine leicht veränderte Interpretation von Konstrukten der Sprache notwendig wird. Im Fall von UML kommt hinzu, dass bei der Standardisierung dieser Gremiensprache unterschiedlichen Interessenslagen der beteiligten Personengruppen und Industrien Rechnung getragen werden muss. Als Konsequenz wurden in UML so genannte *semantic variation points* [OMG09b] eingeführt, die den Spielraum bei der Interpretation bestimmter Sprachkonstrukte beschreiben. Eine Semantik von UML oder anderen mit Variationspunkten ausgestatteten Sprachen kann daher nicht vollständig fixiert werden, sondern muss diese Variabilität explizit berücksichtigen.

In unserem Ansatz wird generell die Variabilität einer Sprachdefinition systematisch berücksichtigt und ein Mechanismus geschaffen, die Varianten explizit zu beschreiben und zu verwalten. Durch die präzise Formulierung der Variabilität verliert die Semantikdefinition nicht an

Präzision, erlaubt aber die geforderte flexible Interpretation.

Systematisches, werkzeugunterstütztes Vorgehen zur vollständigen Definition einer Modellierungssprache

Im Fall der UML sind Syntax und die Konzepte der Sprache im Wesentlichen geklärt. Es fehlt allerdings eine allgemein akzeptierte Semantikdefinition. Anders liegt der Fall bei neu entwickelten, formalen Modellierungssprachen, die mit einer Semantik ausgestattet werden sollen. Hier ist besonders wichtig, dass Sprachentwickler ein strukturiertes Vorgehen an die Hand bekommen, um eine Sprache vollständig (also Syntax und Semantik) zu definieren. Diese Arbeit präsentiert ein solches Vorgehen.

Das Vorgehen zur vollständigen, formalen Sprachdefinition kann rein papierbasiert durchgeführt werden. Semantikdefinitionen werden allerdings leichter akzeptiert, wenn auch eine Werkzeugunterstützung vorhanden ist [Mos01]. Unsere Werkzeugunterstützung hat dabei einige Vorteile. Zunächst erhalten wir eine Semantik, die maschinengeprüft ist und direkt für Verifikationszwecke verwendet werden kann. Des Weiteren sind alle Artefakte der Sprachdefinition besser kontrollierbar und qualitätsprüfbar, da sie Standardwerkzeugen der Softwaretechnik wie einer Versionskontrolle und automatischen Analysen zugänglich sind. Bei der Werkzeugunterstützung ist uns wichtig, die Flexibilität und Ausdrucksstärke eines papierbasierten Ansatzes zu erhalten. Dies wird durch die Verwendung eines Theorembeweisers weitgehend ermöglicht. Die abstrakte Syntax zur Darstellung der Modellierungssprache im Theorembeweiser wird direkt aus der Syntaxdefinition mit MontiCore gewonnen. Ebenfalls lassen sich mit einem Generator konkrete Modelle in die abstrakte Syntax für den Theorembeweiser übersetzen. Die Definition der Variabilität erfolgt mittels Feature-Diagrammen [CE00]. Mit Hilfe eines weiteren Generators lassen sich Konfigurationen einer Sprache erzeugen. Damit ist eine durchgängige werkzeugunterstützte Definition einer Modellierungssprache gegeben.

Wie schon erwähnt, kann die Semantikdefinition direkt zur Verifikation von Modelleigenschaften verwendet werden. Darüber hinaus wurde in [CDGR07] eine ausführbare Version des Systemmodells implementiert. Anhand der Semantik einer Modellierungssprache lässt sich hierauf aufbauend ein Simulator erstellen, in dem Modelle simuliert und validiert werden können.

Qualität der Sprachdefinition

Eine fehlende formale Semantik macht sich auch häufig in der Qualität der Sprachbeschreibung bemerkbar. Ein Beispiel ist hier wiederum die UML, bei deren Spezifikation die Semantik im Wesentlichen durch Freitext angegeben wird. Zahlreiche Publikationen weisen auf Inkonsistenzen oder Auslassungen hin, z.B. [RW99, FSKdR05]. In der Vielzahl der Fälle werden die Mängel in der Spezifikation gerade bei dem Versuch der Formulierung einer formalen Semantik für Teile der Sprache entdeckt. Eine Sprachdefinition selbst kann also von einer formalen Semantik deutlich profitieren.

UML/P-Semantik

Die in der vorliegenden Arbeit betrachtete UML-Teilmenge entspricht im Wesentlichen der UML/P [Rum04a, Rum04b]. Dabei handelt es sich um ein UML-Profil, das besonders zur agilen Softwareentwicklung geeignet ist. In der UML/P wurden, im Vergleich zu der OMG-Standardfassung der UML 2, methodisch fragwürdige oder komplizierte Konstrukte entfernt. Damit bildet die UML/P eine bereinigte, konsistente Teilmenge. In dieser Arbeit werden Semantiken für Klassendiagramme, Objektdiagramme, Statecharts, Sequenzdiagramme, eine vereinfachte, auf Java/P basierende Aktionssprache sowie eine vereinfachte, auf OCL/P basierende Constraint-Sprache geschaffen.

Eine integrierte Semantik wesentlicher UML-Teilsprachen dient auch der Erprobung des in dieser Arbeit definierten Vorgehens. Weitere UML-Teilsprachen können in Zukunft mit Hilfe der bereits beschriebenen Mechanismen einfach integriert werden. Zudem bietet die angegebene UML/P-Semantik die Möglichkeit, andere UML-ähnliche Modellierungssprachen nicht direkt auf das Systemmodell abzubilden, sondern die Semantik als eine syntaktische Transformation auf die UML/P anzugeben. Das kann die Definition einer Semantik erheblich vereinfachen.

1.3 Wichtigste Ziele und Ergebnisse

In diesem Abschnitt werden die wichtigsten Ziele und Ergebnisse dieser Arbeit kurz zusammengefasst.

1. Es wird ein Vorgehen für die vollständige und formale Definition einer Modellierungssprache definiert. Der Fokus liegt dabei auf der Definition der Semantik objektbasierter Modellierungssprachen. Die Semantik soll den in diesem Kapitel entwickelten Anforderungen genügen (Kapitel 2, Abschnitt 5.5):
   - Verständlichkeit durch Bezug zu Eigenschaften möglicher Realisierungen als objektbasiertes System und durch eine modulare Definition der semantischen Abbildung.
   - Unterstützung von Unvollständigkeit und Unterspezifikation durch eine mengenwertige semantische Abbildung.
   - Einfache Komponierbarkeit von Semantiken durch Verwendung einer einzigen semantische Domäne für semantische Abbildungen, die adäquat ist, wesentliche Konzepte der Objektorientierung zu beschreiben.
2. Auf Basis von Vorgängerarbeiten wird als semantische Domäne das Systemmodell definiert, das objektbasierte Systeme charakterisiert (Kapitel 4).
3. Die Variabilität in Modellierungssprachen wird allgemein klassifiziert (Kapitel 3).
4. Ein Mechanismus zur expliziten Beschreibung und Verwaltung von Varianten einer Sprache basierend auf Feature-Diagrammen wird vorgeschlagen (Abschnitt 3.2).
5. Für das Vorgehen wird eine Werkzeugunterstützung implementiert. Das Rahmenwerk nutzt den Theorembeweiser Isabelle/HOL und MontiCore (Kapitel 5, 6).

6. Es wird eine formale Semantik der UML/P und ausgewählter Varianten geschaffen. Dies dient auch der Erprobung des werkzeugunterstützen Vorgehens (Kapitel 7).

7. Unter Verwendung des werkzeugunterstützten Vorgehens wird diskutiert, wie die Semantikdefinition zu Verifikationszwecken eingesetzt werden kann. Verschiedene Verifikationsszenarien werden erläutert. Die praktische Durchführbarkeit von Verifikation auf Basis der UML/P-Semantik in Isabelle/HOL wird an einfachen Beispielen demonstriert (Kapitel 8).

1.4 Nahestehende Arbeiten

An dieser Stelle werden einige wichtige Arbeiten aufgeführt, auf denen diese Arbeit insgesamt basiert oder in engem Zusammenhang steht. Eine Besprechung verwandter Arbeiten bezüglich einzelner Aspekte findet sich in den entsprechenden Kapiteln der Arbeit.

Die Idee, ein Systemmodell zur Beschreibung der relevanten Eigenschaften der Systeme zu formalisieren, stammt aus dem SYSLAB Projekt [KRB96, BHH$^+$97]. Für bestimmte UML-artige Dokumenttypen wurde ebenfalls ein Systemmodell im Focus-Umfeld definiert [BS01, Rum96]. Weitergeführt wurde die Idee des Systemmodells in dem Projekt UML 2 Semantics [BCD$^+$06]. Hier entstanden die Arbeiten [BCR06, BCR07a, BCR07b]. Darauf aufbauend erfolgte eine Erweiterung im Kontext des von der DFG geförderten Projekts rUML, die in [BCGR08] zu finden ist. Das allgemeine Vorgehen zur Semantikdefinition auf Basis des Systemmodells und die Systemmodelldefinition selbst sind in zwei Kapiteln des Buches *UML 2 Semantics and Applications* [BCGR09a, BCGR09b] enthalten. Die drei letztgenannten Publikationen wurden unter Mitwirkung des Autors dieser Arbeit erstellt. Der generelle Ansatz zur Semantikbildung ist aus [Rum96, HR04] übernommen.

Im Kontext der schon erwähnten Projekte UML 2 Semantics und rUML sind einige Arbeiten entstanden, insbesondere die Dissertation von Crane [Cra09] sowie [CD08a], die sich mit der Formalisierung von UML-Aktivitäten beschäftigen und somit kaum Überschneidungen mit dieser Arbeit aufweisen. Vielmehr ist die formale Definition von Aktivitäten als Ergänzung zu den in der vorliegenden Arbeit untersuchten Modellierungssprachen zu sehen. Eine der ersten Arbeiten in diesem Projektkontext stammt von Cengarle [Cen07], die sich, basierend auf der ersten Fassung des Systemmodells, mit der Semantik von UML-Interaktionen beschäftigt. Des Weiteren wurden im Rahmen des genannten DFG-Projekts unter Mitwirkung des Autors Berichte zur Semantik von Klassendiagrammen [CGR08a], Statecharts [CGR08b] und Interaktionen [CG09] erstellt. In [CDGR07] wird mit der Simulation des Systemmodells in der funktionalen Sprache Haskell [Tho99] experimentiert und in [CD08b] wird eine Systemmodell-Implementierung in Java beschrieben.

Die Werkzeugunterstützung basiert auf dem MontiCore-Framework [KRV08], worin auch eine präzise Definition der Syntax der UML/P erfolgt ist [Sch09], die wir in dieser Arbeit verwenden. Zudem setzen wir den Theorembeweiser Isabelle/HOL [NPW02] ein. Erste Erfahrungen im Umgang mit dem Theorembeweiser Isabelle/HOL bei der Formalisierung des Systemmodells wurden in [Rin08] gesammelt und dienen als Grundlage der in dieser Arbeit enthaltenen Kodierung. Der Ansatz und die vorgeschlagene Werkzeugunterstützung wurden in [GRR09] bereits veröffentlicht. In [CGR09] wird die Werkzeugunterstützung um den Aspekt der Variabilität erweitert. Enthalten ist auch die vom Autor mitentwickelte Klassifikation der Variabilität in einer Modellierungssprache, auf denen die Ausführungen in Kapitel 3 beruhen.

1.5 Aufbau der Arbeit

In Kapitel 2 der vorliegenden Arbeit werden Grundlagen der Semantikbildung erarbeitet und unser Ansatz mit einem ersten Beispiel motiviert. Zudem wird die Arbeit in den Kontext existierender Ansätze zur Definition der Semantik von Programmier- und Modellierungssprachen gesetzt. Variabilität in der Definition einer Modellierungssprache wird in Kapitel 3 klassifiziert und adressiert alle Teile einer Sprachdefinition, nämlich Syntax, semantische Domäne und semantische Abbildung. Hierauf aufbauend werden methodische Überlegungen zum systematischen Umgang mit syntaktischen und semantischen Varianten und Konfigurationen der Varianten angestellt. Das Beispiel aus Kapitel 2 wird um Variabilität erweitert. In Kapitel 4 wird das Systemmodell definiert. Dabei werden grundlegende Entwurfsentscheidungen, die Systemmodelldefinitionen selbst und auch eine Menge von Varianten dieser Definitionen explizit beschrieben. Das Kapitel beschließt den ersten Teil über die Grundlagen dieser Arbeit.

Der zweite Teil der Arbeit erläutert die vorgeschlagene Werkzeugunterstützung für unseren Ansatz zur formalen und vollständigen Definition einer Modellierungssprache. Hierzu werden in Kapitel 5 die verwendeten Werkzeuge Isabelle/HOL und MontiCore sowie die textuellen Notationen für Feature-Diagramme und Konfigurationen beschrieben. Außerdem wird der werkzeugunterstützte Ansatz übersichtsartig dargestellt. Die einzelnen Schritte zur Definition einer Modellierungssprache mit Hilfe der vorgestellten Werkzeuge sind in Kapitel 6 dokumentiert. Außerdem wird auf die automatische Übersetzung konkreter Modelle nach Isabelle/HOL eingegangen.

Im dritten Teil der Arbeit wird das definierte Vorgehen anhand der UML/P demonstriert und dabei eine Systemmodell-basierte formale Semantik für die UML/P geschaffen. Kapitel 7 enthält die semantischen Abbildungen und deren Variabilität für Klassendiagramme, Objektdiagramme, Statecharts, Sequenzdiagramme und gemeinsam genutzter Sprachteile sowie für vereinfachte Versionen von Java/P und OCL/P, die als Aktions- bzw. Constraint-Sprache verwendet werden können. Es wird gezeigt, wie UML/P für Anwendungszwecke konfiguriert werden kann. In Kapitel 8 werden zunächst mögliche Verifikationsszenarien in Isabelle/HOL allgemein diskutiert, bevor einige konkrete Beispiele für Verifikation in Isabelle/HOL auf Basis konkreter UML/P-Modelle und ihrer Semantiken angegeben werden. Die Beispiele zeigen die praktische Durchführbarkeit und dienen auch der Plausibilisierung der semantischen Abbildungen.

Die Arbeit schließt im letzten Teil mit einer Zusammenfassung und einem Ausblick. Es werden Erweiterungspunkte durch zukünftige Arbeiten identifiziert. In Anhang A finden sich die verwendeten mathematischen Grundlagen, Anhang B enthält die Isabelle-Theorien des Systemmodells und dessen Varianten. Anhang C listet den syntaktischen Teil der UML/P, das heißt die MontiCore-Grammatiken, die zugehörige abstrakte Syntax in Isabelle/HOL sowie die syntaktischen Varianten. In Anhang D befindet sich ein Glossar. Abkürzungen werden in Anhang E aufgelistet. Der Index in Anhang F dient als Referenz für die Systemmodelldefinitionen und Varianten.

Kapitel 2

Grundlagen der Semantikbildung

Die formale Semantik einer Modellierungssprache legt mathematisch präzise die Bedeutung der Modelle der Sprache fest. Hierfür muss unter anderem auch die Syntax der Sprache präzise formuliert werden. Wir fassen zunächst stichpunktartig das Vorgehen zur Definition einer Modellierungssprache zusammen, bevor in den nachfolgenden Abschnitten eine detaillierte Diskussion folgt.

1. Definition der konkreten Syntax
2. Definition der abstrakten Syntax
3. Definition der Wohlgeformtheit
4. Optionale Reduktion der abstrakten Syntax
5. Definition der semantischen Domäne
6. Definition der semantischen Abbildung

Abschnitt 2.1 stellt die einzelnen Bestandteile der Definition einer Modellierungssprache vor. Jeder Bestandteil wird in einem Schritt des oben angegebenen Vorgehens behandelt. Die Semantikbildung erfolgt denotational auf Basis des Systemmodells (siehe hierzu auch [GRR09, BCGR08, CGR08a, HR04, Rum96]). Die sich hieraus ergebenden charakteristischen Eigenschaften des Ansatzes, wie beispielsweise die einfache semantische Integration mehrerer Modellierungssprachen, werden in Abschnitt 2.2 beschrieben. In Abschnitt 2.3 werden einige Notationskonventionen zur Strukturierung von Definitionen eingeführt, bevor in Abschnitt 2.4 ein erstes Beispiel einer vollständigen Sprachdefinition ohne Variabilität beschrieben wird. Das Beispiel veranschaulicht die einzelnen Vorgehensschritte und Sprachbestandteile. Abschnitt 2.5 dient der Einordnung und Abgrenzung unseres Ansatzes in Bezug auf verwandte Ansätze zur Definition der Semantik von Programmier- und Modellierungssprachen. Die Ausführungen in diesem Kapitel sind Voraussetzung für die in Kapitel 3 diskutierte Variabilität in Modellierungssprachen, die anhand der einzelnen Sprachbestandteile klassifiziert wird.

2.1 Bestandteile einer vollständigen Sprachdefinition

Abbildung 2.1 zeigt die Bestandteile einer formalen, vollständigen Definition einer Modellierungssprache unabhängig davon, ob es sich um eine textuelle oder graphische Sprache handelt.

Vollständig ist die Sprachdefinition in dem Sinn, dass sowohl Syntax als auch Semantik der Sprache thematisiert werden. Variabilität wird zunächst nicht betrachtet.

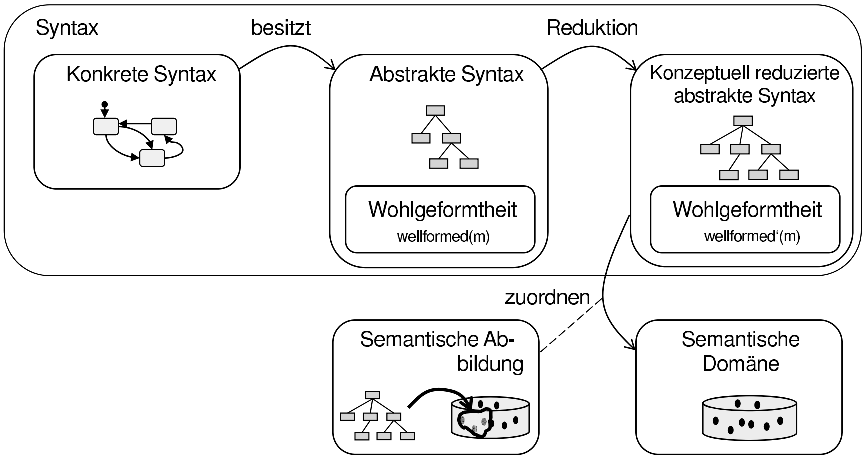


Abbildung 2.1: Bestandteile einer vollständigen Sprachdefinition

Konkrete Syntax Bei der konkreten Syntax einer Modellierungssprache handelt es sich um die Darstellung, mit der der Anwender interagiert, wenn er die Sprache verwendet. Die konkrete Syntax kann in textueller oder graphischer Notation oder einer Mischform vorliegen. Textuelle konkrete Syntax besteht aus Wörtern bzw. einer linearen Folge von Wörtern, die Sätze bilden. Eine graphische bzw. diagrammatische Syntax verwendet geometrische Elemente und Beziehungen wie Verbindungen oder Enthaltensein zwischen Elementen. Zudem werden Angaben über die Positionierung der Elemente gespeichert.

Bei der Definition einer konkreten textuellen Syntax kann auf bewährte Formalismen wie kontextfreie Grammatiken zurückgegriffen werden. Textuelle Modelle können dann mit einem beliebigen Editor bearbeitet werden. Für eine graphische Syntax existiert keine standardisierte Beschreibungsform, so dass ein graphischer Editor manuell programmiert, aus einer spezifischen Beschreibung (z.B. [GMF10]) abgeleitet oder ein generischer Editor (wie z.B. in MetaEdit+ [KT08]) angepasst werden muss. Bei der Definition der konkreten und abstrakten Syntax ist eine modulare Definition wünschenswert. Hierzu gehören Modularisierungskonzepte wie Sprachvererbung, bei der eine abgeleitete Sprache Konzepte einer Sprache wiederverwendet und Sprachparameter, wobei eine Sprache Konzepte verwendet, die erst bei Parameterbelegung durch eine zweite Sprache genau bestimmt werden. Eine konkrete Syntax besitzt eine abstrakte Syntax, die von bestimmten Informationen wie Schlüsselwörtern, Formatierung oder Layout abstrahiert und nur noch die essentiellen Konzepte der Sprache darstellt [Wil97].

Abstrakte Syntax Die abstrakte Syntax für textuelle Sprachen bildet typischerweise der von einem Parser erzeugte *Abstract Syntax Tree* (AST). Graphische Notationselemente sind normalerweise mit Elementen eines Metamodells (zum Beispiel einem Ecore oder MOF

Metamodell [BSM+03, OMG06a]) assoziiert. Eine alternative Darstellung wird später in Form einer Datentypdefinition in Isabelle/HOL eingeführt. Die Struktur der abstrakten Syntax kann bei grammatikbasierten Sprachen typischerweise automatisch gewonnen werden, kann aber bei Bedarf auch separat definiert werden. Dies ist der Normalfall für graphische Notationen, für die ein Metamodell meist manuell erstellt werden muss. Die separate Definition der abstrakten Syntax hat den methodischen Vorteil, dass sich als abstrakte Syntax ein im Projekt entwickeltes Datenmodell (Domänenmodell) direkt verwenden lässt. Zu diesem Domänenmodell können dann eine oder mehrere syntaktische Repräsentationsformen gebildet werden. Nachteil ist jedoch, dass Redundanz entsteht und die Konsistenz zwischen konkreter und abstrakter Syntax sichergestellt werden muss [KRV07b].

Wohlgeformtheit Meist sind noch zusätzliche Kontextbedingungen aufgrund der fehlenden Ausdrucksstärke von kontextfreien Grammatiken oder Metamodellen anzugeben. Die Bedingungen stellen die korrekte Verwendung von Sprachkonstrukten in ihrem Kontext und damit die Wohlgeformtheit eines Modells sicher. Wir stellen dies so dar, dass für eine Syntax $\mathcal{L}$ einer Sprache nur diejenigen Elemente der Syntax wohlgeformt sind, für die ein entsprechendes Prädikat

$$\mathrm{wellformed} : \mathcal{L} \to \mathrm{bool}$$

gilt (bool ist die Menge der Wahrheitswerte, vgl. Anhang A). Bei dieser Form der Kontextbedingung handelt es sich um eine *Intra-Kontextbedingung*. Das bedeutet, dass die Kontextbedingungen nur Wohlgeformtheit innerhalb der Sprache beschreiben. Im Gegensatz dazu gibt es auch *Inter-Kontextbedingungen*, die Wohlgeformtheit über Sprachgrenzen hinweg definieren. Wir machen jedoch in der Regel von der Möglichkeit Gebrauch, solche Kontextbedingungen als Intra-Kontextbedingungen zu formalisieren, indem zumindest konzeptuell eine integrierte Sprache $\mathcal{L}$ angenommen wird, die alle (unabhängigen) Teilsprachen beinhaltet.

Kontextbedingungen lassen sich auch direkt über Grammatiken definieren und Grammatiken können eingesetzt werden, um die abstrakte Syntax zu beschreiben. In dieser Arbeit verzichten wir auf diese Möglichkeiten und legen fest, dass eine Grammatik in erster Linie die konkrete Syntax beschreibt und Kontextbedingungen auf der abstrakten Syntax definiert werden.

Reduzierte abstrakte Syntax Es ist manchmal sinnvoll, die Anzahl der Sprachkonstrukte zu reduzieren, indem entweder die Struktur der abstrakten Syntax verändert wird oder Kontextbedingungen hinzugenommen werden, die die Verwendung bestimmter Konstrukte ausschließen. Konkrete Modelle in der vereinfachten Syntax haben meist eine länglichere und unübersichtlichere Darstellung. Eine andere Sichtweise ist, von einer reduzierten abstrakten Syntax ausgehend, Sprachkonstrukte hinzuzunehmen, die den Komfort bei der Modellierung oder die Ausdrucksfähigkeit der Sprache erhöhen.

Folgende Beispiele motivieren die Betrachtung einer reduzierten Syntax:

- Modellierungs- und Programmierrichtlinien (z.B. [Mat07, MIS10]) schränken die Sprache ein, indem zum Beispiel in Bezug auf Laufzeitfehler unsichere Konstrukte oder qualitativ schlechte Modelle verboten werden. Es wird jedoch im Allgemeinen nicht die Ausdrucksstärke eingeschränkt.

- Zusätzliche Konstrukte können zur Sprache hinzugenommen werden und erhöhen den Modellierungskomfort, sind aber durch bereits existierende Konstrukte erklärbar. Eine entsprechende syntaktische Transformation kann definiert werden.
- Eine Einschränkung der Ausdrucksstärke ist zum Beispiel dann sinnvoll, wenn Konstrukte auf einer speziellen Zielplattform nicht umsetzbar sind.
- Für syntaktisch sehr umfangreiche Sprachen kann es ebenfalls sinnvoll sein, die Anzahl der Sprachkonstrukte der Einfachheit halber zunächst echt einzuschränken, um zumindest die grundlegenden Konstrukte auch semantisch zu behandeln.
- Aus einer Sprache muss zunächst eine sinnvolle Teilmenge von Sprachkonstrukten bestimmt werden, da die Sprache insgesamt als „150%“-Sprache nicht verwendbar ist.

Dieser Teil der Sprachdefinition ist optional, man erreicht hierdurch jedoch eine einfachere Weiterverarbeitung und auch Semantikdefinition, da gegebenenfalls nicht mehr so viele und komplexe Konstrukte berücksichtigt werden müssen. Eine weitere Detaillierung der Anpassbarkeit einer Sprache findet in Kapitel 3 bei der Einführung von Sprachvarianten statt.

Semantische Domäne Die Modelle einer Modellierungssprache werden in Beziehung zu Elementen einer semantischen Domäne $\mathcal{S}$ gesetzt, die diese Modelle erklären. Hierdurch entsteht die Bedeutung der Sprache. Damit kommt der Wahl der semantischen Domäne eine wichtige Bedeutung zu, denn nur die in dieser Domäne vorhandenen Konzepte und Beziehungen stehen zur Verfügung, um die Bedeutung festzulegen. Die semantische Domäne bildet die Basis zur Erklärung einer zunächst noch unbekannten Bedeutung einer Syntax. Daher muss diese Domäne allgemein gut verstanden und akzeptiert sowie präzise definiert sein. Hier bietet es sich an, einen wohldefinierten, mathematisch präzisen Formalismus zu bemühen.

In dieser Arbeit basiert die semantische Domäne auf dem Systemmodell [BCGR08] und baut auf einfachen mathematischen Grundkonzepten wie Mengen, Relationen und Funktionen auf. Mit dem Systemmodell werden eine Reihe verhältnismäßig leicht verständlicher, jedoch umfangreicher Theorien aufgebaut, die wichtige Konzepte objektbasierter Modellierungssprachen formalisieren. Die Definition des Systemmodells ist unterspezifiziert und damit nicht direkt ausführbar, obwohl das Systemmodell objektbasierte Systeme mit ihren Strukturen, ihrem Verhalten und Interaktionen beschreibt. *Ein Element der semantischen Domäne entspricht einem eigenständigen objektbasierten System.* Dies führt auch zu einer verständlichen Semantik, da die Bedeutung eines Modells direkt als Eigenschaften einer möglichen Implementierung ausgedrückt wird. Unterspezifikation ist hierbei das Mittel, um die Implementierungsfreiheit zum Beispiel bei der Umsetzung des Modells in Code zu erhalten.

Das Systemmodell wird als einzige semantische Domäne für beliebige objektbasierte Modellierungssprachen eingesetzt, um später eine einfache Sprachintegration erreichen zu können. Hierdurch muss das Systemmodell allerdings eine gewisse Komplexität besitzen, da alle (häufig orthogonalen) Teile eines Systems mit Hilfe der Systemmodellkonzepte beschrieben werden können müssen. Ein Vorteil gegenüber anderen Formalismen ist jedoch, dass das Systemmodell nicht auf die Beschreibung eines bestimmten Aspekts eingeschränkt ist. Versteht man beispielsweise die Semantik von Assoziationskardinalitäten im Klassendiagramm als Menge von Ungleichungen, deren Lösbarkeit zu überprüfen ist (wie z.B. in [MB07]), ist zwar eine sehr kompakte semantische Domäne möglich, die aber offensichtlich nicht zur Abbildung anderer Aspekte geeignet ist.

Auch $\mathcal{S}$ hat eine syntaktische Repräsentation und damit eine Syntax $\mathcal{L}_{\mathcal{S}}$. In dieser Arbeit werden hierfür bekannte mathematische Notationen (siehe Anhang A) und später bei der Werkzeugunterstützung Theorien in Isabelles Logik höherer Stufe (HOL) [NPW02] verwendet.

Semantische Abbildung Eine semantische Abbildung setzt die Modelle einer Modellierungssprache mit Elementen der semantischen Domäne in Beziehung und legt damit ihre Bedeutung fest. Der in dieser Arbeit eingesetzte Ansatz verwendet eine für eine Syntax $\mathcal{L}$ und eine semantische Domäne $\mathcal{S}$ mengenwertige Abbildung der Form

$$\mathrm{sem} : \mathcal{L} \to \wp(\mathcal{S})$$

Alternativ wird auch eine prädikative Schreibweise der Form $\mathrm{sem} : \mathcal{L} \to \mathcal{S} \to \mathrm{bool}$ oder relationale Schreibweise $\mathrm{sem} \subseteq \mathcal{L} \times \mathcal{S}$ benutzt. Die Semantik eines Modells $m \in \mathcal{L}$ ist genau dann definiert, wenn m wohlgeformt ist, das heißt, $\mathrm{wellformed}\ m$ erfüllt ist. Die Semantik stellt dann eine Menge von Elementen der semantischen Domäne dar, das heißt, $\mathrm{sem}(m) \subseteq \wp(\mathcal{S})$. Im Folgenden gehen wir bei der Semantikbildung von der Wohlgeformtheit eines Modells aus, meist ohne diese Voraussetzung explizit zu erwähnen.

Als semantische Domäne $\mathcal{S}$ wird das Systemmodell eingesetzt. Für ein Modell $m \in \mathcal{L}$ ist die Semantik $\mathrm{sem}(m) \subseteq \wp(\mathrm{SystemModel})$ daher die Menge der möglichen Realisierungen. Jedes Element $s \in \mathrm{sem}(m)$ stellt ein eigenständiges objektbasiertes System dar.

Die Eigenschaften einer solchen mengenwertigen Abbildung in Verbindung mit einer einzigen semantischen Domäne werden in Abschnitt 2.2 beschrieben.

Genau wie bei der Definition der semantischen Domäne wird zur Definition der semantischen Abbildung eine Sprache $\mathcal{L}_{\mathrm{sem}}$ benötigt, die in dieser Arbeit ebenfalls bekannten mathematischen Notationen oder Theorien in Isabelle/HOL entspricht.

2.2 Charakteristische Eigenschaften des gewählten Ansatzes

Im Gegensatz zu Programmiersprachen werden Modellierungssprachen zur Spezifikation der Eigenschaften einer späteren Implementierung verwendet. Modelle können daher unterspezifiziert und unvollständig sein. Dies wird durch die Verwendung einer mengenwertigen Abbildung und des Systemmodells als semantische Domäne genau reflektiert: Je weniger Informationen ein Modell m enthält, desto größer ist die Menge der Systeme, die als Implementierung dienen können. Werden im Laufe der Zeit mehr Informationen zum Modell hinzugefügt und es entsteht das Modell m', verkleinert sich die Menge und es gilt $\mathrm{sem}(m') \subseteq \mathrm{sem}(m)$.

Des Weiteren finden häufig unterschiedliche Modellierungssprachen bei der Systemspezifikation Verwendung. In der UML gibt es beispielsweise 13 verschiedene Modellarten, die unter anderem die Struktur und das zustandsbasierte oder interaktionsbasierte Verhalten von Objekten beschreiben. Zusätzlich kann ein System durch mehrere Modelle derselben Modellart beschrieben sein. Verschiedene, sich gegebenenfalls überlappende Sichten auf ein System werden also modelliert. Eine Semantik muss dementsprechend die Integration dieser Sichten unterstützen. Durch die umfassende semantische Domäne Systemmodell und die mengenwertige Abbildung ist diese Integration einfach zu erreichen und es muss keine explizite, unter Umständen aufwendige Integration von Einzelsemantiken (insbesondere ihrer semantischen Domänen) durchgeführt werden.

Die Definition der integrierten Semantik einer Menge von Modellen $m_1 \in \mathcal{L}_1, \ldots, m_n \in \mathcal{L}_n$ von potentiell unterschiedlichen Modellarten erfolgt über die Schnittmengenbildung ihrer Semantiken:

$$\mathrm{sem}_{\mathcal{L}_1}(m_1) \cap \ldots \cap \mathrm{sem}_{\mathcal{L}_n}(m_n)$$

Ebenso intuitiv wird die Konsistenz eines oder mehrerer Modelle definiert durch

$$\bigcap_{\forall i} \mathrm{sem}_{\mathcal{L}_i}(m_i) \neq \varnothing$$

Ist die Spezifikation insgesamt inkonsistent, also widersprüchlich, enthält die Schnittmenge keine Systeme. Das bedeutet also, dass es keine Realisierung oder Implementierung der Modelle gibt, die alle spezifizierten Eigenschaften aufweist.

Die mengenwertige semantische Abbildung ist auch geeignet, syntaktische Kompositions- oder Verfeinerungsoperatoren zu untersuchen oder Operatoren mit bestimmten wünschenswerten Eigenschaften zu finden. Ein Kompositionsoperator $\oplus : \mathcal{L} \to \mathcal{L} \to \mathcal{L}$ ist beispielsweise gemäß [HKR$^+$07] vollständig eigenschaftserhaltend, falls gilt

$$\forall m_1, m_2 \in \mathcal{L} \,.\, \mathrm{sem}(m_1 \oplus m_2) = \mathrm{sem}(m_1) \cap \mathrm{sem}(m_2)$$

Ein Operator ist $\gamma : \mathcal{L} \to \mathcal{L}$ ist ein syntaktischer Verfeinerungsoperator, falls gilt

$$\forall m \in \mathcal{L} \,.\, \mathrm{sem}(\gamma(m)) \subseteq \mathrm{sem}(m)$$

2.3 Notation

Um eine gute Strukturierung der Definition einer Sprache zu erhalten, verwenden wir einige Konventionen. Definition 2.2 zeigt ein Beispiel, wie Theorien strukturiert werden. Sie ist aus mehreren Bestandteilen aufgebaut, die zur besseren Lesbarkeit optisch voneinander getrennt werden. Dabei können einzelne Bestandteile weggelassen oder mehrfach verwendet werden.

Definition 2.2 (Das ist eine Beispieldefinition)

NameDerDefinition

extend- und use-Referenzen, die auf andere Sprachbestandteile referenzieren

Einführung neuer Elemente (meist Mengen oder Abbildungen)

Notation:
Einführung von Notationskonventionen für neue Elemente

Definition von Eigenschaften

Informelle Erklärung zur Definition

Elemente, die über extend- und use-Referenzen eingebunden werden, können verwendet und erweitert werden. Allerdings werden importierte Elemente nur für extend-Referenzen auch reexportiert. Neben allgemeinen Definitionen werden solche Rahmen gesondert für jeden Sprachbestandteil definiert. Es wird also unterschieden, ob es sich bei der Definition um konkrete Syntax (KS), abstrakte Syntax (AS), Kontextbedingung (KB), semantische Domäne (SysMod) oder semantische Abbildung (SemAbb) handelt. Dabei muss ein Rahmen nicht unbedingt mathematische Definitionen beinhalten, sondern kann direkt Listings für Grammatiken (wie zum Beispiel in Definition 2.3 gezeigt) oder Isabelle/HOL-Theorien enthalten, wobei angegeben wird, um welche Art des Listings es sich handelt. So ist für jede Definition klar, auf welchen Sprachbestandteil sie sich bezieht. Ferner wird noch unterschieden, ob es sich um eine echte (invariante) Definition oder um eine Variante handelt. Alle Definitionen erhalten einen Namen und eine eindeutige Nummerierung für zukünftige Verweise. Damit ist sichergestellt, dass später eindeutig eine Menge von Definitionen als aktuell gültige Menge angesprochen werden kann. In allen Definitionen können die grundlegenden Konzepte aus Anhang A ohne explizite Referenz verwendet werden. In dieser Arbeit werden gelegentlich Abbildungen angegeben, die Diagramme oder Listings enthalten. Abbildungen sind im Gegensatz zu Definitionen nicht formal Bestandteil einer Sprachdefinition, sondern dienen nur zu Illustrationszwecken.

2.4 Beispiel einer Sprachdefinition

Als Illustration der wesentlichen Bestandteile einer Modellierungssprache und zur besseren Nachvollziehbarkeit der Diskussion in Kapitel 3 (Variabilität in Modellierungssprachen) wird an dieser Stelle eine vollständige Sprachdefinition anhand eines einfachen Beispiels demonstriert. In diesem Beispiel soll eine rudimentäre Modellierungssprache für UML-Klassendiagramme CDSimp eingeführt, das heißt ihre konkrete und abstrakte Syntax sowie ihre Semantik definiert werden. Eine Anpassung der abstrakten Syntax in eine reduzierte abstrakte Syntax ist aufgrund der Einfachheit der Sprache nicht notwendig.

2.4.1 Konkrete Syntax von CDSimp

Zur Definition der konkreten Syntax wird eine MontiCore-Grammatik eingesetzt. MontiCore [KRV08] und sein Grammatikformat werden noch detaillierter in den Kapitel 5 und 6 beschrieben. Die Definition 2.3 zeigt die MontiCore-Grammatik für die Sprache `CDSimp`.

Definition KS 2.3 (Einfache Klassendiagramme)

CDSimp

MontiCore-Grammatik

```
1   grammar CDSimp {
2
3    interface CDElem;
4
5    external Invariant;
6
7    CDDia = "classdiagram" DName:Name (CDElem | Invariant)*;
8
9    CDClass implements CDElem = "class" CName:Name
10             ("extends" (scl:Name "," scl:Name*))?
11             "{" (CDAttr ";")* "}";
12
13   CDAttr = Type:Name? AName:Name;
14
15   CDAssoc implements CDElem = From:Name "--" To:Name;
16  }
```

MontiCore-Grammatiken sind wie kontextfreie Grammatiken aus Regeln (oder Produktionen) aufgebaut. Die Regel `CDDia` (Zeile 7) gibt an, dass ein Klassendiagramm mit dem Schlüsselwort `classdiagram` beginnt und dann ein Diagrammname folgt (`Name` ist eine vordefinierte lexikalische Produktion für Namen bzw. Identifikatoren, vorangestellt und durch einen Doppelpunkt abgetrennt kann ein individueller Name für Produktionen vergeben werden). Schließlich werden eine Menge (Kleene Stern `*`) von Klassendiagrammelementen oder Invarianten (Alternativen werden durch `|` getrennt) erwartet. Die Regel `CDClass` implementiert die Schnittstelle (Interface) `CDElem` und legt fest, dass Klassen aus einem Klassennamen, einer Menge von Superklassen und einer Menge von Attributen bestehen. Attribute haben einen optionalen Bezeichner, der den Typ darstellt, und einen Namen. Assoziationen (Zeile 15) implementieren ebenfalls das Interface `CDElem`. MontiCore-Grammatiken unterstützen Schnittstellenregeln (Interfaces) wie in Zeile 3, die als Stellvertreter für alle die Schnittstelle implementierenden Regeln stehen. So wird statt `CDElem` in Zeile 7 entweder eine Klasse `CDClass` oder eine Assoziation `CDAssoc` erwartet. Die Regel `Invariant` ist nicht in der angegebenen Grammatik ausspezifiziert und mit dem Schlüsselwort `external` (Zeile 5) gekennzeichnet. Das bedeutet, dass für diese Regel eine beliebige zweite Sprache zur Beschreibung von Invarianten als Sprachparameter eingebettet werden kann.

Die angegebene Grammatik verwendet nicht alle in MontiCore-Grammatiken verfügbaren Konstrukte, ist aber ausreichend, das Beispiel nachvollziehbar zu verdeutlichen.

2.4.2 Abstrakte Syntax von CDSimp

In MontiCore wird die abstrakte Syntax immer automatisch aus der Grammatik in Form von Java-Klassen abgeleitet, weshalb auch wir nicht von einer unabhängigen Beschreibung der abstrakten Syntax als Datenmodell ausgehen wollen.

Eine MontiCore-Grammatik kann kanonisch mit Hilfe der folgenden Regeln auf eine einfache mengentheoretische Beschreibung der abstrakten Syntax abgebildet werden, siehe dazu auch [CGR08a] und [Kra10].

- Regeln werden auf Mengen abgebildet.
- Die rechte Seite einer Regel bildet ein Tupel, das die Struktur widerspiegelt.

Grammatik	`X = A B C`
Mengen	$X = A \times B \times C$

- Der optionale Name in Regeln wird ignoriert oder als Alias zur besseren Lesbarkeit eingeführt, sofern der Name eindeutig ist.

Grammatik	`X = a:A b:B k:C`
Mengen	$X = A \times B \times K$ und $K = C$

- Schnittstellenregeln 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 das Nichtvorhandensein darstellt. X^ϵ ist dabei die Kurzform für $X \cup \{\epsilon\}$.
- Wiederholungen durch einen Kleeneschen Stern und Listen der Form `A (',' A)*` werden durch die entsprechende Potenzmenge oder Liste dargestellt. Ob eine Darstellung als Menge oder Liste verwendet werden soll, hängt davon ab, ob Wiederholungen gleicher Elemente beibehalten werden sollen und ist jeweils einzeln zu entscheiden.

Grammatik	`X = A*` oder `X = (A (',' A)*)?`
Mengen	$X = \wp(A)$ oder $X = \mathrm{List}(A)$

- Regeln mit dem Schlüsselwort `external` werden in der Grammatik nicht definiert und bleiben auch in der Mengenrepräsentation unspezifiziert.

Zur besseren Lesbarkeit kann auf Komponenten eines Tupels mittels Punktnotation zugegriffen werden: Sei $X = A \times B$ und $x = (x_1, x_2) \in X$, so ist $x.A = x_1$ und $x.B = x_2$.

Die Anwendung der obigen Regeln führt zur Mengendarstellung der abstrakten Syntax für die Grammatik `CDSimp` in Definition 2.4. Die Schnittstelle *CDElem* wurde expandiert, das heißt durch ihre Definition ersetzt. Weiterhin erlauben wir, dass disjunkt vereinigte Mengen auch wieder separat notiert werden können. Das heißt, $\wp(\mathit{CDClass} \cup \mathit{CDAssoc} \cup \mathit{Invariant})$ kann durch $\wp(\mathit{CDClass}) \times \wp(\mathit{CDAssoc}) \times \wp(\mathit{Invariant})$ ersetzt werden. *Name* stellt eine nicht weiter spezifizierte Menge von Namen dar.

Definition AS 2.4 (Abstrakte Syntax für CDSimp)

CDSimpAS

$$\begin{array}{lcl}
\mathit{CDDia} & = & \mathit{DName} \times \wp(\mathit{CDClass}) \times \wp(\mathit{CDAssoc}) \times \wp(\mathit{Invariant}) \\
\mathit{CDClass} & = & \mathit{CName} \times \wp(\mathit{Scl}) \times \wp(\mathit{CDAttr}) \\
\mathit{CDAttr} & = & \mathit{Type}^{\epsilon} \times \mathit{AName} \\
\mathit{CDAssoc} & = & \mathit{From} \times \mathit{To} \\
\mathit{DName}, \mathit{CName}, \mathit{AName}, & & \\
\mathit{From}, \mathit{To}, \mathit{Scl} & = & \mathit{Name}
\end{array}$$

2.4.3 Kontextbedingungen von CDSimp

Wir geben nur eine Kontextbedingung 2.5 als Beispiel für Kontextbedingungen für CDSimp an, wonach der Klassenname jeder Klasse diagrammweit eindeutig sein muss.

Definition KB 2.5 (Klassennamen sind eindeutig)

CNameUnique

use CDSimpAS

$$\begin{array}{l}
\mathrm{cnameUnique} : \mathit{CDDia} \to \mathrm{bool} \\
\mathrm{cnameUnique}\ (\mathit{dname}, \mathit{classes}, \mathit{assoc}, \mathit{invars}) = \\
\quad (\forall\, c_1, c_2 \in \mathit{classes} : c_1.\mathit{CName} = c_2.\mathit{CName} \Rightarrow c_1 = c_2)
\end{array}$$

Weitere Kontextbedingungen ergeben sich beispielsweise aus der Forderung nach referentieller Integrität. Hierfür müssen zum Beispiel alle Superklassen einer Klasse oder Klassen, die in Assoziationen auftauchen, ebenfalls im Klassendiagramm definiert sein.

2.4.4 Semantische Domäne $\mathrm{SystemModel}_{\mathit{simp}}$

Das Systemmodell charakterisiert objektbasierte Systeme mit Hilfe einer Hierarchie von Theorien, die auf einfachen Grundlagen wie Mengen und Funktionen aufbaut. Ein Teil dieser Theorien befasst sich mit der Beschreibung der Struktur eines Systems. Die folgende Definition 2.6 ist ein vereinfachter Teil dieser Definitionen des Systemmodells aus Kapitel 4. Dieser Teil ist ausreichend, um für das gewählte Beispiel eine einfache semantische Abbildung anzugeben.

Definition 2.6 (Struktur des vereinfachten Systemmodells)

StruktSMsimp

$$\begin{array}{|l}
\mathrm{Name}\\
\mathrm{UCLASS}\\
\mathrm{UTYPE}\\
\mathrm{UVAR}\\
\mathrm{nameOf} : \mathrm{UCLASS} \rightarrow \mathrm{Name}\\
\mathrm{attr} : \mathrm{UCLASS} \rightarrow \wp(\mathrm{UVAR})\\
\mathrm{vname} : \mathrm{UVAR} \rightarrow \mathrm{Name}\\
\mathrm{vtype} : \mathrm{UVAR} \rightarrow \mathrm{UTYPE}\\
\mathrm{sub} : \mathrm{UCLASS} \rightarrow \mathrm{UCLASS} \rightarrow \mathrm{bool}\\
\hline
\end{array}$$

In Definition 2.6 werden die Universen (Mengen) UTYPE, UCLASS und UVAR von Typ-, Klassen- bzw. Attributnamen eingeführt. Die Funktion nameOf bzw. vname liefert für Klassen und Attribute ihren Namen aus einer Menge Name. attr liefert die Menge der Attribute einer Klasse und die Funktion vtype ergibt den Typ eines Attributs. Im Sinne der Unterspezifikation ist nichts ausgesagt über die genaue Struktur der Elemente der Universen oder die genaue Definition der Funktionen. Die Subklassenbeziehung zwischen Klassen wird durch sub abgebildet.

Definition 2.7 (Vereinfachtes Systemmodell)

SMsimp

$$\begin{array}{|l}
\mathrm{extend\ StruktSMsimp}\\
\hline
\mathrm{SystemModel}_{simp}\\
\hline
sm \in \mathrm{SystemModel}_{simp} \Rightarrow\\
\quad sm = (\mathrm{UCLASS}, \mathrm{UTYPE}, \mathrm{UVAR}, \mathrm{nameOf}, \mathrm{attr}, \mathrm{vname}, \mathrm{vtype}, \mathrm{sub})\\
\hline
\end{array}$$

Gemäß Definition 2.7 besitzt ein System konkrete Ausprägungen der Elemente aus Definition 2.6. Die Menge der Systeme im vereinfachten Systemmodell enthält formal also Tupel. Zur Selektion einer der Komponenten verwenden wir die Subskriptionsschreibweise. UCLASS_{sm} bezeichnet also die Menge der Klassen des Systems $sm \in \mathrm{SystemModel}_{simp}$. Die Menge der Namen Name ist nicht Teil des Tupels, da wir diese Menge als unveränderlich für alle Systeme betrachten wollen.

2.4.5 Semantische Abbildung von CDSimp

Entsprechend Abschnitt 2.2 definieren wir in Definition 2.8 eine mengenwertige semantische Abbildung *mCDDia*. Wie aus der Definition ersichtlich wird, kann *mCDDia*, der Struktur der abstrakten Syntax folgend, aus kleineren Funktionen (Prädikaten) zusammengesetzt werden, die dann immer von einem konkreten System *sm* abhängig sind. Eine Definition der semantischen Abbildung ist nur für Klassendiagramme (*mCDDia*), Klassen (*mapClass*) und Attribute (*mapAttr*) angegeben. Die Funktionen für die Abbildung von Assoziationen (*mapAssoc*), Superklassen von Klassen (*mapSuperClasses*), für Invarianten (*mapInvariant*), Namen (*mapName*) und Typen (*mapType*) sind zwar deklariert, aber nicht ausformuliert. Die semantische Abbildung der Invarianten hängt davon ab, welche Sprache für die Einbettung von Invarianten gewählt wurde. Alle Teile der semantischen Abbildung (außer *mCDDia*) sind mit einem System des Systemmodells parametrisiert, das den Kontext bildet.

Insgesamt erhalten wir eine Definition der Modellierungssprache CDSimp durch die Definitionen 2.3, 2.4, 2.5, 2.6, 2.7 und 2.8.

Definition SemAbb 2.8 (Semantische Abbildung für CDSimp)

mCDDiagram

$$\text{use CDSimpAS, SMsimp}$$

$$\begin{array}{l}
\mathit{mCDDia} : \mathit{CDDia} \to \wp(\text{SystemModel}_{\mathit{simp}})\\
\mathit{mapClass}_{\text{SystemModel}_{\mathit{simp}}} : \mathit{CDClass} \to \text{bool}\\
\mathit{mapAssoc}_{\text{SystemModel}_{\mathit{simp}}} : \mathit{CDAssoc} \to \text{bool}\\
\mathit{mapInvar}_{\text{SystemModel}_{\mathit{simp}}} : \mathit{Invariant} \to \text{bool}\\
\mathit{mapSuperClasses}_{\text{SystemModel}_{\mathit{simp}}} : \text{UCLASS}_{\text{SystemModel}_{\mathit{simp}}} \to \wp(\mathit{Name}) \to \text{bool}\\
\mathit{mapAttr}_{\text{SystemModel}_{\mathit{simp}}} : \text{UCLASS}_{\text{SystemModel}_{\mathit{simp}}} \to \mathit{CDAttr} \to \text{bool}\\
\mathit{mapType}_{\text{SystemModel}_{\mathit{simp}}} : \text{UTYPE}_{\text{SystemModel}_{\mathit{simp}}} \to \mathit{Name} \to \text{bool}\\
\mathit{mapName}_{\text{SystemModel}_{\mathit{simp}}} : \mathit{Name} \to \text{Name}
\end{array}$$

$$\begin{array}{l}
\mathit{mCDDia}\ \mathit{cd} = \{\mathit{sm} \in \text{SystemModel}_{\mathit{simp}} \mid\\
\quad \forall\, \mathit{cdc} \in \mathit{cd.CDClass} : \mathit{mapClass}_{\mathit{sm}}\ \mathit{cdc} \wedge\\
\quad \forall\, \mathit{cda} \in \mathit{cd.CDAssoc} : \mathit{mapAssoc}_{\mathit{sm}}\ \mathit{cda} \wedge\\
\quad \forall\, \mathit{inv} \in \mathit{cd.Invariant} : \mathit{mapInvar}_{\mathit{sm}}\ \mathit{inv}\}
\end{array}$$

$$\begin{array}{l}
\mathit{mapClass}_{\mathit{sm}}\ \mathit{cl} =\\
\quad (\exists\, C \in \text{UCLASS}_{\mathit{sm}} :\\
\qquad \text{nameOf}_{\mathit{sm}}\, C = \mathit{mapName}_{\mathit{sm}}\ \mathit{cl.CName} \wedge\\
\qquad \mathit{mapSuperClasses}_{\mathit{sm}}\ C\ \mathit{cl.Scl} \wedge\\
\qquad \forall\, \mathit{cdat} \in \mathit{cl.CDAttr} : \mathit{mapAttr}_{\mathit{sm}}\ C\ \mathit{cdat})
\end{array}$$

$$\begin{array}{l}
\mathit{mapAttr}_{\mathit{sm}}\ C\ \mathit{cdat} =\\
\quad (\exists\, \mathit{at} \in \text{attr}_{\mathit{sm}}\, C :\\
\qquad \text{nameOf}_{\mathit{sm}}\, \mathit{at} = \mathit{mapName}_{\mathit{sm}}\ \mathit{cdat.AName} \wedge\\
\qquad \mathit{cdat.Type} \neq \epsilon \Rightarrow \mathit{mapType}_{\mathit{sm}}\ (\text{vtype}_{\mathit{sm}}\ \mathit{at})\ \mathit{cdat.Type})
\end{array}$$

2.5 Einordnung des Ansatzes und verwandte Arbeiten

Eine vielfach verwendete grundlegende Klassifikation von Ansätzen zur formalen Semantikdefinition unterteilt diese in denotationale, operationale und axiomatische Semantik [Win93].

Bei der denotationalen Semantik wird die Syntax einer Sprache durch eine mathematische Funktion auf eine semantische Domäne abgebildet. Bei der Semantik einfacher Programmiersprachen finden sich hier Funktionen über abstrakten Zuständen (im einfachen Fall entsprechen die Zustände Variablenbelegungen), die eine *Scott Domain* [SS71] bilden. Rekursion kann dabei durch Fixpunktbildung beschrieben werden. Dieser Ansatz funktioniert besonders gut bei sequentiellen Sprachen, deren Semantik kompositional entlang der abstrakten Syntax gebildet wird. Für komplexere Sprachen, beispielsweise mit Konstrukten für Kontrollflusssprünge oder Nichtdeterminismus und Parallelität, gibt es Erweiterungen wie zum Beispiel *continuations* (siehe z.B. [NN99]), die jedoch eine deutlich komplexere semantische Domäne zur Folge haben.

Operationale Semantik gilt als weniger abstrakt und beschreibt die Bedeutung eines Modells als mögliche Ausführung auf einer abstrakten Maschine in Form von Berechnungsschritten. *Structured Operational Semantics* (SOS) oder small-step Semantik [Plo81] wird durch Inferenzregeln angegeben, die einen Ausdruck der abstrakten Syntax in ein Zwischenergebnis und den restlichen, noch auszuwertenden Ausdruck überführen. Im Gegensatz dazu beschreibt *natürliche Semantik* oder big-step Semantik [Kah87] mit Hilfe von Inferenzregeln gleich das Endergebnis der Auswertung eines Ausdrucks. Ein weiterer wichtiger Vertreter der operationalen Semantik sind *Abstract State Machines* (ASM) [BS03]. Sie stellen Transitionssysteme dar, in denen Zustände als Algebra beschrieben werden, die typischerweise die abstrakte Syntax gleich mit enthalten, siehe zum Beispiel [BCR04]. Die dynamische Semantik wird dann über Aktualisierungsregeln angegeben, die die Algebra dynamisch verändern (der ursprüngliche Name für ASM war bezeichnenderweise *evolving algebras* [Gur95]).

Bei der axiomatischen Semantik werden Regeln definiert, mit Hilfe derer die partielle Korrektheit von Programmen nachgewiesen werden kann. Die Hoare Logik [Hoa69] mit Vor- und Nachbedingungen ist das bekannteste Beispiel für axiomatische Semantik. In einer etwas weiter gefassten Definition können auch syntaktische Transformationsschritte (zum Beispiel wie die in Abschnitt 2.1 erwähnte Umformung eines Modells in eine reduzierte Darstellung) als Regeln einer axiomatischen Semantik gesehen werden. Eine weitergehende ausführliche Diskussion aktueller Semantikansätze ist in [ZX04] zu finden.

Grundsätzlich wird in dieser Arbeit ein denotationaler Semantikansatz verfolgt. Statt einer einfachen Domäne (z.B. Funktion über Variablenbelegungen) wird die komplexe Domäne Systemmodell eingesetzt. Da das Systemmodell potentielle Implementierungen und deren Ablauf zum Teil sehr feingranular beschreibt, ist auch eine Verwandtschaft zur operationalen Semantik erkennbar. Das Systemmodell kann sogar, wie in [Cra09, CD08a], eher operational eingesetzt werden. Hierbei geht allerdings im Allgemeinen die Fähigkeit Unterspezifikation abzubilden verloren. Im Fall der Definition der Semantik von Aktionen kann dies aber sinnvoll sein, da z.B. die Semantik einer Zuweisung genau festgelegt und nicht unterspezifiziert werden soll.

Eine charakteristische Eigenschaft des Ansatzes ist eine mengenwertige semantische Abbildung, die aus [Rum96] übernommen wurde. Dies entspricht der Idee der losen Semantik aus der Theorie algebraischer Spezifikationen [BW82, BFG^{+}93], wobei *alle* Elemente der semantischen

Domäne als Semantik eines Modells dienen, die den explizit geforderten Bedingungen aus der semantischen Abbildung entsprechen. Weitere, nicht explizit formulierte Eigenschaften werden nicht angenommen. Die Idee der losen Semantik wird auch in [Huß02] bei der Semantikdefinition von UML-Klassendiagrammen und OCL auf Basis von *object algebras* benutzt. [Szl06] verwendet eine mengenwertige semantische Abbildung zur Semantikdefinition einfacher, konzeptueller Klassendiagramme. Die Semantik von verhaltensbeschreibenden Modellierungssprachen wird vornehmlich über operationale Ansätze definiert.

Denotationale Semantik erlaubt eine klare Trennung von Syntax, semantischer Domäne und semantischer Abbildung. Allgemein messen andere Arbeiten zur formalen Semantik von Modellierungssprachen dieser Trennung keine große Bedeutung zu, da der Hauptfokus auf der semantischen Abbildung und Domäne liegt. So findet man häufig eine sehr vereinfachte Darstellung der Syntax, z.B. als kontextfreie Grammatik wie in [CK04], die zwar die Grundkonzepte enthält, aber nicht für einen Anwender geeignet ist. Oder es findet eine Vermischung der Syntax mit der Semantik statt. Beispiele hierfür sind die bereits erwähnte ASM-basierte Arbeit [BCR04], sowie [Bee02] oder [FS06], bei denen eine erweiterte abstrakte Syntax gleichzeitig dazu dient, Konfigurationen oder Zustände in einem Systemablauf und damit die Semantik darzustellen.

Auch im Hinblick auf die später eingeführte Werkzeugunterstützung wird in dieser Arbeit zwischen konkreter und abstrakter Syntax, semantischer Domäne und Abbildung unterschieden und diese Sprachbestandteile explizit getrennt. Ein Vorteil dieser klaren Trennung ist die Möglichkeit zur systematischen Klassifikation von Variabilität in einer Sprachdefinition (siehe Kapitel 3) entsprechend ihrem Auftreten in einem dieser Bestandteile.

Kontextbedingungen, das heißt, Wohlgeformtheitsbedingungen, die eine kontextfreie Grammatik nicht ausdrücken kann, werden häufig auch als statische Semantik [ALSU86] bezeichnet. Der Begriff stammt aus dem Übersetzerbau und wird hier nicht verwendet, da es sich bei Kontextbedingungen allein um syntaktische Eigenschaften eines Modells handelt. Hierzu gehört auch die Frage nach der korrekten Typisierung der Modellelemente. Typtheorie steht nicht im Fokus dieser Arbeit, das heißt, die Wohlgeformtheit von Modellen bezüglich Typisierung wird meist nicht explizit formuliert.

Die mengenwertige semantische Abbildung und die Verwendung einer einzigen semantischen Domäne erlaubt die Integration heterogener Modellierungssprachen. Dabei werden nur solche Sprachen betrachtet, die sich mit Hilfe des Systemmodells beschreiben lassen. Der Heterogenität von Sprachen sind Grenzen in dem Sinne gesetzt, dass nur zeitdiskrete Systeme beschrieben werden können. Obwohl Erweiterungen denkbar sind, werden Betrachtungen von kontinuierlichen Systemen wie in [EJL$^+$03] nicht angestellt. In [GR03] wird ebenfalls die Idee einer gemeinsamen Domäne (Transformationssysteme) zur semantischen Integration mehrerer Modellierungssprachen vorgeschlagen. Jedoch wird jedes Modell in ein „lokales“ Transformationssystem übersetzt, die Integration erfolgt dann explizit über eine geeignete Komposition der Transformationssysteme.

Im Vergleich zu anderen Ansätzen zur semantischen Integration mehrerer Modellierungssprachen kann eine nachträgliche Integration von semantischen Domänen vermieden werden, indem eine ausreichend vielseitige Domäne im Vorhinein definiert wird. Im Gegensatz dazu beruht die Integration in [CKTW08] beispielsweise auf der Komposition von Institutionen, die jeweils eine natürliche semantische Domäne einer Diagrammart der UML darstellen. In [KGKZ09] werden Graphtransformationssysteme für eine Reihe von UML-Diagrammarten spezifisch kombiniert, so dass eine fest definierte integrierte Semantik für genau die betrachteten Diagrammarten entsteht, wobei Unterspezifikation nicht erlaubt ist. Verwandte Arbeiten, die Variabilität bei der Definition einer Modellierungssprache unterstützen, werden im nächsten Kapitel gesondert diskutiert. Ebenso werden Referenzen auf verwandte Arbeiten zur Semantik einzelner UML-Diagrammarten bei der Semantikdefinition in Kapitel 7 angegeben.

Kapitel 3

Variabilität in Modellierungssprachen

Eine vollständige Definition einer Modellierungssprache soll zwar präzise aber nicht unbedingt vollständig fixiert sein, um die Möglichkeit zur Anpassung an einen Projektkontext, eine Problemdomäne oder Zielplattform nicht zu verlieren. Die Forderung nach einer solch flexiblen Definition (für UML) wurde bereits früh schon erhoben [CKW$^+$99]. In [O'K06] werden Kriterien für eine verbesserte Definition der UML aufgestellt. Neben einer präzisen Definition der Sprachbestandteile (vgl. Kapitel 2) wird insbesondere auch die Möglichkeit zur präzisen Behandlung semantischer Varianten gefordert. Die Flexibilität bei der Interpretation bestimmter Konstrukte kann auch genutzt werden, aus einem „kleinsten gemeinsamen Nenner“ verschiedene Varianten der Sprache zu bilden, um unterschiedlichen Stakeholdern mit ihren spezifischen Interessen Rechnung zu tragen.

Der UML-Standard definiert beispielsweise vor diesem Hintergrund für alternative konkrete Repräsentationen so genannte „presentation options“. Anpassungen der Semantik sind durch „semantic variation points“ möglich. Leider ist die Variabilität von UML nicht systematisch festgelegt und unvollständig: Obwohl es bei der Einführung von Metamodellelementen jeweils explizite Unterabschnitte zu semantischen Variationspunkten gibt, sind dort nicht immer alle semantisch unterspezifizierten Elemente aufgeführt. Bei der Beschreibung einer *StateMachine* [OMG09b, Abschnitt 15.3.12] wird die Verarbeitung von eintreffenden Ereignissen ausdrücklich offen gelassen, jedoch nicht als semantischer Variationspunkt aufgeführt. Viele ähnliche Beispiele lassen sich im UML-Standard finden. Der UML-Standard selbst spricht hier von „weniger präzise definierten Dimensionen“ der Sprache [OMG09b, Abschnitt 2.3].

Wir stellen die Hypothese auf, dass eine systematische Definition der Variabilität hilft, eine Modellierungssprache präzise und gleichzeitig flexibel zu definieren. Werkzeughersteller werden so in die Lage versetzt, sich bei Konformitätserklärungen wesentlich genauer auf bestimmte Varianten der Sprache zu beziehen, um so ihre Entwurfsentscheidungen zu dokumentieren. Bei der Zusammenstellung einer Werkzeugkette können Benutzer dann darauf achten, dass die Werkzeuge so konfiguriert werden können, dass eine kompatible Verwendung möglich ist. Dem Vorteil der Flexibilität und der systematischen Erfassung und Konfiguration der Varianten einer Modellierungssprache steht als Nachteil die große Zahl der Varianten und damit die Komplexität einer vollständigen Variantendefinition für umfangreichere Modellierungssprachen gegenüber. Die in Kapitel 5 eingeführte Werkzeugunterstützung soll unter anderem helfen, dieser Komplexität Herr zu werden.

In Abschnitt 3.1 bauen wir eine allgemeine Klassifikation der Variabilität in Modellierungssprachen auf. Die Definition und Konfiguration von Varianten auf der Basis von Feature-Diagrammen wird in Abschnitt 3.2 beschrieben. In Abschnitt 3.3 beschreiben wir anhand der Klassifikation das Vorgehen bei Änderungen an einer Sprachdefinition. Abschließend werden verwandte Arbeiten in Abschnitt 3.4 besprochen und das Kapitel in Abschnitt 3.5 zusammengefasst. Die wesentlichen Inhalte dieses Kapitels wurden auch in [CGR09, GR10] veröffentlicht.

3.1 Klassifikation von Variabilität in einer Sprachdefinition

Die Klassifikation von Variabilität in Sprachdefinitionen wird anhand der Bestandteile einer Sprachdefinition gemäß Kapitel 2 aufgebaut. Zur formalen Darstellung der Zusammenhänge werden zunächst einige Bezeichnungen vereinbart. Anstatt allgemein von einer Sprache $\mathcal{L}$ zu sprechen, bezeichnen wir die Menge der Modelle einer Sprache in konkreter Syntax mit $\mathcal{KS}$. Sie werden überführt in ihre Repräsentation in abstrakter Syntax $\mathcal{AS}$. Die Menge der wohlgeformten Modelle wird als $\mathcal{AS}^{wf}$ bezeichnet und ergibt sich aus allen Modellen, für die das Prädikat $\mathrm{wellformed}$ erfüllt ist, also $\mathcal{AS}^{wf} = \{m \in \mathcal{AS} \mid \mathrm{wellformed}(m)\}$. Da nicht alle Modelle aus $\mathcal{KS}$ wohlgeformt sind, ergibt sich eine partielle Abbildung $\mathrm{p} : \mathcal{KS} \rightharpoonup \mathcal{AS}^{wf}$ zur Überführung der konkreten in die abstrakte Syntax.

Die reduzierte abstrakte Syntax, $\mathcal{AS}^{red} \subseteq \mathcal{AS}^{wf}$, schränkt die Menge der wohlgeformten Modelle einer Modellierungssprache weiter ein. Für einige Modelle gibt es eine syntaktische Transformation $\mathrm{t} : \mathcal{AS}^{wf} \rightharpoonup \mathcal{AS}^{red}$ in die reduzierte abstrakte Syntax.

Die semantische Abbildung sem ordnet schließlich jedem Modell der reduzierten abstrakten Syntax ihre Bedeutung als Menge von Elementen der semantischen Domäne $\mathcal{S}$ zu, das heißt $\mathrm{sem} : \mathcal{AS}^{red} \rightarrow \wp(\mathcal{S})$.

Ausgehend von dieser abstrakten Sichtweise der Bestandteile einer Modellierungssprache betrachten wir nun Möglichkeiten der Anpassung einer Sprache. Variantenbildung erfolgt also durch Veränderungen an Elementen der Abfolge

$$\mathcal{KS} \overset{\mathrm{p}}{\rightharpoonup} \mathcal{AS}^{wf} \overset{\mathrm{t}}{\rightharpoonup} \mathcal{AS}^{red} \overset{\mathrm{sem}}{\rightarrow} \wp(\mathcal{S})$$

In den folgenden Abschnitten detaillieren wir die Mechanismen zur Definition von Variabilität für die Sprachbestandteile. Die Tabelle 3.1 enthält bereits eine Zusammenfassung unserer Klassifikation.

3.1.1 Präsentationsvariabilität

In der konkreten Syntax $\mathcal{KS}$ können unterschiedliche Modelle mittels p auf dieselbe abstrakte Syntax des Modells abgebildet werden. Ein Beispiel ist in der Abbildung 3.2 angegeben. Die beiden dargestellten Modelle enthalten jeweils das Klassendiagramm `M` mit einer öffentlichen und einer privaten Klasse, wobei unterschiedliche Schlüsselwörter hierfür verwendet werden. Außerdem ist das erste Diagramm durch Leerzeilen und Einrückung übersichtlicher gestaltet. Obwohl hier die abstrakte Syntax nicht explizit angegeben ist, ist klar, dass beide Diagramme denselben Inhalt auf unterschiedliche Weise darstellen. Dies wird als Präsentationsoption bezeichnet. Formal ausgedrückt besitzt eine konkrete Syntax Präsentationsoptionen, falls gilt

$$\exists\, m_1, m_2 \in \mathcal{KS} : m_1 \neq m_2 \wedge p(m_1) = p(m_2)$$

Tabelle 3.1: Klassifikation der Variabilität in Modellierungssprachen

Präsentationsvariabilität in	Variabilität, die nicht in einer reduzierten abstrakten Syntax $\mathcal{AS}^{red}$ erhalten bleibt
Präsentationsoptionen	Wirken nur auf konkrete Syntax $\mathcal{KS}$
Abkürzungen	Können zu $\mathcal{AS}^{wf}$ hinzugenommen werden, ohne die Ausdrucksmächtigkeit (die Menge $\mathcal{AS}^{red}$) zu verändern
Syntaktische Variabilität in	Variabilität, die in einer reduzierten abstrakten Syntax $\mathcal{AS}^{red}$ erhalten bleibt
Stereotypen	Syntaktische Kodierung von semantischer Variabilität in $\mathcal{AS}^{red}$
Sprachparameter	Einbettung unabhängiger Teilsprachen in $\mathcal{AS}^{red}$
Spracheinschränkungen /-erweiterungen	Schränken die Menge der wohlgeformten Modelle $\mathcal{AS}^{red}$ ein bzw. erweitern diese durch Veränderung der Syntax. Einschränkungen können auch durch optionale Kontextbedingungen erreicht werden.
Semantische Variabilität in	Variabilität in der Semantik
Semantischer Domäne	Die zugrunde liegende Domäne $\mathcal{S}$
Semantischer Abbildung	Alternative Realisierungen der Abbildung sem

Varianten von Präsentationsoptionen führen zur Anpassung von $\mathcal{KS}$ und p, z.B. $\mathcal{KS}_v$ und p_v. Dabei haben gemeinsame Modelle weiterhin die gleiche abstrakte Syntax:

$$\forall\, m \in \mathcal{KS}_v \cap \mathcal{KS} : \mathrm{p}_v(m) = \mathrm{p}(m)$$

Außerdem kann jedes Modell auch ohne die Variante der Präsentationsoption ausgedrückt werden:

$$\forall\, m_1 \in \mathrm{dom}(\mathrm{p}_v) : \exists\, m_2 \in \mathrm{dom}(\mathrm{p}) : \mathrm{p}_v(m_1) = \mathrm{p}(m_2)$$

Varianten von Präsentationsoptionen werden als Präsentationsvariabilität klassifiziert. Damit lässt sich die Lesbarkeit der Modelle verbessern, ohne die abstrakte Syntax zu ändern. Ihr Einflussbereich erstreckt sich also ausschließlich auf die konkrete Syntax. In UML wird solche

M

Klassendiagramm

```
classdiagram M
{

  public class A;

  private class B;

}
```

M

Klassendiagramm

```
classdiagram M {
+ class A; - class B;
}
```

Abbildung 3.2: Zwei Darstellungen eines Klassendiagramms

Variabilität ebenfalls *presentation option* genannt. Weitere Beispiele sind Schriftgrößen, Linienstärken oder Farben graphischer Modellelemente, die zu keiner Änderung der abstrakten Syntax führen. Das ist eine wichtige Annahme, die für Präsentationsoptionen gelten muss, nämlich dass durch Präsentationsoptionen keine Veränderung der abstrakten Syntax und damit auch keine Veränderung der Semantik erreicht werden kann. Spielt beispielsweise die Farbe von Modellelementen (wie etwa in Life Sequence Charts [DH01]) bei der Interpretation eine Rolle, darf dies natürlich nicht als Präsentationsoption eingestuft werden. Es ist zudem wichtig, die Präsentationsoptionen sorgfältig zu dokumentieren, damit nicht beispielsweise in einem Werkzeug eines bestimmten Werkzeugherstellers eine Bedeutung assoziiert wird.

Die Syntax kann weitere Konstrukte enthalten, um die Lesbarkeit zu erhöhen, die die Ausdrucksmächtigkeit der Sprache bei Hinzunahme oder Weglassen der Konstrukte aber nicht verändern. Diese Konstrukte lassen sich also mit Hilfe anderer (typischerweise primitiverer) Konstrukte ausdrücken und können demnach vernachlässigt werden. Insbesondere kann die abstrakte Syntax in eine reduzierte abstrakte Syntax, $\mathcal{AS}^{red} \subseteq \mathcal{AS}^{wf}$, überführt werden, in der bestimmte *Abkürzungen* oder *Präsentationserweiterungen* nicht mehr existieren. Modelle mit Abkürzungen können durch die Transformation t behandelt werden. Die Transformation t ändert dabei nichts an bereits reduzierten Modellen, das heißt

$$\forall m \in \mathcal{AS}^{red} : t(m) = m$$

Modelle, die echte Abkürzungen enthalten und tatsächlich durch die Transformation geändert werden, sind in $\mathrm{dom}(\mathrm{t}) \backslash \mathcal{AS}^{red}$ enthalten.

Nach [Rum04b] ist es beispielsweise möglich, die Hierarchie in Statecharts semantikerhaltend zu entfernen. Anders ausgedrückt können wir von einer reduzierten abstrakten Syntax ausgehend verschiedene Abkürzungen (wie z.B. Hierarchie in Statecharts) in die Sprache aufnehmen, ohne ihre Semantik zu ändern. Hinzunehmen einer Abkürzung bedeutet gleichzeitig das Definieren einer syntaktischen Transformation, die diese Abkürzung wieder entfernt, so dass die reduzierte abstrakte Syntax unverändert bleiben kann. Variabilität in Abkürzungen bedeuten also Anpassungen von $\mathcal{AS}^{wf}$ und t, z.b. $\mathcal{AS}^{wf}_{v}$ und t_v. Gemeinsame Modelle müssen dabei gleich transformiert werden, das heißt,

$$\forall m \in \mathrm{dom}(\mathrm{t}) \cap \mathrm{dom}(\mathrm{t}_v) : \mathrm{t}(m) = \mathrm{t}_v(m)$$

Weiterhin lässt sich jedes Modell auch ohne die Variante der Abkürzungen ausdrücken

$$\forall m_1 \in \mathrm{dom}(\mathrm{t}_v) : \exists m_2 \in \mathrm{dom}(\mathrm{t}) : \mathrm{t}_v(m_1) = \mathrm{t}(m_2)$$

Ähnlich wie Präsentationsoptionen, die nicht mehr in der abstrakten Syntax auftauchen, sind Abkürzungen nicht mehr in der reduzierten abstrakten Syntax zu finden (werden entfernt oder durch zusätzliche Kontextbedingungen ausgeschlossen). Abkürzungen werden mit Hilfe der Transformation t erklärt. Dies lässt uns Abkürzungen ebenfalls als Präsentationsvariabilität klassifizieren. Es sollte erwähnt werden, dass Präsentationsoptionen und Abkürzungen selbst geeignet sind, um auf der Ebene der Modelle Varianten zu erstellen, es nämlich erlauben, dasselbe Modell auf unterschiedliche Weisen darzustellen. Dies stellt in unserer Klassifikation keine Form der Variabilität dar, da wir Variantenbildung auf Sprachebene und nicht auf Ebene konkreter Modelle betrachten.

Es ist auch möglich, einen Sprachkern, also eine minimale Syntax $\mathcal{AS}^{min}$ zu erreichen, wobei alle Sprachkonstrukte entfernt wurden, die sich mittels anderer Konstrukte ausdrücken lassen. Diese minimale Syntax ist im Allgemeinen nicht eindeutig. So kann bekanntermaßen eine minimale syntaktische Basis für bool'sche Ausdrücke in der Aussagenlogik beispielsweise $\{\neg, \wedge\}$ oder ebenso $\{\neg, \vee\}$ sein. Weiterhin ist zu beachten, dass selbst eine minimale Syntax noch *Synonyme* enthalten kann, das heißt, syntaktisch unterschiedliche Modelle m_1 und m_2 mit derselben Semantik $\mathrm{sem}(m_1) = \mathrm{sem}(m_2)$. Dies stellt ebenfalls in unserer Klassifikation keine Form der Variabilität der Sprache dar, sondern ist beispielsweise kommutativen Operatoren geschuldet. Es ist außerdem nicht unbedingt notwendig, die abstrakte Syntax zu einer minimalen abstrakten Syntax umzuformen. Es kann im Gegenteil gewollt sein, dass die Sprache weiterhin Abkürzungen enthält. Denn dann kann auf Basis der Semantik nachgewiesen werden, dass eine Transformation, die bestimmte Abkürzungen entfernt, tatsächlich semantikerhaltend ist. Ist beispielsweise für $\mathcal{AS}^{red} \supseteq \mathcal{AS}^{min}$ eine Transformation $\mathrm{t}' : \mathcal{AS}^{red} \rightarrow \mathcal{AS}^{min}$ definiert, so lässt sich formal zeigen, dass t' semantikerhaltend wirkt:

$$\forall m \in \mathcal{AS}^{red} : \mathrm{sem}(\mathrm{t}'(m)) = \mathrm{sem}(m)$$

Auf $\mathcal{AS}^{min}$ ist t' die Identität, das heißt

$$\forall m \in \mathcal{AS}^{min} : \mathrm{t}'(m) = m$$

Zusammenfassend klassifizieren wir diejenige Variabilität als Präsentationsvariabilität, die spätestens beim Übergang zu einer reduzierten abstrakten Syntax verschwindet. Dazu zählen Varianten in Präsentationsoptionen und Abkürzungen.

3.1.2 Syntaktische Variabilität

Die Auswahl syntaktischer Varianten hat eine Auswirkung auf die reduzierte abstrakte Syntax $\mathcal{AS}^{red}$. Die Syntax einer Sprache kann flexibel mit so genannten *Stereotypen* erweitert werden. Dieser aus der UML stammende Begriff wird hier wie in [Rum04b, FPR02] verwendet, um ein allgemeines Prinzip der Erweiterung einer Syntax zu bezeichnen: Bestimmte Modellelemente können mit Stereotypen annotiert werden, wobei zunächst nicht festgelegt ist, welche Stereotypen verwendet werden dürfen. Abbildung 3.3 zeigt ein Klassendiagramm, in dem die Klasse `Manager` mit dem Stereotyp `<<singleton>>` annotiert ist. Gemäß dem Entwurfsmuster „Singleton" [GHJV95] soll diese Klasse nur höchstens einmal im System instantiiert werden, was über die herkömmliche Syntax der einfachen Klassendiagramme (vgl. Abschnitt 2.4) nicht möglich wäre. Die Festlegung konkreter Teilmengen von Stereotypen, die zusammen in einer Sprachvariante verwendet werden können, wird als syntaktische Variabilität klassifiziert. Solch eine Menge von Stereotypen wird im UML-Wortgebrauch auch als Profil bezeichnet. Formal können wir dies so darstellen, dass wohlgeformte Modelle in einer Variante $\mathcal{AS}^{red}_v$ der reduzierten abstrakten Syntax zusätzlich nur die in der Variante ausgewählten Stereotypen verwenden:

$$\mathcal{AS}^{red}_v = \{m \in \mathcal{AS}^{red} \mid \mathrm{allowedStereotypes}_v(m)\}$$

In der Funktionsvariante $\mathrm{allowedStereotypes}_v$ ist also kodiert, welche Stereotypen für welche Modellelemente verwendet werden dürfen.

Sys

Klassendiagramm

```
classdiagram Sys {

  <<singleton>> class Manager;

}
```

Abbildung 3.3: Beispielmodell mit Stereotyp

Eine weitere Form der syntaktischen Variabilität sind so genannte *Sprachparameter*, der Mechanismus wird auch als Spracheinbettung bezeichnet [Kra10, KRV08]. Die Syntax besitzt also eine Reihe von Parametern $\mathcal{AS}^{red}(p_1,\ldots,p_n)$. Es werden bestimmte Stellen der Syntax der äußeren, parametrischen Sprache offen gelassen und zu einem späteren Zeitpunkt durch Teile einer anderen, eigenständigen Sprache gefüllt. Die Variabilität ist also durch unterschiedliche Belegungen der Parameter $p_1,\ldots,p_n$ gegeben. Besonders häufig findet man diesen Effekt bei Modellierungssprachen, in denen Aktions- oder Constraint-Sprachen verwendet werden können. In der Grammatik 2.3 in Abschnitt 2.4.1 wurde zum Beispiel die Produktion für Invarianten offen gelassen. Mit demselben Mechanismus ist das Beispiel in Abb. 3.4 realisiert. Es zeigt ein einfaches Zustandsübergangsdiagramm (Statechart), bei dem die Invarianten in eckigen Klammern durch OCL, die Aufrufe und Aktionen an den Transitionen durch Java beschrieben werden.

Buffer

Statechart

```
statechart Buffer {

 state NonEmpty {
   [self.elements->size > 0]
 }

 state Empty {
   [self.elements->size == 0]
 }

 Empty -> NonEmpty : add(x) / {this.elements.add(x);}
 NonEmpty -> Empty : get( ) / {return this.elements.get(0);}
}
```

Abbildung 3.4: Beispielmodell mit Sprachparametern

Die dritte Form der syntaktischen Variabilität sind syntaktische *Spracheinschränkungen* oder *Erweiterungen*. Wie in [DHTT00] motiviert können Varianten von Spracheinschränkungen für einen bestimmten Zweck bei Bedarf „zugeschaltet“ werden und schließen weitere Modelle syntaktisch aus. Dies ist in ähnlicher Form auch bei der Definition von UML-Profilen vorgesehen (vgl. [OMG09a, Abschnitt 13]). Die folgende Kontextbedingung 3.5 gibt beispielsweise an, dass für alle Attribute einer Klasse die Attributtypen definiert sein müssen. Die abstrakte Syntax für einfache Klassendiagramme wurde bereits in Abschnitt 2.4.2 beschrieben und wird hier verwendet. Die Bedingung 3.5 ist im Allgemeinen nicht notwendig, wird womöglich aber benötigt, wenn gewisse Vollständigkeitseigenschaften vorausgesetzt werden sollen (z.B. damit ein Codegenerator korrekt arbeitet). Alle Modelle, die diese Kontextbedingung nicht erfüllen, gelten in der betrachteten Sprachvariante $\mathcal{AS}_v^{red}$ nicht als wohlgeformt. Es werden nur Modelle betrachtet, die über die notwendigen Kontextbedingungen hinaus auch die weiteren Spracheinschränkungen erfüllen, z.B. also,

$$\mathcal{AS}_v^{red} = \{m \in \mathcal{AS}^{red} \mid \mathrm{attrDef}(m)\}$$

Da beliebige zusätzliche Bedingungen als Spracheinschränkung formuliert werden können, können Spracheinschränkungen auch die Ausdrucksstärke einschränken. Die Ausdrucksstärke bleibt erhalten, falls

$$\forall\, m_1 \in \mathcal{AS}^{red} : \exists\, m_2 \in \mathcal{AS}_v^{red} : \mathrm{sem}(m_1) = \mathrm{sem}(m_2)$$

Es ist beispielsweise das Ziel von Modellierungs- oder Programmierrichtlinien [Mat07, MIS10], die Verwendung von unerwünschten (z.B. unsicheren) Modellen zu verhindern, ohne jedoch die Ausdrucksstärke einzuschränken. Eine Einschränkung der Ausdrucksstärke kann beispielsweise notwendig sind, um eine Sprache an eine Zielplattform anzupassen, in der bestimmte Konstrukte nicht realisierbar sind.

Variante KB 3.5 (Attributtypen sind definiert)

AttrDef

use CDSimpAS

$\mathrm{attrDef} : \mathit{CDDia} \rightarrow \mathrm{bool}$
$\mathrm{attrDef}\ (\mathit{dname}, \mathit{classes}, \mathit{assocs}, \mathit{invars}) =$
$\quad (\forall (\mathit{cname}, \mathit{attrs}) \in \mathit{classes}, \mathit{at} \in \mathit{attrs} : \mathit{at}.\mathit{Type} \neq \epsilon)$

Syntaktische Spracherweiterungen, also die Aufnahme von zusätzlichen Konstrukten in die konkrete und (reduzierte) abstrakte Syntax, lassen sich nicht als Präsentationserweiterung verstehen und können daher nicht mittels syntaktischer Transformationen entfernt werden. In dieser Arbeit betrachten wir vornehmlich Spracheinschränkungen und setzen diese als optionale Kontextbedingungen um.

Syntaktische Variabilität bleibt in der reduzierten abstrakten Syntax erhalten und kann damit mit der Semantik interagieren. Zusammenfassend bilden Varianten von Stereotypen, Sprachparameter und Spracheinschränkungen/-erweiterungen die syntaktische Variabilität einer Sprache.

3.1.3 Semantische Variabilität

In [Sel04] wird der Begriff eines „semantic envelope“ für die UML verwendet, um zu beschreiben, dass die Semantik in einem festgelegten Rahmen variiert werden kann. Während im UML-Standard aber nur allgemein die Rede von semantischen Variationspunkte ist, unterteilen wir gemäß Abb. 2.1 semantische Variabilität weiter in *semantische Domänenvariabilität* und *semantische Abbildungsvariabilität*. Eine hilfreiche Analogie ist, dass Variabilität in der semantischen Abbildung Konfigurationsoptionen eines Codegenerators entsprechen, während Variabilität in der semantischen Domäne Auswahlmöglichkeiten von Eigenschaften eines zugrunde liegenden Laufzeitsystems oder einer Zielplattform entsprechen.

Mit der Auswahl von Varianten für die semantische Domäne $\mathcal{S}$ entsteht eine angepasste semantische Domäne $\mathcal{S}_v$, deren Elemente zusätzliche Eigenschaften, zum Beispiel gefordert durch ein Prädikat prop_v, erfüllen:

$$\mathcal{S}_v = \{s \in \mathcal{S} \mid \mathrm{prop}_v(s)\}$$

Die Definition des Systemmodells in Kapitel 4 enthält bereits eine ganze Reihe explizit formulierter Varianten als Erweiterungen durch optionale Definitionen. Semantische Domänenvarianten werden wie herkömmliche Definitionen über den Elementen des Systemmodells angegeben. Sie können bestehende Definitionen durch zusätzliche Eigenschaften einschränken oder auf existierenden Theorien weiteres mathematisches Rahmenwerk zur Abbildung eines bestimmten Sachverhalts aufbauen. Die Relation sub aus der Definition 2.7 des vorherigen Kapitels legt die Subklassenbeziehungen im Systemmodell fest. In einer Variante des Systemmodells soll nun beispielsweise nur einfache Vererbung möglich sein. Die Auswahl der Variante 3.6 stellt diese Eigenschaft der Systeme her. Systeme werden bezüglich ihrer erlaubten Subklassenrelation also eingeschränkt.

Variante SysMod 3.6 (Einfache Vererbung)

SingleInheritance

use SMsimp

$$\forall C_1, C_2, C_3 : (C_1, C_2) \in \mathrm{sub} \wedge (C_1, C_3) \in \mathrm{sub} \wedge C_2 \neq C_3$$
$$\Rightarrow (C_2, C_3) \in \mathrm{sub} \vee (C_3, C_2) \in \mathrm{sub}$$

Semantische Abbildungsvarianten sind alternative Definitionen (von Teilen) semantischer Abbildungen

$$\mathrm{sem}_{v1}, \mathrm{sem}_{v2} : \mathcal{AS}^{red} \to \wp(\mathcal{S})$$

Es können im Allgemeinen Definitionen beliebigen Inhalts als Varianten einer Abbildung deklariert werden. Die alternative Realisierung einer Abbildung wird durch ein kleines Beispiel verdeutlicht. Die semantische Abbildung von Superklassen wurde im Beispiel der semantischen Abbildung von einfachen Klassendiagrammen in Definition 2.8 (Abschnitt 2.4.5) bewusst offengelassen. Hier werden jetzt zwei Varianten skizziert. Variante 3.7 realisiert die Abbildung einer Menge von Superklassen *classes* zu einer Klasse *C* dadurch, dass alle Klassen in direkter Subklassenbeziehung in sub stehen.

Alternativ dazu wird in der Variante 3.8 ein Delegationsmechanismus verwendet (ohne ihn jedoch genau zu spezifizieren). Dies kann zum Beispiel notwendig werden, wenn Systeme im Systemmodell nach Variante 3.6 keine Mehrfachvererbung unterstützen. Hier zeigt sich auch, dass Abhängigkeiten zwischen der Wahl konkreter Varianten auf den verschiedenen Ebenen (Syntax, semantische Domäne und Abbildung) bestehen. Diese Abhängigkeiten werden im Verlauf des Kapitels noch genauer erläutert.

Semantische Variabilität ist zunächst transparent für den Modellierer. Das bedeutet, dass bei der Erstellung eines Modells nicht direkt auf die Bedeutung einzelner Modellelemente Einfluss

Variante SemAbb 3.7 (Abbildung Superklassen direkt)

MapDirect

use mCDDiagram

$mapSuperClasses_{SystemModel_{simp}} : \mathrm{UCLASS}_{SystemModel_{simp}} \rightarrow \wp(Name) \rightarrow \mathrm{bool}$

$$\begin{aligned}&mapSuperClasses_{sm}\ C\ classes =\\ &\quad \forall\, cl \in classes :\\ &\qquad \exists\, C' \in \mathrm{UCLASS}_{sm} : \mathrm{nameOf}_{sm}\ C' = mapIdent_{sm}\ cl.CName \wedge (C, C') \in \mathrm{sub}_{sm}\end{aligned}$$

Für jede Superklasse von C existiert eine Klasse C' im System, die mit C in Subklassenbeziehung steht.

Variante SemAbb 3.8 (Abbildung Superklassen über Delegation)

MapDelegate

use mCDDiagram

$mapSuperClasses_{SystemModel_{simp}} : \mathrm{UCLASS}_{SystemModel_{simp}} \rightarrow \wp(Name) \rightarrow \mathrm{bool}$

$mapSuperClasses_{sm}\ C\ classes = delegate_{sm}\ C\ classes$

Wie Delegation genutzt wird, um multiple Vererbung aufzulösen, ist für das Beispiel nicht von Belang. Wir nehmen der Einfachheit halber an, es gibt die Funktion *delegate*, die das intendierte Verhalten korrekt umsetzt.

genommen werden kann. Es kann jedoch sinnvoll sein, die Entscheidung über die Interpretation bestimmter Konstrukte dem Modellierer zugänglich zu machen. Dies lässt sich einfach erreichen, indem entsprechende syntaktische Varianten, nämlich Stereotypen, definiert werden, auf die dann in der semantischen Abbildung geeignet reagiert werden kann. Das Beispiel in Definition 3.9 nimmt an, dass Stereotypen für Klassen syntaktisch möglich sind und ein Stereotyp „singleton“ für Klassen definiert worden ist, der anzeigen soll, dass diese Klasse nur einfach instantiiert werden darf. Bei der semantischen Abbildung wird der Stereotyp berücksichtigt, indem zusätzlich *existsOnce* gefordert wird (ohne wiederum an dieser Stelle zu klären, wie *existsOnce* genau definiert ist).

3.2 Erfassen und Konfigurieren von Variabilität

Bei der Definition einer Modellierungssprache können Präsentationsvarianten, syntaktische und semantische Varianten definiert werden.

Für eine systematische Verwaltung der Varianten einer Modellierungssprache müssen die unterschiedlichen Varianten festgehalten und ihre Abhängigkeiten dokumentiert werden. Dabei können Abhängigkeiten zwischen Varianten aller Arten auftreten. Zusätzlich sind auch Abhän-

Variante SemAbb 3.9 (Semantische Abbildung für Klassen mit Singleton)

MapSingleton

use mCDDiagram

$$mapClass_{SystemModel_{simp}} : CDCLASS \rightarrow \mathrm{bool}$$

$$\begin{aligned}&mapClass_{sm}\ (stereos, cname, attrs) =\\&\quad \exists\, C \in UCLASS_{sm} : \ldots \wedge\\&\qquad singleton \in stereos \Rightarrow existsOnce_{sm}\ C\end{aligned}$$

gigkeiten zwischen Sprachen möglich. Die folgende Auflistung charakterisiert mögliche Abhängigkeiten zwischen Varianten.

Varianten sind optional Im Sinne der Unterspezifikation ist eine Festlegung auf eine bestimmte Variante nicht vorgeschrieben. Die Auswahl einer Variante kann immer offen gelassen werden und ist somit optional. In der Sichtweise einer losen Spezifikation kann aus einer nicht ausgewählten Variante nicht geschlossen werden, dass die Variante nicht gilt.

Varianten können sich ausschließen Die Auswahl einer bestimmten Variante kann dazu führen, dass eine andere Variante nicht mehr gewählt werden darf. Zum einen kann die ausgeschlossene Variante sich auf denselben Variationspunkt beziehen, so dass eine doppelte Definition nicht sinnvoll ist (Alternativen). So ist es beispielsweise nicht sinnvoll, zwei Varianten für die Definition der Abbildung von Superklassen (vgl. Varianten 3.7 und 3.8, die beide dieselbe Funktion definieren) zu wählen. Zum anderen sind Ausschlüsse zwischen Varianten ganz unterschiedlicher Variationspunkte möglich. Es ist zum Beispiel nicht sinnvoll, die Variante 3.6 (Mehrfachvererbung verboten) im Systemmodell zu wählen, falls die Variante 3.7 (Mehrfachvererbung direkt abbilden) bei der semantischen Abbildung ausgewählt wird.

Varianten können sich bedingen Die Auswahl einer bestimmten Variante kann dazu führen, dass für einen anderen Variationspunkt ebenfalls eine Variante ausgewählt werden muss. Eine Variante kann beispielsweise zusätzliche Bedingungen oder Strukturen einführen, deren Existenz von einer anderen Variante vorausgesetzt wird. Weitere Abhängigkeiten entstehen bei der Festlegung von Sprachparametern. Die Semantik einer Sprache, die z.B. eine Aktionssprache einbettet, hängt von der Semantik der Aktionssprache ab. Dabei ist auch zu prüfen, ob die Semantik der äußeren Sprache mit der Semantik der inneren Sprache verträglich ist und ob die äußere Sprache genügend Kontextinformationen zur Berechnung der Semantik der inneren Sprache zur Verfügung stellen kann. So verwenden OCL-Invarianten beispielsweise Klassennamen aus einem zunächst unbekannten Klassendiagramm.

3.2.1 Feature-Diagramme zur Dokumentation der Varianten

Die Definition von Varianten erfolgt mit Hilfe herkömmlicher Definitionen (siehe z.B. Variante 3.6). Zur Dokumentation möglicher Kombinationen und Abhängigkeiten von Varianten werden Feature-Diagramme [CE00] aus dem Bereich der Software Produktlinien eingesetzt. Für unsere Zwecke sind Feature-Diagramme ausreichend, die optionale Features, alternative Features sowie „requires“ und „excludes“ Beziehungen abbilden können.

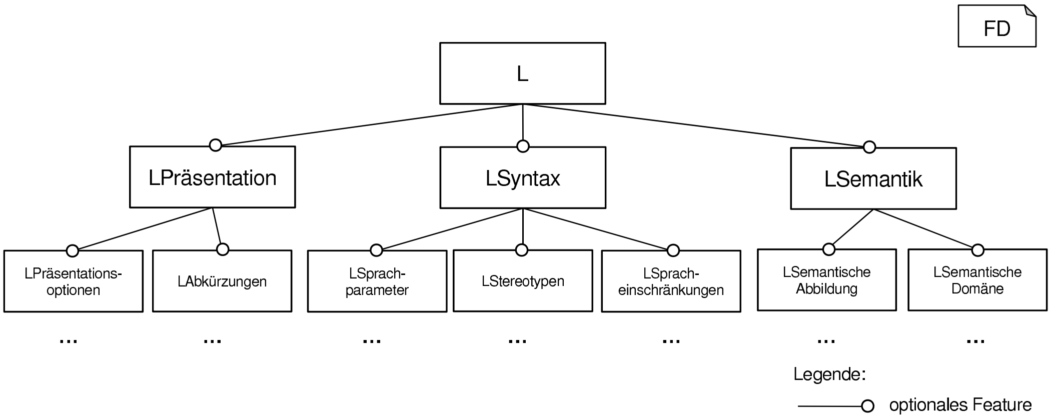


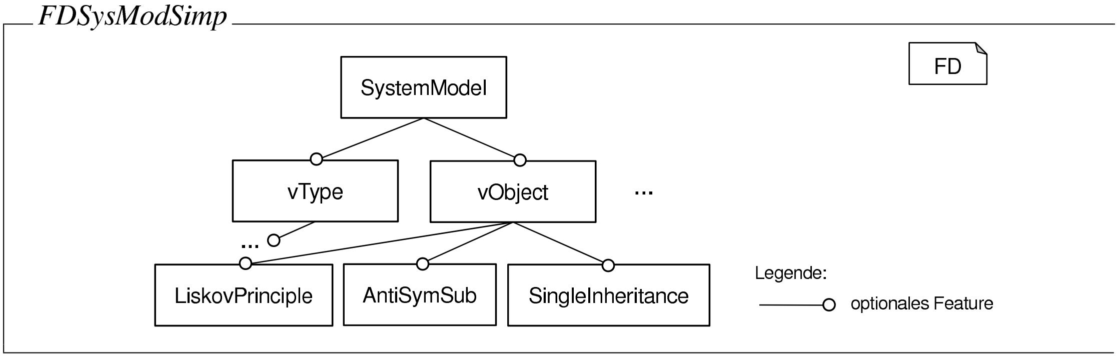
Abbildung 3.10: Vorlage zur Dokumentation der Varianten einer Sprache L

Feature-Diagramme sind Baumstrukturen. Der Wurzelknoten bezeichnet das Artefakt, für das die Variabilität beschrieben werden soll. Zwischenknoten im Baum werden als Variationspunkte bezeichnet und gruppieren Varianten oder weitere Variationspunkte hinsichtlich eines bestimmten Aspekts. Blätter im Feature-Diagramm sind Varianten. In einem Feature-Diagramm modellieren wir die Variabilität der Bestandteile einer Sprachdefinition. Unveränderliche Bestandteile sowie der hierarchischen Aufbau der Theorien werden, wie in [PBL05] vorgeschlagen, nicht im Feature-Diagramm, sondern in einer ergänzenden Beschreibung (z.B. Schichtenbild) erfasst. Abbildung 3.10 enthält ein Feature-Diagramm, das einen generischen Aufbau als Vorschlag zur Dokumentation der Varianten einer Sprache *L* zeigt. Der Aufbau entspricht dabei der Klassifikation aus Tabelle 3.1. Konkrete Varianten sind nicht angegeben, sie werden unterhalb der Blätter des Feature-Diagramms (ggf. weiter zergliedert) eingeführt. Die Verwendung von Feature-Diagrammen wird an zwei Beispielen erläutert.

Definition 3.11 zeigt einen Teil des Feature-Diagramms für das Systemmodell, wie es unterhalb des Knotens *LSemantik* als Feature-Diagramm für die semantische Domäne eingefügt würde (vgl. Abb. 3.10). Die Blätter des Feature-Baums entsprechen mathematischen Definitionen, die unter dem entsprechenden Namen geeignet abgelegt werden (*SingleInheritance* ist in Variante 3.6 angegeben). Für die Theorie *Object*, in der Klassen und Objekte definiert werden, gibt es den Variationspunkt *vObject*, der die Varianten dieser Theorie zusammenfasst. Das Diagramm ist insgesamt unvollständig und dient nur zu Illustrationszwecken. Dargestellt sind einige Varianten, die sich mit der Subklassenbeziehung (Vererbung) im Systemmodell beschäftigen. Die Variante *LiskovPrinciple* stellt sicher, dass eine Subklasse mindestens die Attribute

Definition Variabilität 3.11 (Vererbungsvarianten im Systemmodell)

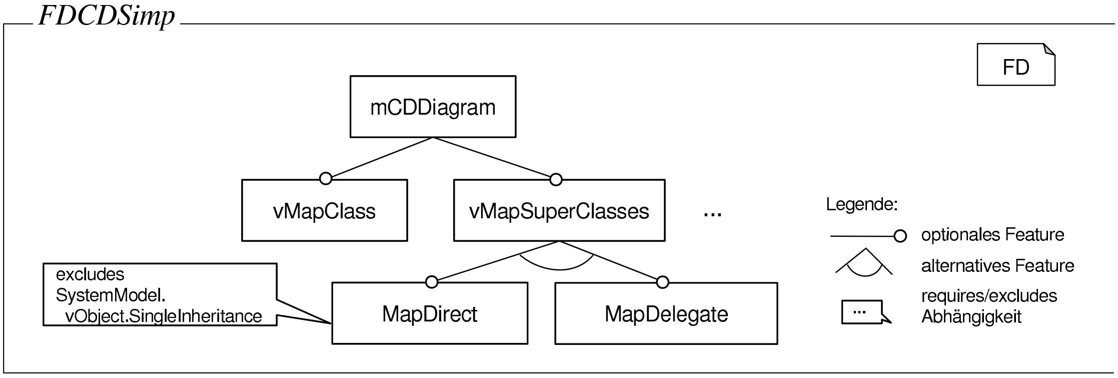
FDSysModSimp


FD
SystemModel
vType
vObject
...
...
LiskovPrinciple
AntiSymSub
SingleInheritance
Legende:
optionales Feature

ihrer Oberklassen enthält[1]. *AntiSymSub* sorgt für die Zyklenfreiheit in der Vererbungsbeziehung zwischen Klassen und *SingleInheritance* schließt für diese Beziehung Mehrfachvererbung aus.

Definition Variabilität 3.12 (Semantische Abbildung von CDSimp)

FDCDSimp


FD
mCDDiagram
vMapClass
vMapSuperClasses
...
excludes
SystemModel.
vObject.SingleInheritance
MapDirect
MapDelegate
Legende:
optionales Feature
alternatives Feature
requires/excludes
Abhängigkeit

In Definition 3.12 ist ein Feature-Diagramm für die semantische Abbildung einfacher Klassendiagramme skizziert. Dieses Feature-Diagramm ordnet sich in eine vollständige Variantendokumentation gemäß Abb. 3.10 unterhalb von *LSemantik* zur Variantendokumentation der semantischen Abbildung ein. Zur Abbildung der Superklassen lassen sich zwei Alternativen wählen, die den Varianten 3.7 und 3.8 entsprechen. Die Verbindung zu den Definitionen wird über Namensgleichheit hergestellt. Ebenfalls angegeben ist eine Ausschlussbedingung für *MapDirect*, die, wie bereits motiviert, die gleichzeitige Auswahl der Einfachvererbung im Systemmodell verhindert.

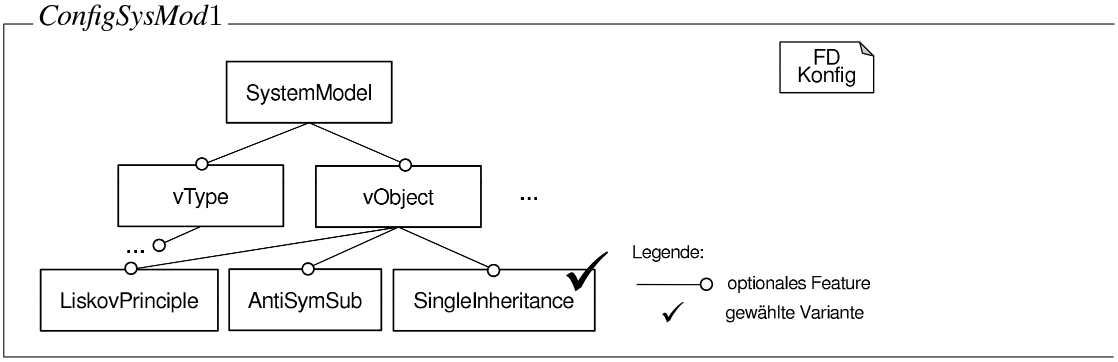
---

[1] Wir beschränken uns hier also nur auf einen Teil des ursprünglichen Substitutionsprinzips aus [LW94], bei dem eine weitergehende starke Verhaltenskonformität des Subtyps gefordert wird.

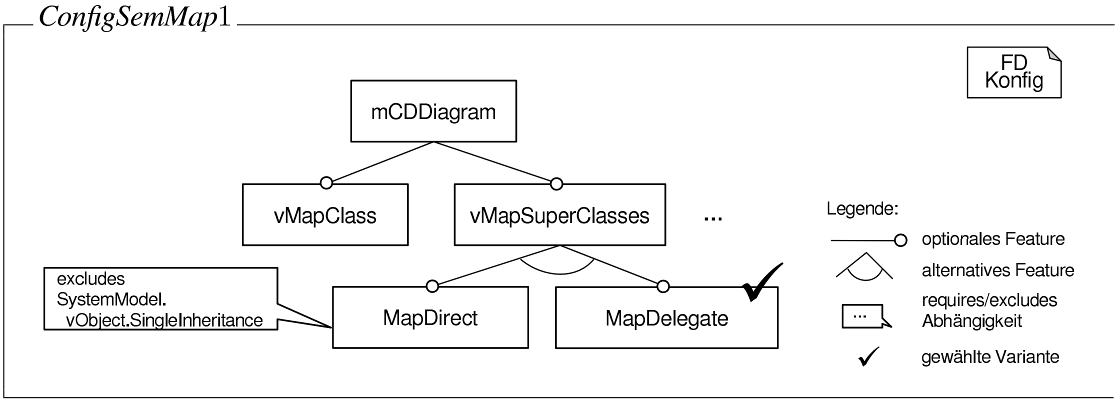
3.2.2 Konfiguration von Variabilität

Mit Hilfe von Feature-Diagrammen werden die möglichen Kombinationen von Varianten erfasst und dokumentiert. Um Modellierungssprachen mit Variabilität tatsächlich einzusetzen, können einige Varianten ausgewählt und damit die Sprachen konfiguriert werden. Hierfür werden Varianten (Blätter) des Feature-Baums als ausgewählt markiert. Die Konformität der Konfiguration zum Feature-Diagramm kann automatisch geprüft werden.

Konfiguration 3.13 (Konfiguration für das Systemmodell)


ConfigSysMod1
FD Konfig
SystemModel
vType
vObject
...
...
LiskovPrinciple
AntiSymSub
SingleInheritance
✓
Legende:
optionales Feature
✓ gewählte Variante

Konfiguration 3.14 (Konfiguration für die semantische Abbildung von CDSimp)


ConfigSemMap1
FD Konfig
mCDDiagram
vMapClass
vMapSuperClasses
...
excludes SystemModel. vObject.SingleInheritance
MapDirect
MapDelegate
✓
Legende:
optionales Feature
alternatives Feature
requires/excludes Abhängigkeit
✓ gewählte Variante

In den Konfigurationen 3.13 und 3.14 wird angedeutet, dass für das Beispiel aus dem vorherigen Abschnitt die Varianten mit der Auswahlmarkierung ($\surd$) als konkrete Konfiguration der Klassendiagramme und des Systemmodells ausgewählt werden. Eine konkrete Konfiguration entspricht also der Menge aller aktuell geltenden Definitionen. Häufig existieren mehrere Konfigurationen für ein Feature-Diagramm (z.B. existiert typischerweise je eine Systemmodellkonfiguration pro Modellierungssprache).

3.2.3 Überprüfen der Konfiguration

Anhand der Feature-Diagramme muss nun geprüft werden, ob die gewählte Konfiguration korrekt ist. Das bedeutet im Einzelnen:

1. Die Konfigurationen desselben Feature-Diagramms werden überlagert. Zwei Konfigurationen werden zu einer resultierenden Konfiguration überlagert, so dass jede Auswahlmarkierung im Ergebnis aus der ersten oder zweiten Konfiguration stammt.
2. Die Konformität der Konfiguration zu den Feature-Diagrammen wird überprüft:
   a) Es ist maximal eine Variante einer Alternative gewählt.
   b) Für jede gewählte Variante sind die exclude-abhängigen Varianten nicht ausgewählt.
   c) Für jede gewählte Variante sind die requires-abhängigen Varianten gewählt.

3.2.4 Konsistenz der Konfiguration

Leider ist nicht automatisch gesichert, dass eine gültige Konfiguration, die konform zu den Feature-Diagrammen ist, auch zu einer konsistenten Menge von Definitionen führt. Das liegt daran, dass das Feature-Modell unvollständig oder fehlerhaft sein kann und nicht alle inkonsistenten oder auch nur ungewollten oder uninteressanten Konfigurationen ausschließt. Insbesondere bei der Integration eigenständiger Modellierungssprachen, die aber eine gewisse Überlappung bezüglich ihrer Semantik besitzen, besteht die Gefahr, dass sich semantische Abbildungen generell widersprechen.

Prinzipiell besteht aber die Möglichkeit, die Konsistenz formal nachzuweisen. Für zwei konfigurierte Sprachen $\mathcal{L}_1$ und $\mathcal{L}_2$ mit semantischen Abbildungen sem_1 und sem_2 beispielsweise ist nachzuweisen, dass

$$\mathrm{sem}_1(m_1) \cap \mathrm{sem}_2(m_2) \neq \varnothing$$

für zwei Zeugen $m_1 \in \mathcal{L}_1$ und $m_2 \in \mathcal{L}_2$ gilt. Zu beachten ist, dass hier die Wahl von m_1 und m_2 wichtig ist. Sie sollten so gewählt werden, dass genau die semantisch überlappenden Konstrukte enthalten sind, da dort am ehesten eine Inkonsistenz zu erkennen ist. Zudem können auch Inkonsistenzen in der konfigurierten semantischen Domäne auftreten, wenn diese hinreichend komplex ist. Hier ist nachzuweisen, dass die Menge der Elemente in der semantischen Domäne nicht von vornherein leer ist. Die Konsistenz der Konfiguration ist unabhängig von der Konsistenz konkreter Modelle. Es kann gewollt sein, dass Modelle redundante und unter Umständen sich widersprechende Informationen beinhalten.

3.3 Vorgehen bei Sprachänderungen

Vor der Verwendung einer Modellierungssprache mit Variabilität kann die Sprache durch Konfiguration auf ihren Einsatzzweck abgestimmt werden. Während der Verwendung im Projekt kann es zudem notwendig werden, die Konfiguration zu verändern, um die Syntax oder Semantik der Sprache an die Gegebenheiten anzupassen. Hierbei müssen im Allgemeinen bereits erstellte Modelle auf ihre syntaktische und semantische Richtigkeit überprüft werden. In welchem Fall zwei

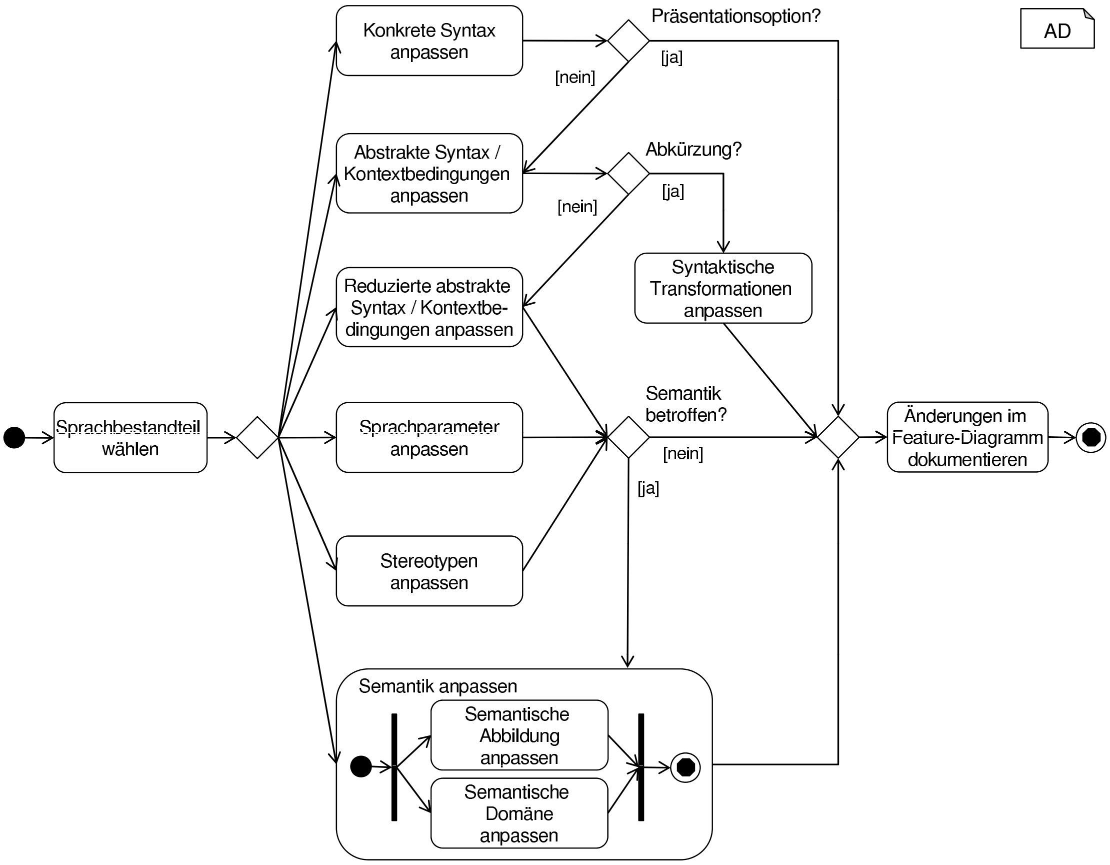

AD
Sprachbestandteil wählen
Konkrete Syntax anpassen
Präsentationsoption?
[ja]
[nein]
Abstrakte Syntax / Kontextbedingungen anpassen
Abkürzung?
[ja]
[nein]
Syntaktische Transformationen anpassen
Reduzierte abstrakte Syntax / Kontextbedingungen anpassen
Sprachparameter anpassen
Semantik betroffen?
[nein]
[ja]
Stereotypen anpassen
Änderungen im Feature-Diagramm dokumentieren
Semantik anpassen
Semantische Abbildung anpassen
Semantische Domäne anpassen

Abbildung 3.15: Vorgehen bei Änderungen an der Sprachdefinition

Konfigurationen semantisch verträglich sind und die Überprüfung entfallen kann, wird kurz in Abschnitt 8.5 an einem Beispiel gezeigt.

Die Notwendigkeit einer Änderung wird häufig erst während der Verwendung deutlich, da erst dann umständliche Konstrukte oder unintuitive Bedeutungen zu Tage treten. Anhand der vorgestellten Klassifikation und der Bestandteile einer Sprachdefinition lässt sich gut nachvollziehen, welche Auswirkungen Änderungen der Sprache haben. Eine bildliche Übersicht des Vorgehens bei Sprachänderungen ist in Abb. 3.15 gezeigt. Generell kann eine Sprachanpassung das Einführen, Löschen oder Ändern beliebiger Sprachbestandteile bedeuten. Sprachentwickler haben dabei die Wahl, bei Hinzunahme oder Änderung den entsprechenden Sprachbestandteil als Variante einzuführen und im Feature-Diagramm zu dokumentieren. Bereits existierende Modelle müssen bei Änderungen der Sprache oder der Sprachkonfiguration überprüft und gegebenenfalls abgeändert werden.

- Handelt es sich bei Änderung der konkreten Syntax um eine Präsentationsoption, hat dies keine Auswirkungen auf die abstrakte Syntax und es reicht eine Aktualisierung des entsprechenden Feature-Diagramms.

- Andernfalls muss die abstrakte Syntax angepasst werden. Dies kann ein manueller Schritt sein oder auch automatisiert geschehen, so dass konkrete und abstrakte Syntax automatisch konsistent gehalten werden. Ist eine separate Änderung der abstrakten Syntax möglich, kann dies zur Folge haben, dass die konkrete Syntax angepasst werden muss. Diese Abhängigkeit ist nicht dargestellt, da wir immer davon ausgehen, dass die abstrakte Syntax aus der konkreten Syntax abgeleitet wird.

- Ist die Änderung an der abstrakten Syntax als Abkürzung aufzufassen, müssen gegebenenfalls vorhandene syntaktische Transformationen angepasst werden, die die Abkürzungen in Elemente der reduzierten abstrakten Syntax übersetzen.

- Andernfalls kann es notwendig sein, die reduzierte abstrakte Syntax an die Änderungen der abstrakten Syntax anzupassen. Es ist darauf zu achten, dass Änderungen an der reduzierten abstrakten Syntax nicht dazu führen, dass Abkürzungen nicht mehr überführt werden können.

- Anpassungen der reduzierten abstrakten Syntax, Sprachparameter oder Stereotypen können dazu führen, dass die Semantik der betroffenen Konstrukte angepasst werden muss, wobei sowohl semantische Abbildung als auch semantische Domäne betroffen sein können.

Die Anpassungen können an jeder Stelle des Aktivitätsdiagramms angesetzt werden. So kann beispielsweise nur eine Änderung der Semantik und nachfolgend die Aktualisierung des Feature-Diagramms erfolgen. Alle anderen Sprachbestandteile bleiben unberührt.

3.4 Verwandte Arbeiten

Unseres Wissens nach gibt es bisher keine allgemeine Klassifikation von Variabilität in Modellierungssprachen. Die Modellierung von Variabilität mit Hilfe von Feature-Diagrammen ist den Software Produktlinienansätzen entlehnt und stammt aus [CE00, PBL05]. Völter [Völ08] beschreibt einen Ansatz zur Definition einer Familie von Architekturbeschreibungssprachen, der ebenfalls mit Feature-Diagrammen arbeitet. Syntaktische Varianten und dazugehörige Codegenerierung können konfiguriert werden. Formale Semantik ist nicht Gegenstand seiner Arbeit. Die Idee, mit Hilfe syntaktischer Konstrukte (Stereotypen) eine Variante der Ausgangssprache zu bilden, ist vergleichbar mit der Profilbildung in der UML und wird auch in [Rum04b, FPR02] verwendet. Eine enge, formale Abstimmung mit der Semantik der Sprache wie in der vorliegenden Arbeit findet dort aber nicht statt. Die Ergebnisse dieses Kapitels können generell genutzt werden, um eine semantisch fundierte Fassung von UML-Profilen und den im UML-Standard heute noch wenig präzise definierten semantischen Variationspunkten zu entwickeln.

Im Bereich der Semantik von Modellierungssprachen unterstützen einige Ansätze semantische Variabilität in gewissem Umfang. Template semantics [NAD03] kann genutzt werden, um die Verhaltenssemantik von zustandsbasierten Modellierungssprachen parametrisch zu beschreiben. Die Ausführungssemantik basiert auf parametrischen hierarchischen Transitionssystemen, deren Verhalten durch vordefinierte template-Parameter konfiguriert werden kann. In [TA06] wird template semantics genutzt, um die Semantik von UML-StateMachines mit ihren semantischen Variationspunkten zu definieren. In [PADS08, ADN$^+$06] werden semantisch konfigurierbare Java-Codegeneratoren und analysierbare Modelle mit Hilfe von template semantics beschrieben. Template semantics bietet eine umfangreiche Theorie für zustandsbasierte Modellierungssprachen, ist aber auf zustandsbasierte Notationen und Mikro-Makro-Step-Semantik in ihrer Beschreibungsmächtigkeit beschränkt.

Templatable metamodels, eingeführt in [CMTG07b, CMTG07a], ist ein vergleichbarer Ansatz, um die abstrakte Syntax und operationale Semantik einer domänenspezifischen Sprache mit Varianten zu definieren. Der Ansatz verwendet den UML 2 Profil- und Templatemechanismus, um Variationspunkte auf Metamodellebene zu definieren und die eingeführten generischen Typen auf Metamodell- oder Modellebene zu binden. Wie template semantics ist der Ansatz auf die Beschreibung von Verhalten mit Hilfe operationaler Semantik ausgelegt, legt aber mehr Wert auf OMG-Standard-konforme Mechanismen.

Ein etwas andersartiger Ansatz ist in [CJ05] beschrieben, in dem semantische Varianten als eigenständige Klassendiagramme modelliert und zusammen mit einem Quellmodell in ein Zielmodell transformiert werden, welches die gewählten Varianten widerspiegelt. Im Fokus steht wiederum die Beschreibung von Verhalten. Varianten entsprechen dabei Operationen im Klassenmodell, die in einer geeigneten Aktionssprache implementiert werden.

3.5 Zusammenfassung

Der Inhalt dieses Kapitels ist zweigeteilt. Zunächst wurde eine Klassifikation von Variabilität in Modellierungssprachen angegeben. Variabilität kann dabei als Präsentationsvariabilität, syntaktische oder semantische Variabilität klassifiziert werden. Semantische Variabilität umfasst Variabilität in der semantischen Domäne und in der semantischen Abbildung. Damit unterscheiden wir deutlicher als der UML-Standard (nur allgemeine semantische Variationspunkte) zwischen Varianten im zugrunde liegenden „Laufzeitsystem“ (Domäne) und Optionen für die Abbildung in diese Domäne. Die Klassifikation wurde in Tabelle 3.1 zusammengefasst.

Der zweite Teil des Kapitels zeigt, wie Varianten und deren Abhängigkeiten untereinander mit Hilfe von Feature-Diagrammen erfasst und Konfigurationen erstellt und überprüft werden können. Eine Werkzeugunterstützung für die Definition von Variabilität in Modellierungssprachen wird in den Kapiteln 5 und 6 präsentiert. Zudem wurde illustriert, wie die Klassifikation dazu verwendet werden kann, die Auswirkungen bei Sprachänderungen abzuschätzen. Nicht betrachtet wurde die Auswirkung auf konkrete Modelle, die an die geänderte Sprache angepasst werden müssen.

Kapitel 4

Systemmodell

In diesem Kapitel wird die in dieser Arbeit verwendete semantische Domäne, das Systemmodell, definiert. Zunächst finden sich in Abschnitt 4.1 einige Erläuterungen zu den verwendeten Entwurfsprinzipien und zum Aufbau des Systemmodells. In den Abschnitten 4.2 bis 4.8 werden die Systemmodelldefinitionen und ihre Varianten angegeben. Abschnitt 4.9 fasst das Kapitel kurz zusammen.

4.1 Einführende Erläuterungen

Ein Systemmodell ähnlicher Art zur Semantikdefinition wurde erstmals im Rahmen des SYSLAB Projekts [KRB96, BHH$^+$97] eingesetzt. [Rum96] entwickelt diese Theorie weiter und benutzt ein Systemmodell zur Semantikdefinition bestimmter objektorientierter Beschreibungssprachen mit einem Fokus auf Zustandsmaschinen. Für das *UML 2 semantics project* [BCD$^+$06] wurde ein Systemmodell in drei Teilen angegeben [BCR06, BCR07a, BCR07b]. Es wurde in [BCGR08] überarbeitet und in zwei Buchkapiteln in [BCGR09a, BCGR09b] in wesentlichen Teilen veröffentlicht wurde. Das in diesem Kapitel definierte Systemmodell entspricht weitgehend den Definitionen aus [BCGR08]. Die Spezifikation des Verhaltens basiert stark auf der Theorie der Ströme und stromverarbeitender Funktionen [BS01, Rum96] und wird in [BCGR08] skizziert. Zur Beschreibung des Objekt- und Systemverhaltens mittels Zustandsmaschinen werden in diesem Kapitel allerdings nur die notwendigen Signaturen angegeben, die genaue Definition bleibt offen. Dies liegt darin begründet, dass die in Anhang B angegebene Umsetzung nach Isabelle/HOL für die Stromtheorien nicht Teil dieser Arbeit ist, aber für eine zukünftige Integration bereits begonnen wurde (siehe [GR07]). Trotzdem können die später definierten Semantiken auch ohne die vollständig ausgearbeiteten Theorien des (zustandsbasierten) Verhaltens definiert werden. Natürlich schränkt das Fehlen gewisser Details dieser Theorien die Nutzbarkeit des Systemmodells und damit der Semantiken für Verifikationszwecke leicht ein.

4.1.1 Entwurfsprinzipien und -entscheidungen

Für die Definition des Systemmodells verwenden wir eine Notation, die selbst eine wohldefinierte und bekannte Semantik besitzt. Wir bauen das Systemmodell auf einfachen mathematischen Grundlagen wie Mengen, Relationen und Funktionen modular auf. Das hat den Vorteil, dass diese Grundlagen als bekannt und allgemein akzeptiert vorausgesetzt werden können. Damit ist auch die Zugänglichkeit der Definitionen für einen breiten Personenkreis gegeben. Nachteil ist, dass nicht auf bereits für diesen Kontext ausgearbeitete Theorien zurückgegriffen werden kann und das Systemmodell von Grund auf definiert werden muss. Dies führt zu einer relativ umfangreichen Formalisierung.

Es wird kein existierender Formalismus zur Definition des Systemmodells verwendet, weil unseres Wissens nach kein Kandidat die erforderliche Allgemeinheit für eine zugängliche Beschreibung von Struktur und Verhalten besitzt, sondern meist einseitig auf die Behandlung bestimmter Aspekte ausgerichtet ist.

- Algebraische und modellorientierte Spezifikationssprachen wie CASL [BM04] oder auch Object-Z [Smi00] sind gut geeignet, die Struktur und das Verhalten einzelner Komponenten zu spezifizieren (siehe z.B. [EFLR99]). Die Schwierigkeit besteht allerdings in der Beschreibung der Interaktion und Kommunikation zwischen Komponenten, weswegen in [SD02] auch die Integration von Object-Z mit der Prozessalgebra CSP [Hoa83] untersucht wird. Unsere Definitionen sind ähnlich den Z-Spezifikationen aufgebaut, die auch einen Spezifikationsrahmen vorschreiben. Es wird allerdings nicht das vollständige syntaktische „Korsett“ übernommen, da dies später bei der Werkzeugunterstützung hinderlich wäre und keine größeren Vorteile, z.B. hinsichtlich Beweisprinzipien, mit sich bringt.

- Eine Verwandtschaft zu ASMs [BS03] besteht ebenfalls. ASMs bilden Transitionssysteme, deren Zustände ganz allgemeine mathematische Strukturen (Algebren) sind. Diese müssen erst ähnlich zu den Definitionen in diesem Kapitel definiert werden. Wir sehen ebenfalls von der Verwendung von ASMs ab. Die existierenden Theorien und Werkzeuge für ASMs sind nicht für Mengen von ASMs geeignet, die bei unserem mengenwertigen Semantikansatz entstehen.

- Alloy [Jac02] ist eine durch Z beeinflusste Spezifikationssprache, die auf einer Logik erster Stufe basiert. Wie in unserem Ansatz können Strukturen (Relationen) definiert und über Prädikate komplexe Bedingungen auch bezüglich des Verhaltens angegeben werden. Zudem steht mit dem Alloy Analyzer [All10] ein automatischer constraint solver zur Verfügung. Die Einschränkung auf eine Logik erster Stufe ist für uns ungeeignet, da sich die Verhaltensbeschreibung, basierend auf der Theorie der Ströme, hierin nur begrenzt darstellen lässt.

- Eine weitere Klasse von Spezifikationssprachen zielt vornehmlich auf die Beschreibung von (nebenläufiger) Interaktion zwischen Komponenten ab und ermöglicht nicht direkt die Modellierung von Intra-Objektverhalten objektorientierter Systeme. Hierzu zählen Prozessalgebren wie CSP [Hoa83] oder der π-Kalkül [Mil93], der zusätzlich in der Lage ist, sich dynamisch ändernde Kommunikationsbeziehungen zu beschreiben. Petrinetze [Pet62] sind ebenfalls in erster Linie zur Beschreibung von Nebenläufigkeit geeignet.

- Graphtransformationsansätze, z.B. [Sch91], sind sehr vielseitig einsetzbar und betonen durch ihre Transformationsregeln eher eine operationale Sicht auf ein System und haben Schwächen bei der Unterspezifikation [KGKZ09].

- Für die Definition des Systemmodells favorisieren wir eine Formalisierung, in der Struktur, zustandsbasiertes Verhalten und Interaktion gleichrangig beschrieben werden können. Daher werden wie erwähnt einfache mathematische Grundlagen verwendet und auf die

Theorie der Ströme [BS01, Rum96] gesetzt, die allerdings nicht im Fokus dieser Arbeit stehen. Die Werkzeugunterstützung auf Basis des Theorembeweisers Isabelle/HOL profitiert ebenfalls von der direkten Verwendung mathematischer Grundlagen, da so eine vergleichsweise einfache Kodierung der Definitionen in Isabelle möglich ist.

Da ein Ziel dieser Arbeit die Definition der Semantik der UML/P ist, wird nicht auf die Möglichkeit zurückgegriffen, UML/P selbst als semantische Domäne heranzuziehen. Eine solche zirkuläre Definition wäre problematisch und muss ohnehin mit Hilfe eines weiteren Formalismus durchbrochen werden. Nach einer Semantikdefinition von UML/P bietet sich allerdings die Möglichkeit, UML/P als semantische Domäne für weitere, UML/P-ähnliche Sprachen einzusetzen.

Die Definition des Systemmodells erfolgt nicht konstruktiv. Stattdessen werden Elemente eingeführt und deren Eigenschaften charakterisiert. Diese Charakterisierung ist meist unterspezifiziert. So werden zum Beispiel Universen von Typnamen oder Klassennamen (die Mengen UTYPE bzw. UCLASS) deklariert, die genaue Zahl oder die Struktur der Elemente aber offengelassen. Klassen werden nicht, wie häufig üblich, als *record* definiert. Ihre genaue Struktur bleibt ebenso unterspezifiziert. Es werden aber Selektorfunktionen definiert, die beispielsweise die Menge der Attribute oder Methoden zu einer Klasse liefern. Jede beliebige konkrete Menge kann angenommen werden, solange die grundlegenden Eigenschaften erfüllt werden.

Implizite Annahmen werden vermieden. Was nicht explizit definiert ist, muss also auch nicht gelten (im Sinne einer losen Semantik [BW82]). Durch die Unterspezifikation ist das Systemmodell nicht ausführbar. Dies würde eine zu große Einschränkung darstellen und Probleme bei der Semantik unvollständiger oder nicht als Ablauf darstellbarer Modelle hervorrufen. Eine Umsetzung in eine ausführbare Implementierung ist allerdings möglich und wurde bereits in [CDGR07] demonstriert.

Unterspezifikation erlaubt eine einfache Variantenbildung im Systemmodell, zum Beispiel durch Verstärkung von Bedingungen und damit Einschränkung der erlaubten Strukturen oder Abläufe. Generell kann das Systemmodell als Laufzeitsystem ähnlich der *Common Language Runtime (CLR)* von *.NET* oder Zielplattform betrachtet werden, wobei typische Varianten eines solchen Systems erfasst werden sollen. Dabei handelt es sich zum Beispiel um Variabilität bei der Definition vordefinierter Datentypen oder Typkonstruktoren, von typsicherem Überschreiben von Methoden bis hin zu systemweiten Schedulingstrategien. Im Zuge der Einführung der Systemmodelltheorien werden interessante Varianten als Variabilität der semantischen Domäne definiert und als Feature-Diagramm modelliert.

4.1.2 Aufbau des Systemmodells

Das Systemmodell beschreibt ein Universum aller möglichen objektbasierten Systeme. Semantische Abbildungen interpretieren Modelle als Prädikate, die dieses Universum auf eine Teilmenge einschränken, diese Teilmenge beschreibt genau die Bedeutung des Modells. Um ein Universum von Systemen zu definieren, die die Eigenschaften von Modellen komplexer Sprachen wie UML widerspiegeln können, muss das Systemmodell selbst eine gewisse Komplexität besitzen. Das Systemmodell wird jedoch auf modulare Weise definiert. In Schichten werden aufeinander aufbauende Theorien eingeführt, wie Abb. 4.1 erkennen lässt.

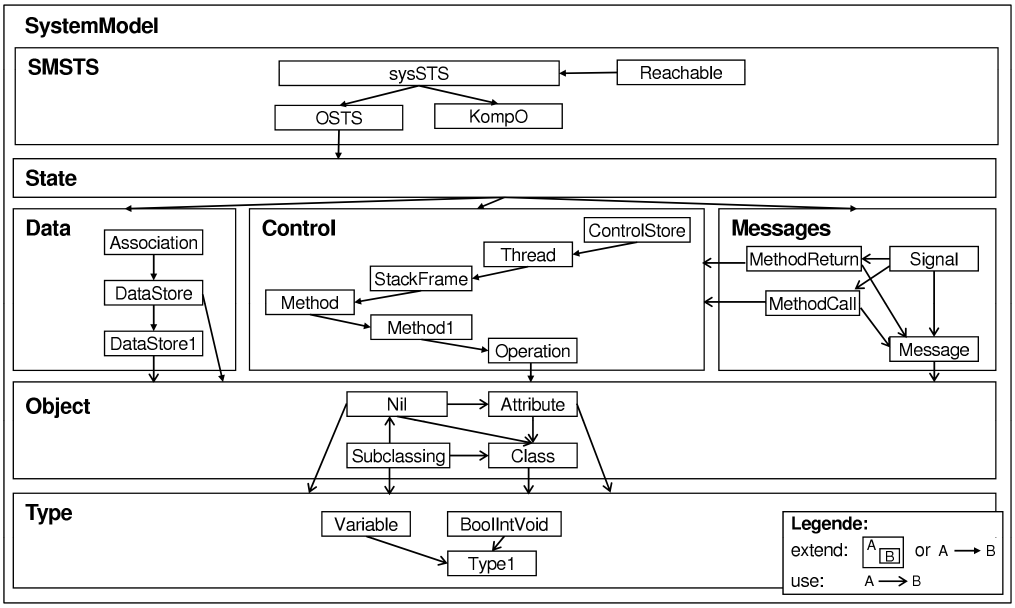

SystemModel
SMSTS
sysSTS
Reachable
OSTS
KompO
State
Data
Association
DataStore
DataStore1
Control
ControlStore
Thread
StackFrame
Method
Method1
Operation
Messages
MethodReturn
Signal
MethodCall
Message
Object
Nil
Attribute
Subclassing
Class
Type
Variable
BoolIntVoid
Type1
Legende:
extend: A B or A → B
use: A → B

Abbildung 4.1: Schichtenbild der Systemmodelltheorien

Rechtecke stellen Theorien dar, die andere Theorien enthalten können (das heißt diese Theorien erweitern). Beziehungen zwischen Theorien wie das Erweitern oder Referenzieren werden durch zwei unterschiedliche Pfeile in der Abbildung ausgedrückt. Jede Theorie kann die in Anhang A eingeführten mathematischen Grundlagen verwenden, ohne sie explizit zu referenzieren. Die Abbildung ist von unten nach oben zu lesen und zeigt als Basisschicht die Theorie *Type*. Diese Theorie definiert beispielsweise vordefinierte Typnamen, Werte und Trägermengen zu Typnamen und führt Variablen ein. Die Theorie *Object* baut hierauf auf und formalisiert bekannte objektorientierte Konzepte wie Klassen und Objekte und eine Subklassenbeziehung. Die Abbildung zeigt die möglichen Varianten nicht. Die Varianten gehören nicht zu den Kerndefinitionen und können optional verwendet werden. Sie werden in den einzelnen Unterabschnitten definiert und ihre Abhängigkeiten als Feature-Diagramm dargestellt.

Es existieren unterschiedliche Paradigmen, ein potentiell verteiltes, objektbasiertes System zu beschreiben. Im Systemmodell wird eine globale Zustandsmaschine (*state transition system, sysSTS*) und Zustandsmaschinen für einzelne Objekte, *OSTS* (siehe Abschnitt 4.8) verwendet. Diese Zustandsmaschinen sind nicht zu verwechseln mit den aus der UML bekannten Zustandsmaschinen (*StateMachines* oder *Statecharts*). Zustandsmaschinen besitzen Transitionen und Zustände (Theorie *State*, siehe Abschnitt 4.7). Zustände bestehen aus einem Nachrichtenpuffer (Theorie *Messages*, Abschnitt 4.6), dem Kontrollzustand, der den aktuellen Ausführungszustand von Methoden durch Threads speichert (Theorie *Control*, Abschnitt 4.5), und einem Datenzustand (Theorie *Data*, Abschnitt 4.4).

Zustandsinformationen im Systemablauf ändern sich *dynamisch*, folgen aber den zuvor *statisch* festgelegten Strukturen (Objekte gehören zu Klassen, deren Attributnamen und -typen statisch festgelegt sind usw.). Die einzelnen Theorien, dargestellt in Abb. 4.1, definieren jeweils sowohl den statischen als auch den dynamischen Teil eines Systems.

Die in den folgenden Unterabschnitten angegebenen Definitionen gelten für jedes System des Systemmodells. Das heißt, jedes System besitzt beispielsweise ein Universum der Typnamen UTYPE und das Systemverhalten ist durch eine eigene Zustandsmaschine bestimmt. Unterschiedliche Systeme im Systemmodell besitzen also nicht zwangsläufig dieselben Merkmale. Das Universum aller Systeme wird als $\mathrm{SystemModel}$ bezeichnet. Ein Element hieraus, also ein System $sm \in \mathrm{SystemModel}$ kann als großes Tupel aufgefasst werden, wobei einzelne Bestandteile selektierbar sind, z.B. ist $sm.\mathrm{UTYPE}$ das Universum der Typnamen für das gegebene System sm. Die folgenden Definitionen werden jedoch „aus Sicht“ eines einzelnen Systems formuliert. Die verwendete Notation wurde bereits in Abschnitt 2.3 eingeführt.

4.2 Typnamen, Werte und Variablen

In einer ersten, grundlegenden Theorie definieren wir Typnamen, Trägermengen und Variablen. Auf diesen Definitionen basieren dann die Definitionen von Klassen und Objekten.

Das Universum UTYPE enthält alle in einem System existierenden Typnamen. Jedem Typnamen ist über die Funktion CAR eine nicht-leere Menge von Werten, die Trägermenge, zugeordnet. Alle Werte werden im Universum der Werte UVAL zusammengefasst, siehe Definition 4.2. Typnamen werden normalerweise nicht weiter detailliert. Sie bleiben abstrakte Namen.

Definition SysMod 4.2 (Typnamen, Werte und Trägermengen)

*Type*1

UTYPE
UVAL
$\mathrm{CAR} : \mathrm{UTYPE} \to \wp(\mathrm{UVAL})$

$\forall u \in \mathrm{UTYPE} : \mathrm{CAR}(u) \neq \varnothing$

UTYPE ist das Universum der Typnamen. UVAL ist das Universum der Werte. Für jeden Typnamen liefert CAR eine nicht-leere Trägermenge von Werten zu dem Typnamen. Trägermengen müssen nicht disjunkt sein.

Insbesondere werden Typnamen nicht als Typen eines zugrunde liegenden Typsystems der mathematischen Struktur aufgefasst. Vielmehr werden Typen im Systemmodell direkt repräsentiert. Sprechen wir im Folgenden gelegentlich auch von Typen statt von Typnamen, meinen wir damit immer Elemente aus UTYPE, die bestimmte Typen direkt repräsentieren. Eine ausführliche Definition eines Typsystems für das Systemmodell erfolgt nicht. Dies ist auch nicht notwendig, da mit Hilfe des Systemmodells keine typtheoretischen Fragen wie Wohlgetyptheit beantwortet werden sollen, die auch syntaktisch geklärt werden können.

Wir fordern nicht, dass aus einem Wert ein zugehöriger Typ eindeutig abgeleitet werden kann. Mit Hilfe der Funktion typeOf aus Variante 4.3 kann jedoch der entsprechende Typ abgeleitet werden. Wie die Funktion typeOf den Typ ermittelt, ist nicht beschrieben. Der Typ könnte beispielsweise im Wert selbst kodiert sein. Bestimmte Werte, wie Nil aus Definition 4.11, haben keinen (eindeutigen) Typ. Die Funktion ist daher partiell.

Variante SysMod 4.3 (Werte mit eindeutig bestimmten Typen)

TypeOf

use Type1

$\text{typeOf} : \text{UVAL} \rightharpoonup \text{UTYPE}$

$\forall\, v \in \text{UVAL} : v \in \text{dom}(\text{typeOf}) \Rightarrow v \in \text{CAR}(\text{typeOf}(v))$

typeOf ordnet partiell Werten einen Typ zu.

Einige minimale Eigenschaften der Universen UTYPE und UVAL sind in Definition 4.4 formuliert. Wir gehen davon aus, dass mindestens Typnamen für ganze Zahlen und Wahrheitswerte in jedem System vorhanden sind. Typische logische oder arithmetische Operationen auf den Werten werden angenommen, aber nicht weiter ausgearbeitet. Ein spezieller Typname in UTYPE ist Void. Seine Trägermenge besteht nur aus einem einzigen Wert void. Die Konstruktion wird verwendet, um Methodenaufrufe ohne echten Rückgabewert abbilden zu können.

Definition SysMod 4.4 (Vordefinierte Typen)

BoolIntVoid

use Type1

$\text{Bool}, \text{Int} \in \text{UTYPE}$
$\text{Void} \in \text{UTYPE}$
$\text{true}, \text{false} \in \text{UVAL}$
$\text{void} \in \text{UVAL}$

$\text{CAR}(\text{Bool}) = \{\text{true}, \text{false}\}$
$\text{CAR}(\text{Int}) = \mathbb{Z} \subseteq \text{UVAL}$
$\text{CAR}(\text{Void}) = \{\text{void}\}$

UTYPE (UVAL) enthält mindestens Boolean und Integer (-werte). *void* kann zum Beispiel zur Kontrollrückgabe bei Methodenaufrufen genutzt werden, ohne eine tatsächlichen Wert zu senden.

Über weitere vordefinierte Typen werden keine Aussagen gemacht. Es erfolgt auch über (Subtyp-)beziehungen zwischen Typen keine Aussage, obwohl sie wie allgemein üblich leicht durch Teilmengenbeziehung zwischen ihren Trägermengen modelliert werden könnten. Vordefinierte Typen bilden auch einen Variationspunkt. In Variante 4.5 werden Typen und Werte für Zeichen und Zeichenketten eingeführt, wobei die konkrete Repräsentation von Zeichen in der Menge CharSet offengelassen wird.

Definition 4.6 wird benötigt, um die Semantik von Attributen (von Objekten) und Parame-

Variante SysMod 4.5 (Zeichen und Zeichenketten)

CharAndString

use Type1

$\mathrm{Char}, \mathrm{String} \in \mathrm{UTYPE}$
$\mathrm{CharSet} \subseteq \mathrm{UVAL}$

$\mathrm{CAR}(\mathrm{Char}) = \mathrm{CharSet}$
$\mathrm{CAR}(\mathrm{String}) = \mathrm{CharSet}^* \subseteq \mathrm{UVAL}$

Variante mit zusätzlichen vordefinierten Typen für Zeichen (Char) und Zeichenketten (String).

---

tern oder lokalen Variablen von Methoden beschreiben zu können. Wir nehmen zunächst eine

Definition SysMod 4.6 (Variablen, Attribute, Parameter)

Variable

use Type1

UVAR
Name
$\mathrm{vtype} : \mathrm{UVAR} \to \mathrm{UTYPE}$
$\mathrm{vname} : \mathrm{UVAR} \to \mathrm{Name}$
$\mathrm{VarAssign} \subseteq (\mathrm{UVAR} \rightharpoonup \mathrm{UVAL})$

Notation:
$a : T$ bezeichnet eine getypte Variable, Name und Typ sind explizit angegeben. Es gilt $\mathrm{vtype}(a : T) = T$ und $\mathrm{vname}(a : T) = a$.

$\forall\, v1, v2 \in \mathrm{UVAR} : \mathrm{vname}(v1) = \mathrm{vname}(v2) \Rightarrow v1 = v2$
$\forall\, VA \in \mathrm{VarAssign}, var \in \mathrm{dom}(VA) :$
$\quad VA(var) \in \mathrm{CAR}(\mathrm{vtype}(var))$

UVAR ist das Universum der Variablennamen. Name ist eine Menge von Namen. Aus Einfachheitsgründen nehmen wir an, dass jede Variable eindeutig durch ihren Namen bestimmt ist. VarAssign ist die Menge der (partiellen) Variablenbelegungen für Variablen aus UVAR.

Menge Name von Identifikatoren und Namen im Systemmodell an. Diese Menge wird nicht weiter beschrieben, dient jedoch dazu, allen typischerweise benannten Konstrukten in objektbasierten Systemen, wo sinnvoll, einen Namen zuordnen zu können. Hierzu zählen beispielsweise Variablennamen und später auch Methoden- oder Klassennamen.

Das Universum der Variablennamen ist UVAR. Jede Variable hat einen Namen und einen Typ. Damit können zum Beispiel zwei Variablen unterschieden werden, die denselben Namen aber unterschiedliche Typen, oder umgekehrt, unterschiedliche Namen, aber denselben Typ besitzen.

In Programmiersprachen kommt es häufig vor, dass zwei Variablen mit demselben Namen und Typ in unterschiedlichen Geltungsbereichen definiert und daher unterschieden werden. Dies ist mit der flachen Menge UVAR nicht direkt unterstützt, kann aber zum Beispiel durch Kodierung eines Namensraums im Variablennamen simuliert werden. Eine ähnliche Strategie kann auch genutzt werden, um Variablennamen global eindeutig zu machen. Wir nehmen daher vereinfachend an, dass alle Variablen in UVAR unterschiedliche Namen haben.

Beliebige Variablenbelegungen sind mit VarAssign modellierbar. Eine Variablenbelegung kann als partielle Abbildung aufgefasst werden. Der Definitionsbereich ist eine Menge von interessierenden Variablen, denen ein Wert zugewiesen wird. Es ist sichergestellt, dass jeder betrachteten Variablen ein Wert aus der Wertemenge ihres Typs zugewiesen wird.

Definition SysMod 4.7 (Type)

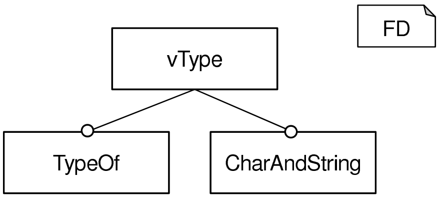
Type

extend Type1, BoolIntVoid, Variable

Definition 4.7 fasst nur die bisherigen Definitionen zur Theorie *Type* zusammen, die gemäß Abb. 4.1 den Grundstein für alle weiteren Theorien bildet. Die Variabilität wird mit Hilfe des Feature-Diagramms in Definition 4.8 zusammengefasst. *vType* ist der Variationspunkt der Theorie *Type* und bietet die optionalen Varianten *TypeOf* und *CharAndString*. Weitere in [BCGR08] betrachtete Varianten (Definitionen von records und Tupeln) werden hier nicht berücksichtigt.

Definition Variabilität 4.8 (vType)

vType



4.3 Klassen und Objekte

In diesem Abschnitt werden Klassen und Objekte im Systemmodell definiert. Definition 4.9 führt das Universum UCLASS ein. Anstatt Elemente von UCLASS konstruktiv, zum Beispiel aus *records*, aufzubauen, bleiben diese Elemente abstrakt. Mit Hilfe des Selektors `attr` können jedoch Informationen über die Attribute einer Klasse gewonnen werden. Ein definierendes Merkmal der Objektorientierung ist, dass Objekte eine Identität besitzen. Damit können strukturell gleiche Objekte (im Gegensatz zu *records*) unterschieden werden. Systeme im Systemmodell besitzen daher ein Universum von Objektidentifikatoren UOID. Zu einer Klasse werden mit der Funktion oids eine Menge von Identifikatoren assoziiert. Neben Objektidentifikatoren (oder Objektreferenzen) gibt es noch die Instanzen (INSTANCE), also die eigentlichen Objekte, deren Aufbau man an der Definition der Funktion objects erkennen kann. Eine Instanz besteht aus einem Identifikator und einer Variablenbelegung für die Attribute der zugehörigen Klasse. Objektidentifikatoren gehören laut der Funktion classOf zu genau einer Klasse. Instanzen enthalten immer einen zur Klasse passenden Identifikator, so dass jedes Objekt zu genau einer Klasse gehört. Wie trotzdem Polymorphie der Objekte bei Unterklassenbildung erreicht werden kann, wird später in diesem Abschnitt beschrieben.

Definition SysMod 4.9 (Klassen und Instanzen)

Class

use Type

$$
\begin{array}{l}
\mathrm{UCLASS}\\
\mathrm{UOID}\\
\mathrm{INSTANCE} \subseteq \mathrm{UOID} \times \mathrm{VarAssign}\\
\mathrm{attr} : \mathrm{UCLASS} \to \wp_f(\mathrm{UVAR})\\
\mathrm{oids} : \mathrm{UCLASS} \to \wp(\mathrm{UOID})\\
\mathrm{objects} : \mathrm{UOID} \to \wp(\mathrm{INSTANCE})\\
\mathrm{objectsOfClass} : \mathrm{UCLASS} \to \wp(\mathrm{INSTANCE})\\
\mathrm{classOf} : \mathrm{UOID} \to \mathrm{UCLASS}
\end{array}
$$

$$
\begin{array}{l}
\forall\, C \in \mathrm{UCLASS}, oid \in \mathrm{UOID} :\\
\quad \mathrm{objects}(oid) = \{(oid, r) \mid r \in \mathrm{VarAssign} \wedge \mathrm{dom}(r) = \mathrm{attr}(\mathrm{classOf}(oid))\}\\
\quad \mathrm{objectsOfClass}(C) = \bigcup_{oid \in \mathrm{oids}(C)} \mathrm{objects}(oid)\\
\forall\, oid \in \mathrm{oids}(C) : \mathrm{classOf}(oid) = C\\
\mathrm{INSTANCE} = \bigcup_{C \in \mathrm{UCLASS}} \mathrm{objectsOfClass}(C)
\end{array}
$$

UOID ist das Universum der Objektidentifikatoren, UCLASS das der Klassennamen und INSTANCE ist die Menge aller Objekte, die aus Objektidentifikator und Attributbelegung bestehen. attr liefert die Attribute jeder Klasse. oids ordnet den Klassen Objektidentifikatoren zu. objects bzw. objectsOfClass liefern die Menge aller Instanzen für einen Identifikator bzw. eine Klasse. classOf legt die Beziehung jedes Identifikators zu einer zugehörigen Klasse fest.

UOID und INSTANCE sind statisch und enthalten alle möglichen Objektreferenzen und Instanzen, die prinzipiell in einem laufenden System auftreten können. Diese Mengen sind normalerweise nicht endlich. Um die (typischerweise endliche) Menge der in einem Systemzustand aktuell existierenden Objekte erfassen zu können, benötigen wir die Definition des Datenzustandes, der ebenfalls später in diesem Abschnitt eingeführt wird.

Die Definition 4.10 regelt den Attributzugriff für eine Instanz. getAttr liefert für eine gegebene Instanz und ein gegebenes Attribut den Wert des Attributs. Da jedem Objektidentifikator eine Klasse zugeordnet ist, kann mit attrOid auch über einen Identifikator eindeutig auf die Menge der Attribute zugegriffen werden. Die spezielle Funktion this liefert den Identifikator einer

Definition SysMod 4.10 (Attributzugriff)

Attribute

use Class, Type

$\mathrm{this} : \mathrm{INSTANCE} \to \mathrm{UOID}$
$\mathrm{getAttr} : \mathrm{INSTANCE} \times \mathrm{UVAR} \rightharpoonup \mathrm{UVAL}$
$\mathrm{attrOid} : \mathrm{UOID} \to \wp_f(\mathrm{UVAR})$

Notation:
$o.\mathrm{this}$ ist Abkürzung von $\mathrm{this}(o)$
$o.a$ ist Abkürzung von $\mathrm{getAttr}(o, a)$

$\forall (oid, r) \in \mathrm{INSTANCE}, a \in \mathrm{UVAR} :$
$\quad \mathrm{this}((oid, r)) = oid$
$\quad \mathrm{getAttr}((oid, r), a) = r(a)$
$\quad \mathrm{attrOid}(oid) = \mathrm{attr}(\mathrm{classOf}(oid))$

$o.\mathrm{this}$ wirkt wie ein Attributzugriff. this ist aber tatsächlich kein echtes Attribut einer Klasse.

---

gegebenen Instanz. Es ist gewollt, dass die Notation für this einem normalen Attributzugriff ähnelt. Da this aber kein echtes Attribut ist, brauchen wir this keinen Typ zuzuordnen. Wir haben bisher auch noch nicht festgelegt, ob und wie eine Klasse auch als Typ aufgefasst werden kann.

Definition SysMod 4.11 (Einführung von Nil)

Nil

use Class, Type, Attribute

$\mathrm{Nil} \in \mathrm{UOID}$

$\forall\, C \in \mathrm{UCLASS} : \mathrm{Nil} \notin \mathrm{oids}(C)$
$\forall\, o \in \mathrm{INSTANCE} : o.\mathrm{this} \neq \mathrm{Nil}$
$\mathrm{UOID} = \{\mathrm{Nil}\} \cup \bigcup_{C \in \mathrm{UCLASS}} \mathrm{oids}(C)$

Nil ist ein spezieller Objektidentifikator, der nicht einer Klasse oder einem Objekt zugeordnet ist. UOID enthält nur Identifikatoren und Nil.

In vielen Programmiersprachen wird ein expliziter Wert für die Ungültigkeit einer Objektreferenz vorgesehen. Definition 4.11 führt diesen speziellen Identifikator ein und legt fest, dass Nil kein echter Identifikator einer Klasse ist (und damit nicht als Wert von this vorkommen darf).

In Definition 4.12 vereinbaren wir jetzt, dass alle Klassen mit Hilfe des Typkonstruktors $.^{\&}$ als Typ aufgefasst werden können. Zudem wird in dieser Definition durch die Relation sub ein weiteres grundlegendes Merkmal der Objektorientierung, nämlich die Vererbung oder Unterklassenbildung eingeführt. Klasse C_1 ist eine Unterklasse von Klasse C_2, falls $(C_1, C_2) \in \mathrm{sub}$. Gibt es mehrere solcher Klassen C_2, sprechen wir von Mehrfachvererbung. Wir fordern außerdem die Reflexivität und Transitivität der Relation. Mit Hilfe von sub und dem Konstruktor $.^{\&}$ lässt sich nun auch präzise definieren, welche Trägermenge der Typ einer Klasse besitzen soll. Es handelt sich dabei um alle Objektidentifikatoren, die entweder zur Klasse selbst oder zu einer ihrer Unterklassen gehören. Hierdurch wird eine Substituierbarkeit eines Objektidentifikators einer Klasse durch Identifikatoren von Unterklassen erreicht. Anders ausgedrückt können Identifikatoren von Unterklassen polymorph als Identifikatoren ihrer Oberklassen verwendet werden.

Definition SysMod 4.12 (Subclassing)

Subclassing

use C*lass*, *Nil*, *Type*

$\mathrm{sub} \subseteq \mathrm{UCLASS} \times \mathrm{UCLASS}$
$.^{\&} : \mathrm{UCLASS} \to \mathrm{UTYPE}$

$\mathrm{UOID} \subseteq \mathrm{UVAL}$

$\mathrm{transitive}(\mathrm{sub}) \wedge \mathrm{reflexive}(\mathrm{sub})$
$\forall C \in \mathrm{UCLASS} : \mathrm{CAR}(C^{\&}) = \{\mathrm{Nil}\} \cup \bigcup_{C_1 \,\mathrm{sub}\, C} \mathrm{oids}(C_1)$

sub ist die transitive und reflexive Subklassenrelation. Der Typ $C^{\&}$ enthält alle Objektidentifikatoren der Klasse C und ihrer Unterklassen.

Es ist zu beachten, dass die Subklassenbeziehung zwischen Klassen nicht auf struktureller Kompatibilität der Klassen basiert. Die beteiligten Klassen können ganz unterschiedliche Attributsätze definieren. In der Variante 4.13 wird deshalb gemäß [LW94] gefordert, dass eine Subklasse mindestens die Attribute ihrer Oberklasse enthält, um in struktureller Hinsicht tatsächlich substituierbar zu sein ([LW94] stellt weitergehende, auf dem Verhalten basierende Bedingungen an die Substituierbarkeit).

Variante SysMod 4.13 (Strukturverträgliche Subklassenbildung)

LiskovPrinciple

use Subclassing, Type

$\forall C_1, C_2 \in \mathrm{UCLASS} : C_1 \,\mathrm{sub}\, C_2 \Rightarrow \mathrm{attr}(C_2) \subseteq \mathrm{attr}(C_1)$

Namenskonflikte können bei Mehrfachvererbung von Attributen im Systemmodell nicht auftreten, da wir angenommen haben, dass Attribute global eindeutig sind.

Definition 4.12 legt nur fest, dass die Subklassenbeziehung transitiv und reflexiv sein muss. Um zyklische Vererbungsbeziehungen auszuschließen, kann mit Variante 4.14 die Antisymmetrie der Subklassenbeziehung gefordert werden.

Mehrfachvererbung, die durch sub prinzipiell gestattet ist, kann unerwünscht sein, wenn beispielsweise das Systemmodell Eigenschaften einer Programmiersprache abbilden soll, in der nur einfache Vererbung zur Verfügung steht. Mit Hilfe der Variante 4.15 wird sub entsprechend eingeschränkt.

Variante SysMod 4.14 (Antisymmetrie der Subklassenrelation)

AntiSymSub

use Subclassing, Type

$\forall\, C_1, C_2 \in \mathrm{UCLASS} : C_1 \,\mathrm{sub}\, C_2 \land C_2 \,\mathrm{sub}\, C_1 \Rightarrow C_1 = C_2$

Variante SysMod 4.15 (Einfache Vererbung)

StrictSingleInheritance

use Subclassing, Type

$\forall\, C_1, C_2, C_3 \in \mathrm{UCLASS} : C_1 \,\mathrm{sub}\, C_2 \land C_1 \,\mathrm{sub}\, C_3 \Rightarrow C_2 \,\mathrm{sub}\, C_3 \lor C_3 \,\mathrm{sub}\, C_2$

In Variante 4.16 wird ein weiterer Typkonstruktor für Klassen eingeführt. Dieser Typkonstruktor erlaubt es uns, Objekte (Instanzen) als Werte im System zu betrachten. Nun können neben den standardmäßig verwendeten Objektreferenzen auch die eigentlichen Objekte im System als Werte ausgetauscht werden. In dieser Variante wird keine Substituierbarkeit von Objekten entlang der Subklassenrelation hergestellt. Eine alternative Variante könnte dies anders definieren.

Variante SysMod 4.16 (Objekte als Werte)

ValueObjects

use Subclassing, Type

$\mathrm{INSTANCE} \subseteq \mathrm{UVAL}$

$.^{*} : \mathrm{UCLASS} \to \mathrm{UTYPE}$

$\forall\, C \in \mathrm{UCLASS} : \mathrm{CAR}(C^{*}) = \mathrm{objects}(C)$

Die Definition 4.17 fasst alle Theorien dieses Abschnitts zusammen. Die Variabilität der Theorien wird in Definition 4.18 als Feature-Diagramm dargestellt. Nicht betrachtet wird die Variante *Generics* aus [BCGR08], da Generizität in dieser Arbeit nicht untersucht wird.

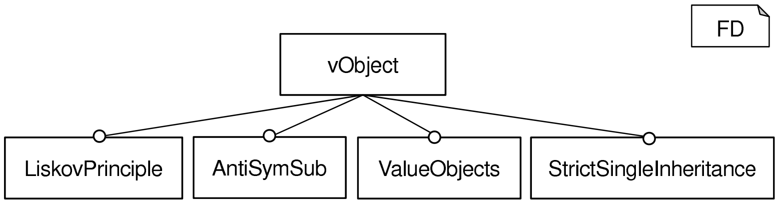
Definition SysMod 4.17 (Object)

Object

extend Class, Attribute, Nil, Subclassing

Definition Variabilität 4.18 (vObject)

vObject



4.4 Datenzustand und Assoziationen

Jedes System im Systemmodell ist eine Zustandsmaschine. Ein Bestandteil des Zustandsraums ist der Datenzustand der Objekte. Ein Datenzustand gemäß Definition 4.19 ist eine Momentaufnahme der aktuellen Datenzustände aller Objekte im System. Für jedes zu diesem Zeitpunkt existierende Objekt enthält der $\mathrm{DataStore}$ die aktuelle Instanz zu einem gegebenen Objektidentifikator. Die Menge der aktuell existierenden Objekte kann mit oids abgefragt werden. Ein Datenzustand ist nur dann gültig, wenn alle Instanzen unter erlaubten Identifikatoren abgelegt sind und zu diesem Identifikator passen.

Definition SysMod 4.19 (Datenzustand)

*DataStore*1

use Object

$\mathrm{DataStore} \subseteq (\mathrm{UOID} \rightharpoonup \mathrm{INSTANCE})$
$\mathrm{oids} : \mathrm{DataStore} \to \wp(\mathrm{UOID})$

$\forall\, ds \in \mathrm{DataStore} : \mathrm{oids}(ds) = \mathrm{dom}(ds)$
$\forall\, ds \in \mathrm{DataStore}, oid \in \mathrm{oids}(ds) :$
$\quad ds(oid).\mathrm{this} = oid \wedge$
$\quad ds(oid) \in \mathrm{objects}(oid)$

$\mathrm{DataStore}$ ist die Menge möglicher Datenzustände. oids ist die Menge der existierenden Objekte für einen gegebenen Datenzustand.

Um den Umgang mit einem Datenzustand zu erleichtern, definieren wir in Definition 4.20 Funktionen zum Auslesen und Aktualisieren des Zustands. Die Funktion val liefert für einen Identifikator und ein Attribut den entsprechenden Attributwert. Die Abbildung ist partiell, da der Identifikator nicht existieren oder das Attribut nicht zur Klasse gehören muss. Ist beides nicht der Fall, wird auf jeden Fall ein Wert geliefert (unter Umständen Nil bei Objektreferenzen). Wir nehmen also an, dass der Wert zu jedem Zeitpunkt definiert ist. Es ist aber möglich, dass der Wert nicht bekannt ist (weil er z.B. nach der Objekterzeugung erst initialisiert werden muss). Die Annahme eines immer definierten Wertes vermeidet den Umgang mit einem expliziten undefinierten Wert. setval aktualisiert, das heißt überschreibt, einen Attributwert mit dem angegebenen und addobj fügt eine neue Instanz in den Datenzustand ein.

Definition SysMod 4.20 (DataStore Infrastruktur)

DataStore

$\mathrm{extend\ Object, DataStore1}$

$\mathrm{val} : \mathrm{DataStore} \times \mathrm{UOID} \times \mathrm{UVAR} \rightharpoonup \mathrm{UVAL}$
$\mathrm{setval} : \mathrm{DataStore} \times \mathrm{UOID} \times \mathrm{UVAR} \times \mathrm{UVAL} \rightharpoonup \mathrm{DataStore}$
$\mathrm{addobj} : \mathrm{DataStore} \times \mathrm{INSTANCE} \to \mathrm{DataStore}$

Notation:
$ds(oid.at)$ ist Abkürzung von $\mathrm{val}(ds, oid, at)$
$ds[oid.at = v]$ ist Abkürzung von $\mathrm{setval}(oid, at, v)$

$\forall\, ds \in \mathrm{DataStore}, oid \in \mathrm{oids}(ds), at \in \mathrm{attr}(oid) :$
$\quad \mathrm{val}(ds, oid, at) = ds(oid).at$
$\quad \forall\, v \in \mathrm{CAR}(\mathrm{vtype}(at)) :$
$\qquad \mathrm{setval}(ds, oid, at, v) = ds \oplus [oid = (oid, \pi_2(ds(oid)) \oplus [at = v])]$
$\forall\, ds \in \mathrm{DataStore}, o \in \mathrm{INSTANCE} :$
$\quad \mathrm{addobj}(ds, o) = ds \oplus [o.\mathrm{this} = o]$

val liefert den Wert für ein gegebenes Objekt und Attribut. setval aktualisiert den Wert für ein gegebenes Objekt und Attribut. addobj fügt ein neues Objekt hinzu. Die Funktionsvereinigung $\oplus$ ist in Anhang A beschrieben.

Normalerweise existiert zu jedem Zeitpunkt im Systemablauf nur eine endliche Menge von Objekten. In Variante 4.21 wird genau dies gefordert.

Variante SysMod 4.21 (Endliche Menge existierender Objekte)

FiniteObjects

$\mathrm{use\ DataStore1}$

$\forall\, ds \in \mathrm{DataStore} : \#(\mathrm{oids}(ds)) \in \mathbb{N}$

Zur Zeit haben wir die Möglichkeit, Attribute von Objekten mit Werten zu belegen. Dies stellt die häufigste Form der Speicherung von Daten in objektbasierten Systemen dar. Daneben gibt es weitere Konzepte wie Konstanten oder statische Attribute. Konstanten brauchen im Systemmodell nicht explizit beschrieben zu werden: Ein Konstantenname repräsentiert einen unveränderlichen Wert und kann anstatt des Wertes verwendet werden. Die Abbildung eines Namens auf einen bestimmten Wert sowie die Sichtbarkeit des Namens sind Teil einer semantischen Abbildung.

Statische Attribute einer Klasse existieren „außerhalb“ von Objekten und werden gemeinsam von allen Objekten einer Klasse verwendet. In Variante 4.22 werden statische Attribute eingeführt. Wir wollen mit dieser Variante keine Implementierung von statischen Attributen vorschlagen, sondern zeigen, dass statische Attribute konzeptuell im Systemmodell beschrieben werden können. Durch die möglicherweise auftretenden Seiteneffekte sollte die Verwendung von statischen Attributen nach Möglichkeit vermieden werden. $\mathrm{StaticC}$ wird als eine spezielle Klasse identifiziert, die alle statischen Attribute $\mathrm{StaticAttr}$ enthält. Zudem gibt es in jedem Datenzustand nur eine Instanz dieser Klasse. Da im Systemmodell Sichtbarkeiten nicht berücksichtigt werden müssen und Attributnamen global eindeutig sind, reicht es, nur diese spezielle Klasse zur Definition aller statischen Attribute zu verwenden. Eine modulare Definition könnte aber eine ähnliche Kodierung für jede Klasse separat vorsehen.

Variante SysMod 4.22 (Statische Attribute)

StaticAttr
use DataStore

$\mathrm{StaticC} \in \mathrm{UCLASS}$
$\mathrm{staticOid} \in \mathrm{UOID}$
$\mathrm{StaticAttr} \subseteq \mathrm{UVAR}$

$\mathrm{oids}(\mathrm{StaticC}) = \{\mathrm{staticOid}\}$
$\forall\, ds \in \mathrm{DataStore} : \mathrm{staticOid} \in \mathrm{oids}(ds)$
$\mathrm{attr}(\mathrm{StaticC}) = \mathrm{StaticAttr}$

4.4.1 Assoziationen

Nachdem mit der Definition des Datenzustandes die nötigen Voraussetzungen geschaffen sind, wenden wir uns weiteren wichtigen Konzepten objektorientierter Modellierungssprachen zu – Assoziationen und Links. Assoziationen stellen Beziehungen zwischen Klassen her. Gerade in der UML gibt es vielfältige Arten und Eigenschaften von Assoziationen. Häufig sind Assoziationen binär, also zwischen zwei Klassen definiert. Sie können jedoch auch beliebig viele Klassen miteinander in Beziehung setzen. Sie können eigenständige Elemente mit beispielsweise eigenen Attributen sein. Oder ihre Enden können als Bestandteile der beteiligten Klassen aufgefasst werden. Je nachdem welche Eigenschaften die Assoziation besitzt, können die Instanzen der Assoziationen, ihre Links zwischen Objekten der beteiligten Klassen, im Datenzustand unterschiedlich gespeichert und abrufbar sein. Um diese Vielfalt adäquat beschreiben zu können, führen wir in diesem Abschnitt auch eine Reihe von Varianten ein, die auf der grundlegenden Definition 4.23 aufbauen.

Definition 4.23 führt das Universum der Assoziation UASSOC ein. Die Funktion $\mathrm{classes}$ liefert die Liste der beteiligten Klassen. Zudem können für eine Assoziation zusätzliche Werte aus $\mathrm{extraVals}$ gespeichert werden. Für eine gegebene Assoziation und einen Datenzustand kann mit Hilfe von relOf die Menge der im Datenzustand gespeicherten Links (Liste von Objektidentifikatoren der beteiligten Klassen und ein zusätzlicher Wert) ermittelt werden. Durch Verwendung von $\mathrm{CAR}(C_i^{\&})$ wird erreicht, dass auch Objekte von Unterklassen verwendet werden können. Assoziation übertragen sich also im Allgemeinen auf Unterklassen.

Definition SysMod 4.23 (Grundlegende Definitionen für Assoziationen)

Association

extend DataStore

UASSOC
$\mathrm{classes} : \mathrm{UASSOC} \to \mathrm{List}(\mathrm{UCLASS})$
$\mathrm{extraVals} : \mathrm{UASSOC} \to \wp(\mathrm{UVAL})$
$\mathrm{relOf} : \mathrm{UASSOC} \times \mathrm{DataStore} \to \wp(\mathrm{List}(\mathrm{UOID}) \times \mathrm{UVAL})$

$$\begin{array}{l}
\forall R \in \mathrm{UASSOC}, ds \in \mathrm{DataStore}, n, C_1, \dots, C_n \in \mathrm{UCLASS} : \\
\quad \mathrm{classes}(R) = [C_1, \dots, C_n] \Rightarrow \\
\qquad \forall\, ([o_1, \dots, o_m], v) \in \mathrm{relOf}(R, ds) : \\
\qquad\quad m = n \wedge \forall\, i \in \{1, \dots, n\} : o_i \in \mathrm{CAR}(C_i^{\&}) \wedge v \in \mathrm{extraVals}(R)
\end{array}$$

UASSOC ist das Universum der Assoziationsnamen. classes beschreibt, welche Klassen eine Assoziation in Beziehung setzt. extraVals ist eine Menge zusätzlicher Werte zur Unterscheidung von Links. relOf ist die Funktion zum Auffinden von Links für eine Assoziation basierend auf einem gegebenen Datenzustand.

Die Funktion zum Auffinden von Links relOf hängt von der konkreten Realisierung der Assoziation ab. Hier gibt es sehr vielfältige Realisierungsvarianten, so dass relOf nicht ausspezifiziert wird. Wir nehmen nur an, dass Links im Datenzustand abgelegt werden und hierfür keine Erweiterung des Datenzustandes notwendig ist. Im Folgenden definieren wir einige Varianten, mit denen Standardfälle von Assoziationen einfach erfasst werden können. Die Varianten zeigen auch, wie Links im Datenzustand gespeichert werden können und damit die Funktion relOf genau definiert werden kann. Wir erhalten hierdurch leider keine vollständige Beschreibung aller möglichen Assoziationsarten und Realisierungsvarianten. Falls keine der angegebenen Varianten ausreicht oder noch vorhandene Unterspezifikation aufgelöst werden muss, muss auf Basis dieser Beispiele das Systemmodell geeignet erweitert werden.

Die meisten Assoziationen sind binär ohne weitere Attribute. In Variante 4.24 werden die erforderlichen Definitionen aufgebaut. An allen binären Assoziationen sind zwei nicht notwendigerweise verschiedene Klassen beteiligt. Es gibt nur einen Extrawert, der ignoriert werden kann (Verwendung der wildcard $*$). Die Signatur von binaryRelOf ist gegenüber relOf vereinfacht, da Links nur durch Paare von Objektidentifikatoren dargestellt werden können. Mit sources und destinations werden zu einer Objektreferenz alle Quellen bzw. Ziele eines Links gefunden, an denen das angegebene Objekt beteiligt ist. Mit Hilfe von binaryAssocMultis können die Multiplizitäten auf beiden Seiten einer binären Assoziation eingeschränkt werden.

In Variante 4.25 wird gezeigt, wie eine binäre „zu-1“-Assoziation einseitig als Attribut realisiert werden kann. Links in einem gegebenen Zustand werden dann aus einem Objekt der Klasse C_1 und dem Wert des angegebenen Attributs *at* berechnet. Damit diese Variante auch für Unterklas-

Variante SysMod 4.24 (Binäre Assoziationen)

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BinaryAssociation}} \\
\hline
\text{use Association} \\
\hline
\text{binaryAssoc} : \text{UASSOC} \to \text{bool} \\
\text{binaryRelOf} : \text{UASSOC} \times \text{DataStore} \to \wp(\text{UOID} \times \text{UOID}) \\
\text{sources}, \text{destinations} : \text{UASSOC} \times \text{DataStore} \times \text{UOID} \to \wp(\text{UOID}) \\
\hline
\forall R \in \text{UASSOC}, ds \in \text{DataStore}, oid_d, oid_s \in \text{UOID} : \\
\quad \text{binaryAssoc}(R) \Rightarrow \\
\qquad \#(\text{classes}(R)) = 2 \wedge \#(\text{extraVals}(R)) = 1 \wedge \\
\qquad \text{binaryRelOf}(R, ds) = \{(oid_1, oid_2) \mid ([oid_1, oid_2], *) \in \text{relOf}(R, ds)\} \\
\qquad \text{sources}(R, ds, oid_d) = \{oid_1 \mid (oid_1, oid_d) \in \text{binaryRelOf}(R, ds)\} \\
\qquad \text{destinations}(R, ds, oid_s) = \{oid_2 \mid (oid_s, oid_2) \in \text{binaryRelOf}(R, ds)\} \\
\hline
\text{binaryAssocMultis} : \text{UASSOC} \to \wp(\mathbb{N}) \to \wp(\mathbb{N}) \to \text{bool} \\
\hline
\forall R \in \text{UASSOC}, N_1, N_2 \subseteq \mathbb{N}, ds \in \text{DataStore} : \\
\quad \text{binaryAssocMultis}(R, N_1, N_2) \Rightarrow \\
\qquad \text{binaryAssoc}(R) \wedge \\
\qquad (\exists\, C_1, C2 : [C_1, C_2] = \text{classes}(R) \wedge \forall\, o_1 \in \text{CAR}(C_1^{\&}), o_2 \in \text{CAR}(C_2^{\&}) : \\
\qquad\quad \#(\text{destinations}(R, ds, o_1)) \in N_2 \wedge \#(\text{sources}(R, ds, o_2)) \in N_1)) \\
\hline
\end{array}
$$

Variante SysMod 4.25 (Realisierung einer Assoziation einseitig als Attribut)

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AttributeAssociation}} \\
\hline
\text{use Association}, \text{BinaryAssociation}, \text{LiskovPrinciple} \\
\hline
\text{attributeAssoc} : \text{UASSOC} \times \text{UVAR} \to \text{bool} \\
\hline
\forall R \in \text{UASSOC}, ds \in \text{DataStore}, at \in \text{UVAR} : \\
\quad \text{attributeAssoc}(R, at) \Rightarrow \\
\qquad \text{binaryAssoc}(R) \wedge \\
\qquad (\exists\, C_1, C_2 : [C_1, C_2] = \text{classes}(R) \wedge at \in \text{attr}(C_1) \wedge \text{vtype}(at) = C_2^{\&} \wedge \\
\qquad\; \text{binaryRelOf}(R, ds) = \{(o_1, ds(o_1.at)) \mid o_1 \in \text{CAR}(C_1^{\&}) \cap \text{oids}(ds)\}) \\
\hline
\end{array}
$$

sen von C_1 funktioniert, wird zusätzlich Variante 4.13 benötigt. Damit ist sichergestellt, dass Unterklassen ebenfalls das die Assoziation definierende Attribut enthalten.

Eine Realisierung über Assoziationsklassen für „*-zu-*“-Assoziationen enthält Variante 4.26. Das Prädikat mtomAssoc fordert die Existenz einer speziellen Assoziationsklasse M mit zwei Attributen, die die beteiligten Objekte speichern. Der Mechanismus lässt sich auf Assoziationen beliebiger Arität erweitern.

Die letzten Varianten 4.27, 4.28 und 4.29 enthalten Umsetzungsvorschläge für Kompositio-

Variante SysMod 4.26 (Realisierung einer Assoziation durch eine Assoziationsklasse)

MtoMAssociation

$$\begin{array}{l}
\text{use Association, BinaryAssociation} \\
\hline
\text{mtomAssoc} : \text{UASSOC} \times (\text{UCLASS} \times \text{UVAR} \times \text{UVAR}) \rightarrow \text{bool} \\
\hline
\forall R \in \text{UASSOC}, ds \in \text{DataStore}, M \in \text{UCLASS}, a, b \in \text{UVAR} : \\
\quad \text{mtomAssoc}(R, (M, a, b)) \Rightarrow \\
\quad\quad \text{binaryAssoc}(R) \wedge a, b \in \text{attr}(M) \wedge \\
\quad\quad (\exists\, C_1, C_2 : [C_1, C_2] = \text{classes}(R) \wedge \text{vtype}(a) = C_1^{\&} \wedge \text{vtype}(b) = C_2^{\&} \wedge \\
\quad\quad\; \text{binaryRelOf}(R, ds) = \{(ds(m.a), ds(m.b)) \mid m \in \text{CAR}(M^{\&}) \cap \text{oids}(ds)\})
\end{array}$$

nen, abgeleitete und qualifizierte Assoziationen. Für qualifizierte und abgeleitete Assoziationen werden im Wesentlichen nur die notwendigen Signaturen vorgestellt und auf eine ausführlichere Behandlung verzichtet. Ein Beispiel für die Umsetzung qualifizierter Assoziationen über ein Attribut ist in [BCGR08] angegeben.

Wir sprechen von einer Komposition (Variante 4.27), falls die rechte Seite eines Links aus dem Datenzustand entfernt wird, sobald die zugehörige linke Seite entfernt wird. Die Lebenszeit des zweiten Objekts ist also an die des ersten gekoppelt.

Variante SysMod 4.27 (Kompositionen)

CompositeAssociation

$$\begin{array}{l}
\text{use Association, BinaryAssociation} \\
\hline
\text{compositeAssoc} : \text{UASSOC} \rightarrow \text{bool} \\
\hline
\forall R \in \text{UASSOC} : \text{compositeAssoc}(R) \Rightarrow \\
\quad \forall\, ds_1, ds_2 \in \text{DataStore}, o_1, o_2 \in \text{UOID} : \\
\quad\quad (o_1, o_2) \in \text{binaryRelOf}(R, ds_1) \wedge o_1 \notin \text{oids}(ds_2) \Rightarrow o_2 \notin \text{oids}(ds_2)
\end{array}$$

Links abgeleiteter Assoziationen (Variante 4.28) werden auf Basis einer irgendwie gearteten Funktion f berechnet. Weitere Eigenschaften von f bleiben unterspezifiziert.

Variante SysMod 4.28 (Abgeleitete Assoziationen)

DerivedAssociation

$$\begin{array}{l}
\text{use Association, BinaryAssociation} \\
\hline
\text{derivedAssoc} : \text{UASSOC} \times (\text{DataStore} \rightarrow \wp(\text{List}(\text{UOID}) \times \text{UVAL})) \rightarrow \text{bool} \\
\hline
\forall R \in \text{UASSOC}, ds \in \text{DataStore}, f : \text{derivedAssoc}(R, f) \Rightarrow \text{relOf}(R, ds) = f(ds)
\end{array}$$

Binäre Assoziationen können an ihren Enden qualifiziert sein. Ist beispielsweise die rechte

Seite qualifiziert (Variante 4.29) und gibt es einen Link zur rechten Seite, so gibt es auch einen Wert aus einer gegebenen Menge, der diese rechte Seite eindeutig identifiziert (mit Hilfe von qualifiedRelOf). Ein Beispiel für eine solche Qualifikation ist das Speichern der Objekte in einer Liste, wobei der Listenindex als Qualifikator genau ein Objekt identifiziert.

Variante SysMod 4.29 (Qualifizierte Assoziationen)

QualifiedAssociation

$$\text{use Association, BinaryAssociation}$$

$$\begin{aligned}
&\text{rightQualifiedAssoc} : \text{UASSOC} \times \wp(\text{UVAL}) \to \text{bool}\\
&\text{qualifiedRelOf} : \text{UASSOC} \times \text{DataStore} \to \wp(\text{UOID} \times \text{UVAL} \times \text{UOID})
\end{aligned}$$

$$\begin{aligned}
&\forall R \in \text{UASSOC}, \mathit{values} \subseteq \text{UVAL} : \quad \text{rightQualifiedAssoc}(R, \mathit{values}) \Rightarrow\\
&\quad \text{binaryAssoc}(R) \wedge\\
&\quad (\exists\, C_1, C_2 : [C_1, C_2] = \text{classes}(R) \wedge \forall\, ds \in \text{DataStore}, o_1 \in \text{CAR}(C_1^{\&}) :\\
&\quad\; \forall\, o_2 \in \text{CAR}(C_2^{\&}) : (o_1, o_2) \in \text{binaryRelOf}(R, ds) \Rightarrow\\
&\quad\quad \exists^1\, v \in \mathit{values} : (o_1, v, o_2) \in \text{qualifiedRelOf}(R, ds))
\end{aligned}$$

In Definition 4.30 fassen wir die Theorien dieses Abschnitts zusammen.

Definition SysMod 4.30 (Data)

Data

$$\text{extend DataStore1, DataStore, Association}$$

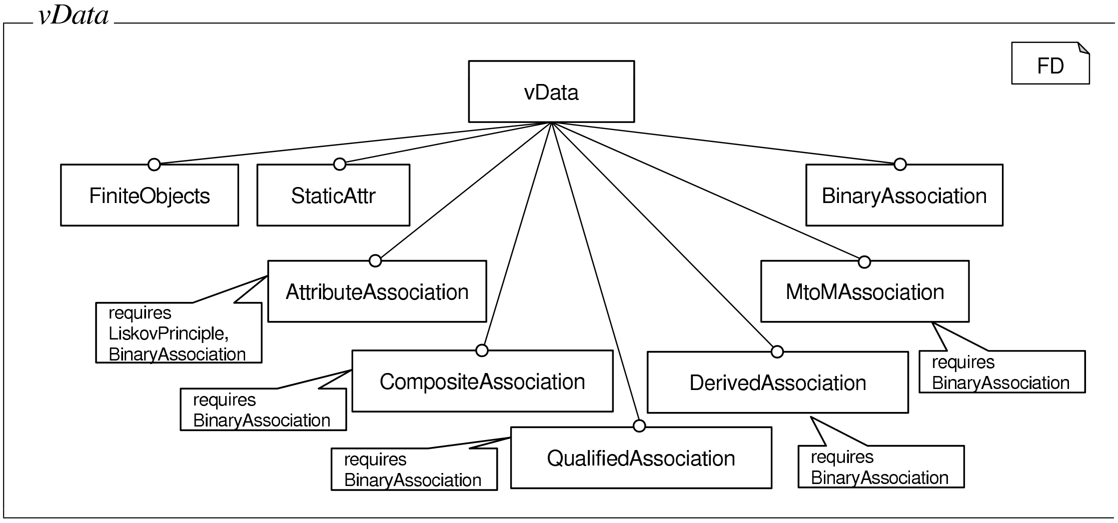
Die beschriebenen Varianten und ihre Abhängigkeiten werden im Variationspunkt 4.31 dargestellt. Die Assoziationsvarianten führen im Wesentlichen neue Funktionen im Systemmodell ein. Eine Konfiguration stellt die Menge aller gültigen Definitionen dar. Erst eine ausgewählte Assoziationsvariante erlaubt also die Verwendung der von ihr eingeführten Funktionen in einer Systemmodell-Konfiguration. Durch die bloße Hinzunahme von Funktionen zum Systemmodell ist noch nichts über deren Nutzung ausgesagt. Im Fall der Assoziationsvarianten ist also noch nicht bestimmt, welche Assoziation wie realisiert wird. Dies stellt eine weitere Variabilität bei der Nutzung der Funktionen, insbesondere durch semantische Abbildungen, dar.

Weitere Varianten für Referenzen und Locations, wie sie in [BCGR08] eingeführt werden, werden hier nicht betrachtet. Statt der Varianten für einfache Assoziationen und geordnete Assoziationen nehmen wir die Variante für abgeleitete Assoziationen auf.

4.5 Methoden, Threads und Kontrollzustand

Im vorherigen Abschnitt haben wir den Datenzustand, zusammen mit allen dafür benötigten statischen Elementen eines Systems eingeführt. In diesem Abschnitt wird als zweite Zustandskomponente der Kontrollzustand des Systemmodells, ebenfalls mit allen dafür benötigen statischen

Definition Variabilität 4.31 (vData)


vData
FD
vData
FiniteObjects
StaticAttr
BinaryAssociation
AttributeAssociation
MtoMAssociation
requires LiskovPrinciple, BinaryAssociation
CompositeAssociation
DerivedAssociation
requires BinaryAssociation
requires BinaryAssociation
QualifiedAssociation
requires BinaryAssociation
requires BinaryAssociation

Bestandteilen, definiert.

Im Kontrollteil des Systemmodells beschäftigen wir uns mit der Definition von Operationen und Methoden von Klassen und ihrer potentiell nebenläufigen Ausführung durch Threads. Der Kontrollzustand enthält daher Informationen über den Zustand aktuell ausgeführter Methoden der Objekte.

In Definition 4.32 werden Operationen aus dem Universum UOPN definiert. Operationen legen fest, wie ein Verhalten eines Objekts initiiert werden kann. Für jede Operation wird definiert, zu welcher Klasse sie gehört, welchen Namen, welche Parametertypen und welchen Ergebnistyp sie besitzt. Somit stellt eine Operation eine Signatur dar. Das Verhalten selbst ist nicht der Teil Operationsdefinition, sondern wird separat in einer Methode (siehe Definition 4.35) spezifiziert. Parameternamen gehören zur Implementierung (Methode) einer Operation und sind nicht Teil der Signatur.

Wir legen nicht fest, ob und wie die Vererbung von Operationen (und Implementierungen) an Unterklassen geschieht, ob Operationen überladen werden können oder ob eine statische oder dynamische Bindung eines Methodenaufrufs erfolgt. Solche Eigenschaften können in einer semantischen Abbildung beschrieben werden. Damit eine Substituierbarkeit der Objekte bei Unterklassenbildung auch bezüglich ihrer Verhaltenssignatur sichergestellt ist, müssen Unterklassen zu allen Operationen der Oberklasse eine kompatible Operation zur Verfügung stellen. Die bekannteste Art dieser Kompatibilität ist die zum Beispiel in [Mey97] beschriebene Kontravarianz der Parametertypen und Kovarianz der Rückgabetypen. Das bedeutet, dass für jede Operation der Oberklasse eine gleichnamige Operation in der Unterklasse existiert, die mindestens alle Parameter der Operation der Oberklasse akzeptiert und höchstens alle ihre Rückgabewerte zurück liefern kann. Dieser Sachverhalt ist in Variante 4.33 formalisiert. In einer strengeren Variante, die keine Spezialisierung der Signatur erlaubt, müssen Parameter und Rückgabewerte gleichgesetzt werden.

Definition SysMod 4.32 (Operationen)

Operation

extend Object

UOPN
$\mathrm{nameOf} : \mathrm{UOPN} \to \mathrm{Name}$
$\mathrm{classOf} : \mathrm{UOPN} \to \mathrm{UCLASS}$
$\mathrm{parTypes} : \mathrm{UOPN} \to \mathrm{List}(\mathrm{UTYPE})$
$\mathrm{resType} : \mathrm{UOPN} \to \mathrm{UTYPE}$
$\mathrm{params} : \mathrm{UOPN} \to \wp(\mathrm{List}(\mathrm{UVAL}))$

$\forall\, op \in \mathrm{UOPN}, n, t_1, \ldots, t_n \in \mathrm{UTYPE} :$
$\quad \mathrm{parTypes}(op) = [t_1, \ldots, t_n] \Rightarrow \mathrm{params}(op) = \{[v_1, \ldots, v_n] \mid v_i \in \mathrm{CAR}(t_i), i \leq n\}$

UOPN ist das Universum der Operationen, nameOf liefert den Namen einer Operation, classOf liefert die Klasse, zu der die Operation gehört. parTypes ist die Liste der Parametertypen, resType ist der Rückgabetyp der Operation und params liefert alle möglichen Argumente einer Operation.

Variante SysMod 4.33 (Typsicheres Vererben von Operationen)

TypeSafeOps

extend Operation

$\forall\, op_1 \in \mathrm{UOPN}, c \in \mathrm{UCLASS} : c\,\mathrm{sub}\,\mathrm{classOf}(op_1) \Rightarrow$
$\quad \exists\, op_2 \in \mathrm{UOPN} : \mathrm{classOf}(op_2) = c \,\wedge$
$\qquad \mathrm{nameOf}(op_1) = \mathrm{nameOf}(op_2) \,\wedge$
$\qquad \mathrm{params}(op_1) \subseteq \mathrm{params}(op_2) \,\wedge$
$\qquad \mathrm{CAR}(\mathrm{resType}(op_1)) \supseteq \mathrm{CAR}(\mathrm{resType}(op_2))$

Für jede Subklasse werden die Operationen typsicher vererbt.

Während Operationen nur die Signatur angeben, besitzen Methoden aus dem Universum UMETH (Definition 4.34) eine Signatur und eine Implementierung. Die Signatur einer Methode mit Namen, Parameter- und Rückgabetypen entspricht der Signatur von Operationen, wenn die Parameter auf ihre Typen projiziert werden.

Um ein genaues Verständnis einer Methodenausführung zu entwickeln, benötigen wir ein Universum von Programmzählern UPC und ordnen jeder Methode eine endliche Menge dieser Zähler zu. In Methoden können lokale Variablen deklariert werden (localNames). Alle möglichen Werte von Parametern (parOf) und lokalen Variablen (localsOf) werden in Variablenbelegungen gespeichert.

In Definition 4.35 definieren wir den Zusammenhang zwischen Operationen und Methoden. Die partielle Funktion impl ordnet Operationen eine Methode zu. Falls m die Implementierung

Definition SysMod 4.34 (Methoden)

*Method*1

extend Operation

$$\begin{array}{l}
\mathrm{UMETH} \\
\mathrm{UPC} \\
\mathrm{nameOf} : \mathrm{UMETH} \to \mathrm{Name} \\
\mathrm{definedIn} : \mathrm{UMETH} \to \mathrm{UCLASS} \\
\mathrm{parNames}, \mathrm{localNames} : \mathrm{UMETH} \to \mathrm{List}(\mathrm{UVAR}) \\
\mathrm{parOf}, \mathrm{localsOf} : \mathrm{UMETH} \to \wp(\mathrm{VarAssign}) \\
\mathrm{resType} : \mathrm{UMETH} \to \mathrm{UTYPE} \\
\mathrm{pcOf} : \mathrm{UMETH} \to \wp_f(\mathrm{UPC})
\end{array}$$

$$\begin{array}{l}
\forall\, m \in \mathrm{UMETH} : \\
\quad \mathrm{parOf}(m) = \{ va \in \mathrm{VarAssign} \mid \mathrm{dom}(va) = \mathrm{parNames}(m) \} \\
\quad \mathrm{localsOf}(m) = \{ va \in \mathrm{VarAssign} \mid \mathrm{dom}(va) = \mathrm{localNames}(m) \} \\
\quad \mathrm{parNames}(m) \cap \mathrm{localNames}(m) = \varnothing \\
\quad \mathrm{parNames}(m) \cap \mathrm{attr}(\mathrm{definedIn}(m)) = \varnothing \\
\quad \mathrm{localNames}(m) \cap \mathrm{attr}(\mathrm{definedIn}(m)) = \varnothing
\end{array}$$

UMETH ist das Universum der Methoden, UPC ist das Universum der Programmzähler, definedIn liefert die Klasse, in der die Methode definiert ist, parNames sind die formalen Parameter, localNames ist die Menge der lokalen Variablen, parOf sind alle gültigen Variablenbelegungen der Parameter, localsOf sind entsprechend die Belegungen der lokalen Variablen, resType ist der Rückgabetyp einer Methode, pcOf ist die Menge der möglichen Programmzählerwerte einer Methode. Zur Vereinfachung nehmen wir an, dass die Mengen der Namen von Parametern, lokalen Variablen und Attributen disjunkt sind. Dies kann syntaktisch sichergestellt werden.

einer Operation *op* ist, dann stimmen die Namen überein, *op* ist in derselben oder einer Unterklasse definiert und hat die gleiche oder eine speziellere Signatur als die Methode. Da wir erlauben, dass die Operation eine speziellere Signatur haben darf als die Methode (die Methode ist robuster), kann eine überschreibende Operation mit spezialisierter Signatur sogar dieselbe Implementierung verwenden. Natürlich kann die Implementierung einer Operation in einer Unterklasse auch durch eine weitere Methode überschrieben werden. Wird Variante 4.33 verwendet, ist die Substituierbarkeit trotzdem gesichert, da die Operationen kompatibel sind.

An dieser Stelle bietet es sich auch an zu definieren, was wir unter einer Schnittstelle (einem *Interface*) im Systemmodell verstehen wollen. Hierfür führen wir kein weiteres Universum ein, sondern verwenden Klassen aus dem Universum UCLASS, die bestimmte Eigenschaften aufweisen: Ein Interface kann selbst nicht direkt instantiiert werden, denn keine Operation des Interface ist implementiert.

Als Erweiterung des strikten Verbots der Mehrfachvererbung in Variante 4.15 definieren wir die Variante 4.36. In dieser Variante darf nicht von mehreren Klassen, wohl aber von mehreren

Definition SysMod 4.35 (Zusammenhang zwischen Methoden und Operationen)

Method

extend Method1

$$\begin{array}{l}
\text{impl} : \text{UOPN} \rightharpoonup \text{UMETH} \\
\text{isInterface} : \text{UCLASS} \rightarrow \text{bool}
\end{array}$$

$$\begin{array}{l}
\forall\, op \in \text{UOPN}, m \in \text{UMETH} : m = \text{impl}(op) \Rightarrow \\
\quad \text{nameOf}(m) = \text{nameOf}(op) \wedge \\
\quad \text{classOf}(op)\ \text{sub}\ \text{definedIn}(m) \wedge \\
\quad \text{CAR}(\text{resType}(m)) \subseteq \text{CAR}(\text{resType}(op)) \wedge \\
\quad \text{conv}(\text{parNames}(m), \text{parOf}(m)) \supseteq \text{params}(op) \\
\forall\, op \in \text{UOPN} : \\
\quad (\exists\, C \in \text{UCLASS} : \text{classOf}(op) = C \wedge \text{oids}(C) \neq \varnothing) \Rightarrow op \in \text{dom}(\text{impl})
\end{array}$$

$$\begin{array}{l}
\text{conv} : \text{List}(\text{UVAR}) \times \wp(\text{VarAssign}) \rightarrow \wp(\text{List}(\text{UVAL})) \\
\forall\, nas \in \text{List}(\text{UVAR}), vas \in \wp(\text{VarAssign}) : \\
\quad \text{conv}(nas, vas) = \{ l \mid \exists\, va \in vas : \forall\, i : l[i] = va(nas[i]) \}
\end{array}$$

$$\begin{array}{l}
\forall\, C \in \text{UCLASS} : \text{isInterface}\ C \Rightarrow \\
\quad \text{oids}(C) = \varnothing \wedge (\forall\, op \in \text{UOPN} : \text{classOf}(op) = C \Rightarrow op \notin \text{dom}(\text{impl}))
\end{array}$$

impl ordnet Operationen eine Methode zu. Für alle Operationen, die zu einer Klasse gehören, deren Objektidentifikatormenge nicht leer ist, muss die partielle Funktion impl definiert sein. Alle Operationen von Interfaces sind nicht implementiert.

Interfaces geerbt werden.

Variante SysMod 4.36 (Mehrfachvererbung nur für Interfaces)

ClassSingleInheritance

use Method

$$\begin{array}{l}
\forall\, C_1, C_2, C_3 \in \text{UCLASS} : C_1\ \text{sub}\ C_2 \wedge C_1\ \text{sub}\ C_3 \Rightarrow \\
\quad C_2\ \text{sub}\ C_3 \vee C_3\ \text{sub}\ C_2 \vee \text{isInterface}\ C_2 \vee \text{isInterface}\ C_3
\end{array}$$

Analog zu statischen Attributen werden statische Methoden behandelt. Diese werden in Variante 4.37 eingeführt. Dabei werden die statischen Methoden von derselben Klasse StaticC zur Verfügung gestellt, die auch schon für die Definition statischer Attribute genutzt wurde. Wiederum gilt, dass wir von einer global eindeutigen Menge von statischen Methoden ausgehen, die dann in dieser Klasse zusammengefasst werden können.

Um die Ausführung von Methoden zu beschreiben, können wir von einem Kontrollstack nicht abstrahieren. In diesem Kontrollstack wird ein Berechnungszustand einer Methode in einem *Frame* zwischengespeichert. Gemäß Definition 4.38 besteht ein Frame aus sechs Komponenten: der

Variante SysMod 4.37 (Statische Methoden)

StaticOpn

use Method, StaticAttr

$$\mathrm{StaticOpn} \subseteq \mathrm{UOPN}$$

$$\begin{aligned}
&\forall\, op \in \mathrm{StaticOpn} : \mathrm{classOf}\, op = \mathrm{StaticC} \wedge \\
&\quad \exists\, m \in \mathrm{UMETH} : \mathrm{impl}\, op = m \wedge \mathrm{definedIn}\, m = \mathrm{StaticC}
\end{aligned}$$

aufgerufenen Objektreferenz, dem Methodennamen, der aktuellen Belegung für Parameter und lokale Variablen, dem aktuellen Programmzähler und der aufrufenden Objektreferenz. Die einzelnen Komponenten können unter entsprechenden Namen wie caller und callee angesprochen werden. Mit framesOf wird die Menge aller möglichen Frames für eine Methode charakterisiert.

Definition SysMod 4.38 (Stack Frames)

StackFrame

extend Method

$$\begin{aligned}
&\mathrm{FRAME} = \mathrm{UOID} \times \mathrm{Name} \times \mathrm{VarAssign} \times \mathrm{VarAssign} \times \mathrm{UPC} \times \mathrm{UOID} \\
&\mathrm{framesOf} : \mathrm{UMETH} \to \wp(\mathrm{FRAME})
\end{aligned}$$

$$\begin{aligned}
&\forall\, m \in \mathrm{UMETH} : \\
&\quad \mathrm{framesOf}(m) = \{(callee, \mathrm{nameOf}(m), pars, locals, pc, caller) \mid \\
&\qquad \exists\, op \in \mathrm{UOPN} : m = \mathrm{impl}(op) \wedge \\
&\qquad callee \in \mathrm{oids}(\mathrm{classOf}(op)) \wedge pc \in \mathrm{pcOf}(m) \wedge \\
&\qquad pars \in \mathrm{parOf}(m) \wedge locals \in \mathrm{localsOf}(m)\}
\end{aligned}$$

FRAME sind die möglichen Frames für die Methoden. Ein Frame speichert das aufgerufene Objekt, den Namen der aufgerufenen Methode, Parameter und lokale Variablen, den Programmzähler und den Aufrufer. framesOf ist die Menge der möglichen Frames für eine gegebene Methode.

In einem sequentiellen System ist ein Stack ausreichend, um rekursive Methodenaufrufe zu beschreiben. Im Systemmodell betrachten wir aber auch verteilte, nebenläufige Abläufe. Wir benötigen zunächst das Universum UTHREAD, das uns abstrakt eine Menge von Threads zur Verfügung stellt.

Die intuitive Vorstellung ist nun, dass Threads nebenläufig in Objekten aktiv sind. Dabei können sowohl mehrere Threads in einem Objekt aktiv sein, als auch sich die Aktivität eines Threads über mehrere Objekte verteilen. Zu jedem Objekt und Thread gehört demnach jeweils ein Stack von Frames, der die von diesem Thread auf diesem Objekt (direkt oder indirekt über weitere Objekte) rekursiv aufgerufenen Methoden speichert. Genau diese Vorstellung ist in Definition 4.40 formalisiert, in der der Kontrollzustand eingeführt wird. Die angegebene Bedingung stellt si-

Definition SysMod 4.39 (Threads)

Thread

extend StackFrame

UTHREAD

UTHREAD ist das Universum der Threads.

cher, dass für jeden Frame, der unter einem bestimmten Objekt *oid* und Thread zu finden ist, das aufgerufene Objekt dem Objekt *oid* entspricht. Zudem gehört zu jedem aufrufenden auch ein aufgerufenes Objekt.

Definition SysMod 4.40 (Der Kontrollzustand in objektorientierter Sicht)

ControlStore

extend Thread

$\mathrm{ControlStore} \subseteq (\mathrm{UOID} \rightharpoonup \mathrm{UTHREAD} \rightharpoonup \mathrm{Stack}(\mathrm{FRAME}))$

$$\begin{array}{l}
\forall\, ctls \in \mathrm{ControlStore}, oid \in \mathrm{dom}(ctls), th \in \mathrm{dom}(ctls)(oid) :\\
\quad \forall\, i < \#ctls(oid)(th) :\\
\qquad \mathrm{callee}(ctls(oid)(th)[i]) = oid \wedge\\
\qquad (\exists\, oid2, j : \mathrm{caller}(ctls(oid)(th)[i]) = oid2 = \mathrm{callee}(ctls(oid2)(th)[j]) = oid)
\end{array}$$

ControlStore teilt jeden Stack gemäß den dazugehörigen Objekten auf.

Die Abbildung 4.41 illustriert einen Kontrollzustand für zwei Objekte A und B. Es gibt zwei Threads, wobei Thread1 sowohl in A als auch in B Methoden ausführt. Frame1.3 ist für Thread1 der aktive Frame. Sobald hier die Ausführung beendet ist, kann Frame1.3 vom Stapelspeicher von A und Thread1 entfernt werden. Wir nehmen an, dass Frame1.1 eine Information enthält (z.B. kodiert in den Programmzähler), dass noch auf einen Rückgabewert von Frame1.2 gewartet werden muss, obwohl für A Frame1.1 der aktive Frame ist.

Um Systeme im Systemmodell auf sequentielle Systeme zu beschränken, kann Variante 4.42 verwendet werden.

Im Gegensatz dazu formalisiert Variante 4.43, dass alle Objekte aktiv sind und keine Methodenaufrufe zwischen Objekten stattfinden (asynchroner Nachrichtenaustausch wird im nächsten Abschnitt definiert). Jedem Objekt ist also eindeutig genau ein Thread zugeordnet, so dass wir die Universen UOID und UTHREAD gleichsetzen.

Es ist nicht sinnvoll, die Varianten 4.43 und 4.42 gemeinsam zu verwenden, da dann nur Systeme mit höchstens einem Objekt und nur interner Kommunikation charakterisiert werden können.

Definition 4.44 fasst die Theorien dieses Abschnitts zusammen. Der Kontrollzustand erlaubt uns eine Beschreibung der Methodenausführung. Die zugehörigen Methodenaufrufe und Rück-

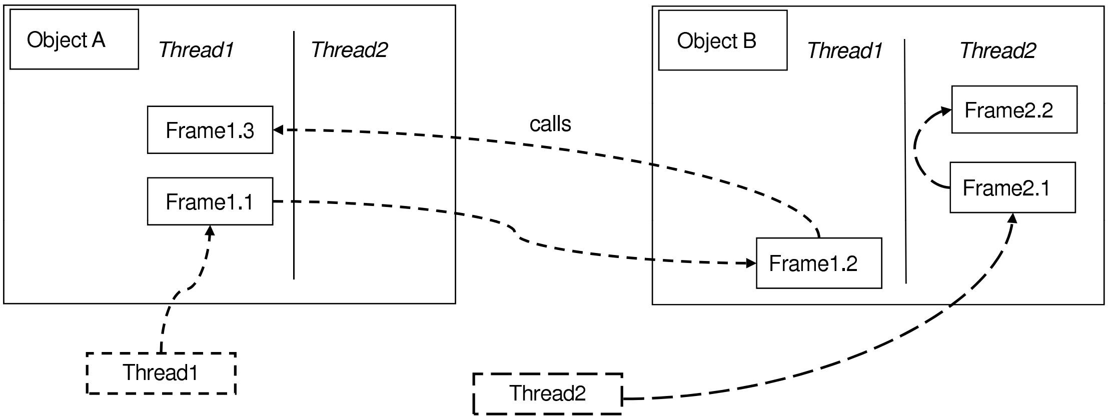

Object A
Thread1
Thread2
Frame1.3
Frame1.1
calls
Object B
Thread1
Thread2
Frame2.2
Frame2.1
Frame1.2
Thread1
Thread2

Abbildung 4.41: Nebenläufig, über zwei Objekte verteilt ausführende Threads [BCGR08]

Variante SysMod 4.42 (Sequentielle Systeme)

SingleThread

use Thread

$\#(\mathrm{UTHREAD}) = 1$

Variante SysMod 4.43 (Aktive Objekte)

ActiveObjects

use ControlStore

$\mathrm{UTHREAD} = \mathrm{UOID}$

$\forall\, cs \in \mathrm{ControlStore}, oid \in \mathrm{UOID}, n :$
$\quad cs(oid)(oid)[n] = (oid, *, *, *, *, oid)$

Aufrufer und aufgerufenes Objekt entsprechen immer dem Objekt selbst. Methodenaufrufe können damit höchstens Objekt-intern durchgeführt werden.

gaben werden im nächsten Abschnitt eingeführt. Die Variabilität ist in Definition 4.45 zusammengestellt.

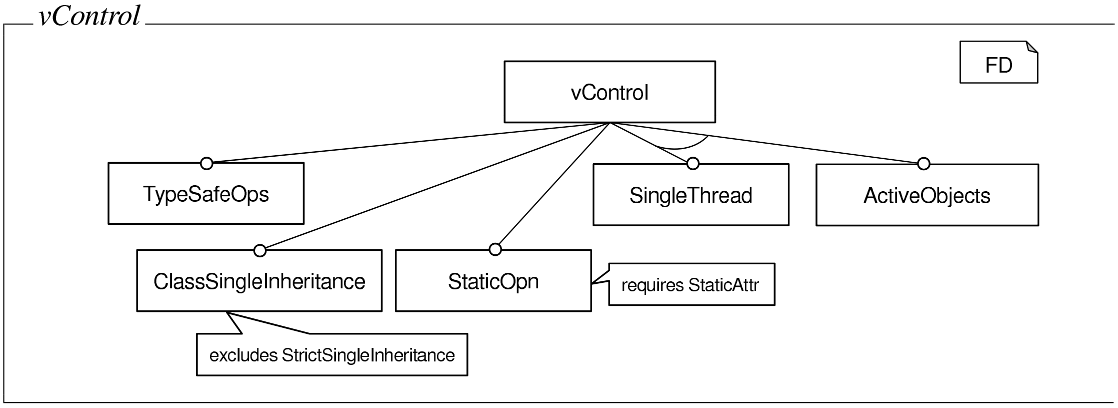
Definition SysMod 4.44 (Control)

Control

extend Operation, Method1, Method, StackFrame, Thread, ControlStore

Im Vergleich zum Datenteil fällt auf, dass dieser Abschnitt weniger ausführliche und deutlich

Definition Variabilität 4.45 (vControl)



unterspezifizierte Definitionen enthält. Dies ist der großen Vielfalt der Beschreibungsmöglichkeiten von nebenläufigen Systemen geschuldet. Allgemein verwendbare Varianten oder spezifischere Definitionen sind nicht offensichtlich. Allerdings bietet die Theorie *Control* Potential für Erweiterungen. Angesprochen wurden z.B. die Formalisierungen von Methodenbindungen (statisch, dynamisch) oder eine weitere Präzisierung der Nebenläufigkeit (z.B. durch Regionen aktiver Objekte wie sie in [BCGR08] skizziert wird).

4.6 Nachrichten und Nachrichtenzustand

Als letzten Teil des Zustandsraums betrachten wir den Nachrichtenpuffer von Objekten. Jegliche Interaktion zwischen Objekten im Systemmodell findet über den Austausch von Nachrichten statt. Aufrufe und Rückgaben (Returns) von Methoden werden genau wie asynchrone Signale als Nachrichten kodiert.

Gemäß Definition 4.46 bestehen Nachrichten aus fünf Komponenten: dem Absender und Empfänger der Nachricht, einem Namen (z.B. der aufgerufenen Methode), einer Liste von Werten und einem Thread, mit dem eine Kontrollflussübergabe stattfinden kann. Der Thread wird daher speziell bei Nachrichten benötigt, die Methodenaufrufe oder -rückgaben kodieren. Für asynchrone Nachrichten kann er entfallen oder ein Dummy-Wert eingesetzt werden.

Da jede Nachricht genau einen Empfänger besitzt, ist es auf dieser Ebene nicht möglich, als Adressaten der Nachricht viele oder alle Objekte (Multi- oder Broadcast) auszuwählen. Hierfür wird eine zusätzliche Infrastruktur benötigt, die hier nicht weiter betrachtet wird.

Der Nachrichtenzustand MessageStore speichert für Objekte die noch zur Verarbeitung anstehenden Nachrichten in einem Puffer. Wir nehmen an, dass ein Puffer eine gegebene Datenstruktur zum Speichern und Auffinden von Nachrichten ist. Im einfachsten Fall handelt es sich um eine Warteschlange (FIFO Queue), es sind jedoch auch kompliziertere Strukturen, z.B. mit Priorisierung oder Neuanordnung gemäß einer Schedulingstrategie denkbar.

Häufig benötigte Nachrichten sind Methodenaufrufe oder Returns. Methodenaufrufe werden in Definition 4.47 beschrieben und sind normale Nachrichten, bei denen der Name dem Namen

Definition SysMod 4.46 (Nachrichten und Nachrichtenzustand)

Message

use Object, Control

$$\begin{aligned}
&\mathrm{MESSAGE} \subseteq \mathrm{UOID} \times \mathrm{Name} \times \mathrm{List(UVAL)} \times \mathrm{UOID} \times \mathrm{UTHREAD}\\
&\mathrm{msgIn}, \mathrm{msgOut} : \mathrm{UOID} \to \wp(\mathrm{MESSAGE})\\
&\mathrm{messages} : \mathrm{UOID} \to \wp(\mathrm{MESSAGE})\\
&\mathrm{MessageStore} \subseteq (\mathrm{UOID} \rightharpoonup \mathrm{Buffer(MESSAGE)})
\end{aligned}$$

$$\begin{aligned}
&\forall\, oid \in \mathrm{UOID} :\\
&\quad \mathrm{msgIn}(oid) = \{m \mid \mathrm{receiver}(m) = oid\}\\
&\quad \mathrm{msgOut}(oid) = \{m \mid \mathrm{sender}(m) = oid\}\\
&\quad \mathrm{messages}(oid) = \mathrm{msgIn}(oid) \cup \mathrm{msgOut}(oid)\\
&\quad \forall\, mst \in \mathrm{MessageStore} : mst(oid) \in \mathrm{Buffer}(\mathrm{msgIn}(oid))\\
&\mathrm{MESSAGE} = \{m \mid \exists\, oid \in \mathrm{UOID} : m \in \mathrm{messages}(oid)\}
\end{aligned}$$

Eine Nachricht $(r, n, v, s, th) \in \mathrm{MESSAGE}$ enthält den Empfänger r, einen Namen n, eine Liste von Werten v, den Sender s und einen Thread th. Die Funktionen sender und receiver liefern den ersten bzw. vorletzten Eintrag des Tupels. Die Menge aller möglichen ein- bzw. ausgehenden Nachrichten für ein Objekt wird mit msgIn bzw. msgOut erhalten. Der MessageStore speichert für jedes Objekt die noch zu verarbeitenden Nachrichten in einem Puffer.

einer Operation entspricht. Alle möglichen Methodenaufrufe von *op* auf einem Objekt *r* durch Objekt *s* und Thread *th* werden durch die Funktion callsOf zusammengestellt. Alle möglichen eingehenden Methodenaufrufe auf dem Objekt *r* erhalten wir mit callsOfO.

Definition SysMod 4.47 (Methodenaufrufe als Nachrichten)

MethodCall

use Message, Control

$$\begin{aligned}
&\mathrm{callsOf} : \mathrm{UOID} \times \mathrm{UOPN} \times \mathrm{UOID} \times \mathrm{UTHREAD} \to \wp(\mathrm{MESSAGE})\\
&\mathrm{callsOfO} : \mathrm{UOID} \to \wp(\mathrm{MESSAGE})
\end{aligned}$$

$$\begin{aligned}
&\forall\, r, s \in \mathrm{UOID}, op \in \mathrm{UOPN}, th \in \mathrm{UTHREAD} :\\
&\quad \mathrm{callsOf}(r, op, s, th) = \{(r, \mathrm{nameOf}(op), pars, s, th) \mid\\
&\qquad r \in \mathrm{oids}(\mathrm{classOf}(op)) \wedge\\
&\qquad pars \in \mathrm{params}(op)\}\\
&\mathrm{callsOfO}(r) = \{msg \mid \exists\, s, op, th : msg \in \mathrm{callsOf}(r, op, s, th)\}
\end{aligned}$$

callsOf ist die Menge der möglichen Methodenaufrufe von Objekt *s* auf Objekt *r* mit Operation *op* und Thread *th*. callsOfO liefert für ein Objekt alle möglichen Methodenaufrufe an dieses Objekt.

Nachrichten für Methodenrückgaben sind ähnlich aufgebaut und in Definition 4.48 formalisiert. Der Aufrufer r erhält von Objekt s zur Operation op einen Rückgabewert (der auch void sein kann). Für einen Thread th erstellt returnsOf alle möglichen Rückgabenachrichten. returnsOfO liefert analog zu callsOfO alle durch Objekt s verschickbaren Rückgabenachrichten.

Definition SysMod 4.48 (Methodenrückgaben als Nachrichten)

MethodReturn

use Message, Control

$$\mathrm{returnsOf} : \mathrm{UOID} \times \mathrm{UOPN} \times \mathrm{UOID} \times \mathrm{UTHREAD} \to \wp(\mathrm{MESSAGE})$$
$$\mathrm{returnsOfO} : \mathrm{UOID} \to \wp(\mathrm{MESSAGE})$$

$$\begin{array}{l} \forall\, r, s \in \mathrm{UOID}, op \in \mathrm{UOPN}, th \in \mathrm{UTHREAD} : \\ \mathrm{returnsOf}(r, op, s, th) = \{(r, \mathrm{nameOf}(op), v, s, th) \mid \\ \quad s \in \mathrm{oids}(\mathrm{classOf}(op)) \wedge \#v = 1 \wedge v[1] \in \mathrm{CAR}(\mathrm{resType}(op))\} \\ \mathrm{returnsOfO}(s) = \{msg \mid \exists\, r, op, th : msg \in \mathrm{returnsOf}(r, op, s, th)\} \end{array}$$

returnsOf ist die Menge aller möglichen Methodenrückgaben von Objekt s zu Objekt r für eine Thread th. returnsOfO liefert für ein Objekt alle möglichen Methodenrückgaben durch dieses Objekt.

Durch die Übermittlung des Threads vom Aufrufer zum aufgerufenen Objekt muss der Aufrufer solange für diesen Thread blockiert bleiben, bis eine Rückantwort (die wiederum den Thread enthält) eingetroffen ist. Dies muss durch weitere Definitionen festgelegt werden, zum Beispiel durch Einführung einer speziellen Menge von Programmzählerwerten *waitPC* und der Festlegung, dass eine Methodenausführung in diesem Zustand erst fortschreitet, wenn ein entsprechender Return im Nachrichtenpuffer eingegangen ist. Wir belassen es bei der intuitiven Darstellung und verweisen auf die Definition von Kontrollfluss-Zustandsmaschinen in [BCGR08]. Beim Austausch von Nachrichten, die Methodenaufrufe und Returns beinhalten, findet also eine synchrone Kommunikation statt. Ist eine asynchrone Kommunikation gewünscht, braucht kein Thread in der Nachricht übermittelt werden. Wir bezeichnen asynchrone Nachrichten als Signale und führen sie in Definition 4.49 ein. Signale sind Nachrichten, deren Inhalt nicht weiter definiert wird. Einzige Annahme ist, dass sie von allen Methodenaufrufen und Returns unterscheidbar sind.

Erhält ein Objekt eine asynchrone Nachricht, muss es selbst aktiv sein (z.B. gemäß Variante 4.43), damit die Nachricht verarbeitet werden kann. Eine Möglichkeit ist, eine spezielle, ständig den Nachrichtenpuffer inspizierende Methode vorzusehen, die für eintreffende Signale eine weitere Verarbeitung veranlasst.

Wir möchten noch einmal betonen, dass jegliche Interaktion zwischen Objekten durch Nachrichtenaustausch (Paradigma des *message passing*) erfolgt. Diese einheitliche Behandlung von synchroner und asynchroner Kommunikation hat den Vorteil einer klar definierten Schnittstelle für Objektinteraktion und erlaubt später eine einfache Komposition von Objektverhalten.

[BCGR08] führt statt des MessageStore einen allgemeineren EventStore für Ereignisse ein,

Definition SysMod 4.49 (Signale als asynchrone Nachrichten)

Signal

$\text{use Message, MethodCall, MethodReturn}$

$\text{USIGNAL} \subseteq \text{MESSAGE}$

$\text{callsOf}(*, *, *, *) \cap \text{USIGNAL} = \varnothing$
$\text{returnsOf}(*, *, *, *) \cap \text{USIGNAL} = \varnothing$

wobei Ereignisse Nachrichten umfassen. Allerdings ist diese Formalisierung nicht in der Lage, ein wichtiges Ereignis, nämlich das Versenden einer Nachricht, direkt abzubilden. Wir schlagen vor, Ereignisse nach Bedarf im Systemmodell einzuführen. Für zwei wichtige Ereignisse, das Senden und Empfangen von Nachrichten, kann die Variante 4.58 aus Abschnitt 4.8 verwendet werden. Andere Ereignisse, z.B. das Registrieren eines Datenzustandswechsels (Änderung eines Attributwertes) sollten auf anderen Theorien des Systemmodells aufsetzen.

Definition 4.50 fasst die Definitionen dieses Abschnitts zusammen. Varianten wurden nicht definiert.

Definition SysMod 4.50 (Nachrichten)

Messages

$\text{extend Message, MessageStore, MethodCall, MethodReturn, Signal}$

Die Theorien *Data*, *Control* und *Messages* zeigen, dass die Integration von Objekten, Threads und Nebenläufigkeit, Methodenaufrufe und Nachrichtenaustausch komplex ist. Alle drei Komponenten besitzen zwar viele orthogonale Aspekte, Interferenzen sind jedoch nicht ausgeschlossen und schwer zu erfassen.

4.7 Objekt- und Systemzustände

Wir sind jetzt in der Lage, den Zustandsraum eines Objekts und des Gesamtsystems in Definition 4.51 zu beschreiben. Dabei hilft uns, dass alle Bestandteile (DataStore, ControlStore und MessageStore) als Abbildungen von UOID auf das entsprechende Zustandselement definiert wurden. Der Zustand eines Objekts ist vollständig beschreibbar durch ein Element der Menge OSTATE. Für den Zustand des gesamten Systems fügen wir die drei oben genannten Abbildungen zusammen. Gültige Zustände setzen voraus, dass die Definitionsbereiche der Abbildungen gleich sind, also jedes Objekt einen Daten-, Kontroll- und Nachrichtenzustand besitzt. Aus einem Gesamtzustand kann mit state der Objektzustand für ein Objekt extrahiert werden. Die Menge aller potentiellen Objektzustände erhalten wir mit statesO. Der Zustandsraum STATE ist kompositional bezüglich der einzelnen Objektzustände, da alle Bestandteile pro Objekt einzeln gespeichert und aufgrund der Objektidentität wieder zu einem Objektzustand zurückgeführt werden können.

Definition SysMod 4.51 (Zustandsraum)

State

extend Data, Control, Messages

$\mathrm{STATE} \subseteq \mathrm{DataStore} \times \mathrm{ControlStore} \times \mathrm{MessageStore}$
$\mathrm{oids} : \mathrm{STATE} \to \wp(\mathrm{UOID})$
$\mathrm{OSTATE} = \mathrm{INSTANCE} \times (\mathrm{UTHREAD} \rightharpoonup \mathrm{Stack}(\mathrm{FRAME})) \times \mathrm{Buffer}(\mathrm{MESSAGE})$
$\mathrm{state} : \mathrm{STATE} \times \mathrm{UOID} \rightharpoonup \mathrm{OSTATE}$
$\mathrm{statesO} : \mathrm{UOID} \to \wp(\mathrm{OSTATE})$
$\mathrm{dsOf} : \mathrm{STATE} \to \mathrm{DataStore}$
$\mathrm{csOf} : \mathrm{STATE} \to \mathrm{ControlStore}$
$\mathrm{msOf} : \mathrm{STATE} \to \mathrm{MessageStore}$

$\mathrm{hasMsg} : \mathrm{STATE} \to \mathrm{UOID} \to \mathrm{MESSAGE} \to \mathrm{bool}$
$\mathrm{running} : \mathrm{Name} \to \mathrm{UOID} \to \mathrm{UOID} \to \mathrm{UTHREAD} \to \mathrm{STATE} \to \mathrm{bool}$
$\mathrm{notExecuting} : \mathrm{Name} \to \mathrm{STATE} \to \mathrm{UOID} \to \mathrm{UOID} \to \mathrm{UTHREAD} \to \mathrm{bool}$

$\mathrm{STATE} = \{(ds, cs, ms) \mid \mathrm{dom}(ds) = \mathrm{dom}(cs) = \mathrm{dom}(ms)\}$
$\forall (ds, cs, ms) \in \mathrm{STATE} :$
$\quad \mathrm{oids}((ds, cs, ms)) = \mathrm{oids}(ds) = \mathrm{dom}(ds)$
$\quad \forall\, oid \in \mathrm{oids}((ds, cs, ms)) : \mathrm{state}((ds, cs, ms), oid) = (ds(oid), cs(oid), ms(oid))$
$\forall\, oid \in \mathrm{UOID} :$
$\quad \mathrm{statesO}(oid) = \{\mathrm{state}(s, oid) \mid s \in \mathrm{STATE} \wedge oid \in \mathrm{oids}(s)\}$

$\forall\, mn \in \mathrm{Name}, oid, oid' \in \mathrm{UOID}, m \in \mathrm{MESSAGE}, th \in \mathrm{UTHREAD}, s \in \mathrm{STATE} :$
$\quad \mathrm{hasMsg}\ \ s\ oid\ m \Rightarrow m \in (\mathrm{msOf}\ s)(oid)$
$\quad \mathrm{running}\ \ mn\ oid\ oid'\ th\ s \Rightarrow (oid, mn, *, *, *, oid') = \mathrm{top}(\mathrm{csOf}(s)(oid)(th))$
$\quad \mathrm{notExecuting}\ \ mn\ s\ oid\ oid'\ th \Rightarrow (oid, mn, *, *, *, oid') \notin \mathrm{csOf}(s)(oid)(th)$

Der globale Zustandsraum STATE besteht aus Datenzuständen, Kontrollzuständen und Nachrichtenpuffern aller Objekte. Der Zustand eines Objekts kann mittels state aus einem Gesamtzustand erhalten werden und besteht aus Attributbelegung, der in ihm aktiven Threads und der anstehenden Nachrichten (OSTATE). Die Funktion statesO definiert die potentiellen Zustände eines Objekts. Die Funktionen dsOf, csOf und msOf sind einfache Projektionen auf die entsprechenden Komponenten des Tupels STATE.

hasMsg, running und notExecuting sind Hilfsdefinitionen, um spätere Aussagen bei der semantischen Abbildung zu vereinfachen. Sie geben an, ob eine bestimmte Nachricht in einem Nachrichtenzustand vorhanden ist, ob eine bestimmte Methode gerade ausgeführt bzw. nicht ausgeführt wird.

Bevor wir im nächsten Abschnitt die notwendigen Signaturen einführen, um das Verhalten der Objekte und des Systems charakterisieren zu können, geben wir die Variante 4.52 an, mit der unter einem Namen Abfragen (Queries) abgelegt werden können. Damit lassen sich Abfragen über dem aktuellen Systemzustand (und einer gegebenen Variablenbelegung, einem Zielobjekt, einem Thread und einer Parameterliste) auswerten. Die Abfragen sind garantiert seiteneffektfrei, da sich der aktuelle Systemzustand nicht ändern kann.

Variante SysMod 4.52 (Abfragen über einem Systemzustand)

Query

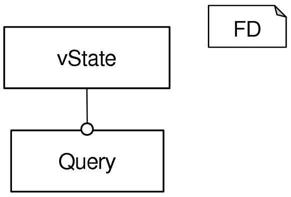
use State

$$\text{query} : \text{Name} \to (\text{STATE} \times \text{VarAssign} \times \text{UOID} \times \text{UTHREAD} \times \text{List(UVAL)} \to \text{UVAL})$$

Dieser Abschnitt enthält nur die Definition 4.51, ihre einzige Variante ist im Feature-Diagramm in Definition 4.53 dokumentiert.

Definition Variabilität 4.53 (vState)

vState



4.8 Objekt- und Systemverhalten

Wir kommen zur Definition von Objektverhalten und dem daraus komponierbaren Systemverhalten. Wie bereits in Abschnitt 4.1 erläutert, werden nur die notwendigen Signaturen skizziert.

Definition 4.54 gibt mit osts an, dass für jedes Objekt eine Zustandsmaschine (gemäß Anhang A.3) existiert. Zustandsraum und Ausgabemenge O sind festgelegt. Die Eingabemenge I umfasst mindestens die eingehenden Nachrichten des Objekts msgIn. Wir haben also noch die Möglichkeit, die Eingabemenge weiter zu präzisieren, indem beispielsweise zusätzliche Eingaben in Form von Scheduling-Anweisungen als Eingaben erlaubt werden (vgl. [BCGR08]). Die Komposition der Zustandsmaschinen wird in Definition 4.55 für zwei Objekte definiert.

Mit sts aus Definition 4.56 wird eine komponierte Zustandsmaschine für eine Gruppe von Objekten erstellt. OSTS ist die Zustandsmaschine für das ganze Universum UOID. Es gibt keine Ausgaben, die Eingaben enthalten keine Nachrichten der Menge MESSAGE. Das Gesamtsystemverhalten wird durch die Zustandsmaschine SYSSTS dargestellt. Dabei werden für die Übergangsfunktion Δ keine Ein- und Ausgaben mehr benötigt, das System ist also „closed-world“. Der Zustandsraum ist der globale Zustandsraum STATE und $s0$ ist eine Teilmenge mit Anfangszuständen. Es ist offengelassen, wie das Gesamtsystem gebildet wird. Hierfür kann,

Definition SysMod 4.54 (Zustandsmaschine für ein Objekt)

OSTS

extend STS, State

$\mathrm{osts} : \mathrm{UOID} \to \mathrm{STS}(S, I, O)$

$$
\begin{array}{l}
\forall\, oid \in \mathrm{UOID} : \mathrm{osts}(oid) \in \mathrm{STS}(S, I, O) \wedge \\
\quad S = \mathrm{states}(oid) \\
\quad O = \mathrm{msgOut}(oid) \\
\quad \mathrm{msgIn}(oid) \subseteq I
\end{array}
$$

Zustandsmaschine für ein Objekt. Sie beruht auf der Definition von Zustandsmaschinen (STS) gemäß Anhang A.3.

Definition SysMod 4.55 (Komposition von Zustandsmaschinen für Objekte)

KompO

extend OSTS

$$
\begin{array}{l}
\forall\, oid_1, oid_2 \in \mathrm{UOID}, \\
\quad (\mathrm{statesO}(oid_1), I_1, O_1, \delta_1, s0_1) = \mathrm{osts}(oid_1), \\
\quad (\mathrm{statesO}(oid_2), I_2, O_2, \delta_2, s0_2) = \mathrm{osts}(oid_2) : \\
\quad \mathrm{osts}(oid_1) \otimes \mathrm{osts}(oid_2) = (S, I, O, \delta, s0), \text{ wobei} \\
\qquad S = \{s \in \mathrm{STATE} \mid \mathrm{state}(s, oid_i) \in \mathrm{statesO}(oid_i), i = 1, 2\} \\
\qquad I = I_1 \cup I_2 \backslash O_1 \cup O_2 \\
\qquad O = O_1 \cup O_2 \backslash I_1 \cup I_2 \\
\qquad s0 = \{st \in \mathrm{STATE} \mid \mathrm{state}(st, oid_i) \in s0_i, i = 1, 2\} \\
\qquad \forall\, s, x : \delta(s, x) = \{(t, y\,|_O) \mid \\
\qquad\quad y \in (I \cup O_1 \cup O_2)^* \wedge \\
\qquad\quad (\mathrm{state}(t, oid_1), y\,|_{O_1}) \in \delta_1(\mathrm{state}(s, oid_1), y\,|_{I_1}) \wedge \\
\qquad\quad (\mathrm{state}(t, oid_2), y\,|_{O_2}) \in \delta_2(\mathrm{state}(s, oid_2), y\,|_{I_2})\}
\end{array}
$$

Ähnlich zur herkömmlichen Komposition von Zustandsmaschinen in Definition A.6 werden Zustandsmaschinen für Objekte komponiert, wobei der komponierte Zustandsraum nicht durch Kreuzprodukt, sondern durch den globalen Zustandsraum STATE gebildet wird. Die Ausgabemengen O_1 und O_2 sind per Konstruktion disjunkt, da jede Nachricht einen eindeutigen Sender besitzt. Der Operator $\otimes$ wird dann auf Mengen erweitert.

wie in [BCGR08] skizziert, eine Variante angegeben werden, die die noch übrigen Eingaben von OSTS mit Scheduling-Anweisungen füllt.

Definition 4.57 definiert auf Basis von SYSSTS erreichbare Zustände und Systemabläufe. Die Hilfsdefinition following drückt aus, dass zwei Zustände aufeinander folgen. created ist eine Abkürzung dafür, dass das angegebene Objekt im letzten Zustandsübergang erzeugt wurde.

Definition SysMod 4.56 (Zustandsmaschine für das System)

sysSTS

$\text{extend OSTS}, \text{KompO}$

$\text{sts} : \wp(\text{UOID}) \to \text{STS}(S, I, O)$
$\text{OSTS} \in \text{STS}(\text{STATE}, I', \varnothing)$
$\text{SYSSTS} = (\text{STATE}, \Delta, s0)$

$\forall\, oids \subseteq \text{UOID} :$
$\quad \text{sts}(oids) = \bigotimes_{oid \in oids} \text{osts}(oid)$
$\text{OSTS} = \text{sts}(\text{UOID}) \wedge I' \cap \text{MESSAGE} = \varnothing$

$\Delta : \text{STATE} \to \wp(\text{STATE}) \wedge s0 \subseteq \text{STATE}$

Definition SysMod 4.57 (Erreichbare Zustände und Abläufe)

Reachable

extend sysSTS

$\text{reachable} \subseteq \text{STATE}$
TRACE

$\forall\, s, t \in \text{STATE} :$
$\quad s \in \text{SYSSTS}\,.s0 \Rightarrow s \in \text{reachable}$
$\quad s \in \text{reachable} \wedge t \in \text{SYSSTS}\,.\Delta(s) \Rightarrow t \in \text{reachable}$

$s_1 \cdot s_2 \cdot s_3 \cdots \in \text{TRACE} \Rightarrow$
$\quad \forall\, i \geq 1 : s_{i+1} \in \text{SYSSTS}\,.\Delta(s_i)$

$\text{following} : \text{TRACE} \to \text{STATE} \to \text{STATE} \to \text{bool}$
$\text{created} : \text{TRACE} \to \text{STATE} \to \text{UOID} \to \text{bool}$

$\forall\, t \in \text{TRACE}, s \in \text{STATE}, s' \in \text{STATE}, oid \in \text{UOID} :$
$\quad \text{following}\, t\; s\; s' \Rightarrow \exists\, i\, j : (i < j \wedge t[i] = s \wedge t[j] = s')$
$\quad \text{created}\, t\; s\; oid \Rightarrow \exists\, i : s = t[i] \wedge oid \in \text{oids}(s) \wedge oid \notin \text{oids}(t[i-1])$

Die Menge der erreichbaren Zustände reachable ist induktiv definiert. TRACE ist die Menge aller (potentiell unendlichen) Systemabläufe. following $t\ s\ s'$ prüft, ob s' dem Zustand s in einem Systemablauf folgt, während created $t\ s\ oid$ prüft, ob ein Objekt mit dem Identifikator *oid* in Zustand s erzeugt wurde.

In Variante 4.58 definieren wir die im vorletzten Abschnitt erwähnten Ereignisse, die beim Senden oder Empfangen einer Nachricht ausgelöst werden können. Es werden ein Universum von Ereignissen UEVENT sowie zwei Ereignisarten ReceivedEvent und SentEvent eingeführt. Der Systemzustand wird per esOf um eine Komponente erweitert, in der Ereignisse ge-

Variante SysMod 4.58 (Ereignisse zum Senden und Empfangen einer Nachricht)

Events

use State

UEVENT
$\mathrm{ReceivedEvent}, \mathrm{SentEvent} : \mathrm{UOID} \rightarrow \mathrm{MESSAGE} \rightarrow \mathrm{UEVENT}$
$\mathrm{esOf} : \mathrm{STATE} \rightarrow (\mathrm{UOID} \rightharpoonup \mathrm{Buffer}(\mathrm{UEVENT}))$
$\mathrm{msgReceivedNow}, \mathrm{msgSentNow} : \mathrm{UOID} \rightarrow \mathrm{STATE} \rightarrow \mathrm{MESSAGE} \rightarrow \mathrm{TRACE} \rightarrow \mathrm{bool}$

$\forall\, t \in \mathrm{TRACE}, s \in \mathrm{STATE}, m \in \mathrm{MESSAGE}, oid \in \mathrm{UOID} :$
$\quad \mathrm{msgReceivedNow}\, oid\ s\ m\ t = (\exists\, i : s = t[i] \wedge \neg\, \mathrm{hasMsg}\, s\ oid\ m \wedge \mathrm{hasMsg}\, t[i+1]\ oid\ m)$
$\quad \mathrm{msgSentNow}\, oid\ s\ m\ t =$
$\qquad (\mathrm{sender}(m) = oid \wedge (\exists\, oid' \in \mathrm{UOID}, : \mathrm{msgReceivedNow}\, oid'\ s\ m\ t))$
$\quad \mathrm{msgReceivedNow}\, oid\ s\ m\ t \Rightarrow \mathrm{ReceivedEvent}\, oid\ m \in \mathrm{esOf}(s)(oid)$
$\quad \mathrm{msgSentNow}\, oid\ s\ m\ t \Rightarrow \mathrm{SentEvent}\, oid\ m \in \mathrm{esOf}(s)(oid)$

speichert werden. Die Hilfsfunktion msgReceivedNow gibt an, ob eine bestimmte Nachricht in einem Zustand empfangen wurde. Analog gibt msgSentNow an, ob eine bestimmte Nachricht gesendet wurde. Die Bedingung im letzten Abschnitt der Variante stellt sicher, dass für jede empfangene oder gesendete Nachricht ein Ereignis im Ereignispuffer abgelegt wird.

Die Definitionen dieses Abschnitts sind in Definition 4.59 zusammengefasst. Es wurde nur die Variante *Events* definiert, die in Definition 4.60 als Variante dokumentiert ist.

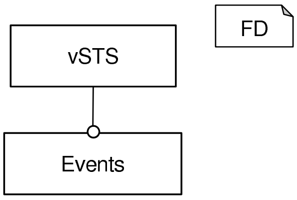
Definition SysMod 4.59 (Verhalten des Systems)

SMSTS

extend OSTS, KompO, sysSTS, Reachable

Definition Variabilität 4.60 (vSTS)

vSTS



[BCGR08] enthält ergänzend eine Präzisierung des Objektverhaltens bei Methodenaufrufen. Einer Methode wird als Implementierung eine so genannte Kontrollfluss-Zustandsmaschine

(CFSTS) zugeordnet. Das Objektverhalten bestimmt sich dann aus der Verarbeitung eingehender Nachrichten, die zur Ausführung der Kontrollfluss-Zustandsmaschine für die in der Nachricht aufgerufenen Methode führen.

Eine weitere wichtige, in [BCGR08] definierte, Eigenschaft des Systemmodells ist die Verträglichkeit von zustandsbasierter Sicht und einer Schnittstellensicht bei Komposition. Ohne die Definition von Strömen und stromverarbeitenden Funktionen ist diese Eigenschaft hier nicht mathematisch formulierbar, weswegen wir auf [BCGR08] verweisen.

4.9 Zusammenfassung

In diesem Kapitel haben wir das Systemmodell als mathematische semantische Domäne eingeführt. Das Systemmodell beschreibt Struktur, Verhalten und Interaktionen objektbasierter Systeme, zum Teil auf sehr feingranularer Ebene. Als Paradigma zur Verhaltensbeschreibung wurden Zustandsmaschinen gewählt. Deren Zustandsraum ist aus Daten-, Kontroll- und Nachrichtenzustand aufgebaut. Der Gesamtzustandsraum ist in Objektzustände zerlegbar.

Das Systemmodell ist in Schichten (vgl. Abb. 4.1) definiert, um eine verständliche Formalisierung trotz des erheblichen Umfangs zu erreichen. Das Systemmodell ist durch einfache mathematische Grundnotationen beschrieben und häufig unterspezifiziert, um Spezialisierungen zu erlauben. Einige Spezialisierungen sind explizit durch Varianten angegeben. Abbildung 4.61 stellt die Variationspunkte im Systemmodell noch einmal gesammelt dar.

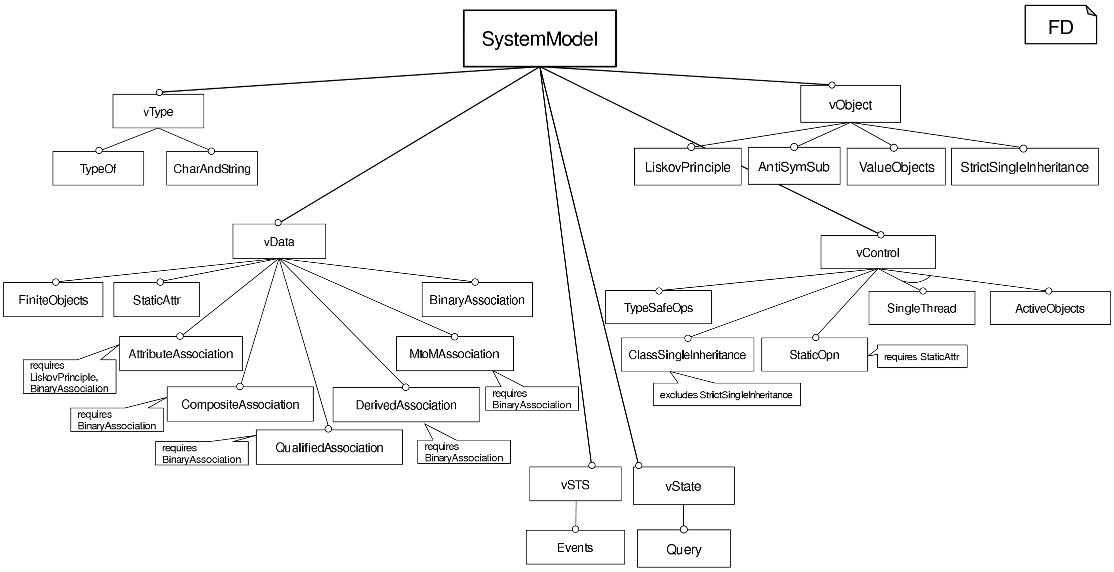

FD
SystemModel
vType
TypeOf
CharAndString
vObject
LiskovPrinciple
AntiSymSub
ValueObjects
StrictSingleInheritance
vData
FiniteObjects
StaticAttr
BinaryAssociation
AttributeAssociation
requires LiskovPrinciple, BinaryAssociation
MtoMAssociation
requires BinaryAssociation
CompositeAssociation
requires BinaryAssociation
DerivedAssociation
requires BinaryAssociation
QualifiedAssociation
requires BinaryAssociation
vControl
TypeSafeOps
SingleThread
ActiveObjects
ClassSingleInheritance
excludes StrictSingleInheritance
StaticOpn
requires StaticAttr
vSTS
Events
vState
Query

Abbildung 4.61: Variationspunkte und Varianten des Systemmodells

Teil II

Werkzeugunterstützung

Kapitel 5

Verwendete Werkzeuge und Übersicht über das Vorgehen

Dieses Kapitel gibt einen Überblick über die für die Syntax- und Semantikdefinition gewählten Werkzeuge und die werkzeugunterstützte Vorgehensweise. Zunächst werden in Abschnitt 5.1 die Anforderungen und Ziele dargestellt, die mit der Werkzeugunterstützung erfüllt bzw. erreicht werden sollen. Anschließend wird die Auswahl der Werkzeuge dargestellt und begründet. Eine kurze Einführung in den Theorembeweiser Isabelle/HOL wird in Abschnitt 5.2 gegeben. Das MontiCore- und das DSLTool-Framework werden in Abschnitt 5.3 diskutiert und Feature-Diagramme mit MontiCore in Abschnitt 5.4 eingeführt. Abschnitt 5.5 beschreibt dann die einzelnen Schritte bei der werkzeugunterstützten Sprachdefinition sowie mögliche alternative Vorgehensweisen. In Abschnitt 5.6 werden verwandte Arbeiten angegeben.

5.1 Auswahl der Werkzeuge

5.1.1 Anforderungen und Ziele

Einem Sprachentwickler soll eine weitgehende Werkzeugunterstützung zur vollständigen Definition von Modellierungssprachen zur Verfügung gestellt werden. Die Anforderungen sind im Einzelnen:

1. Auf Grundlage des in Kapitel 2 beschriebenen Vorgehens zur Definition von Modellierungssprachen soll eine Werkzeugunterstützung entwickelt werden, die eine ähnliche Flexibilität erreicht, wie sie bei einer Definition mit „Papier und Bleistift“ wie im Beispiel in Abschnitt 2.4 möglich ist.

2. Die Semantik einer Modellierungssprache soll auf Basis des Systemmodells definiert werden. Dabei soll auf eine möglichst genaue Kodierung des Systemmodells aus Kapitel 4 Wert gelegt werden.

3. Die Möglichkeit zur Unterspezifikation soll bei der Kodierung der semantischen Domäne und der semantischen Abbildung erhalten bleiben.

4. Es soll ein systematischer Umgang mit Variabilität ermöglicht werden. Der Fokus liegt dabei auf der semantischen Variabilität.

5. Es soll eine Möglichkeit zur Verifikation geschaffen werden, die direkt auf der Semantikdefinition beruht. Über die Verifikation von Eigenschaften konkreter Modelle hinaus sollen auch Aussagen über eine semantische Abbildung oder das Systemmodell im Allgemeinen beweisbar sein.
6. Artefakte wie die Beschreibung der Syntax, die Definition der semantischen Domäne und der semantischen Abbildung sollen in Textform vorliegen, um eine überall lesbare Repräsentation zu erhalten, die auch etablierten Werkzeugen des Software Engineering wie Versionskontrollsystemen zugänglich ist. Durch eine weitergehende Werkzeugunterstützung ist damit eine kollaborative Entwicklung von Semantik denkbar.
7. Modelle der Modellierungssprachen sollen aus denselben Gründen ebenfalls textuell definiert werden. Es entfällt dann unter anderem eine aufwendige Erstellung von Editoren.

5.1.2 Eingesetzte Werkzeuge

Zur Syntaxbeschreibung wurde schon in früheren Kapiteln auf MontiCore-Grammatiken zurückgegriffen, die eine modulare Definition der Syntax erlauben. Für die automatisierte Verarbeitung der Syntaxbeschreibung kann nun das MontiCore-Framework erweitert und eingesetzt werden. Mit der Verwendung von MontiCore werden die Anforderungen 6 und 7 hinsichtlich einer textuellen Definition der Syntax und der Modelle erfüllt. Das MontiCore-Framework wird in der Version 1.2.4 benutzt. Alternative Werkzeuge zur Syntaxdefinition werden in Abschnitt 5.6 besprochen. Basierend auf einer textuellen, ebenfalls mit MontiCore definierten Version von Feature-Diagrammen wird die Variabilität einer Modellierungssprache dokumentiert (Anforderung 4). Mit Hilfe von MontiCore wird das generative Werkzeug *ThyConfig* erstellt, das Feature-Diagramme automatisiert verarbeitet und beispielsweise Konfigurationen auf Konsistenz prüft.

Theorembeweiser sind ein mächtiges Werkzeug zur Kodierung mathematischer Theorien und erlauben eine genaue Umsetzung der in Kapitel 4 angegebenen Definitionen des Systemmodells (Anforderung 2). In dieser Arbeit werden mathematische Theorien im Theorembeweiser Isabelle und seiner Objektlogik HOL (Higher Order Logic) kodiert. Logische Spezifikationen in HOL sind im Allgemeinen zwar nicht ausführbar, bieten aber die Möglichkeit, die Unterspezifikation bei der Kodierung der Semantik zu erhalten (Anforderung 3). Zudem kommen sie einer maschinenlesbaren aber dennoch ähnlich flexiblen Notation wie mit „Papier und Bleistift“ sehr nahe (Anforderung 1 und 6). Die Anforderung 5, also die Möglichkeit zur Verifikation, ist bei der Verwendung von Isabelle/HOL gegeben, da es sich um einen interaktiven Theorembeweiser handelt. Bei entsprechender Kodierung der semantischen Abbildung (tiefe Einbettung, siehe Abschnitt 5.5.2) sind auch Aussagen hierüber formulierbar. Eine Automatisierung der Verifikation ist nur begrenzt möglich. Isabelle/HOL wird in der Version 2008 eingesetzt. Alternative Werkzeuge zur Kodierung der Semantik werden in Abschnitt 5.6 angegeben.

5.2 Isabelle/HOL

Isabelle [Isa10] ist ein generischer Theorembeweiser, in dem konkrete Logiken (so genannte Objektlogiken) implementiert werden können. Isabelle/HOL [NPW02] ist die bekannteste dieser Objektlogiken, die im Wesentlichen als funktionales Programmieren mit Logik verstanden werden kann. In dieser Arbeit kann keine umfassende Einführung in Isabelle/HOL gegeben werden. Es werden nur die für diese Arbeit relevanten Konzepte und Sprachkonstrukte vorgestellt.

5.2.1 Theorien

Isabelle arbeitet mit Dateien, die jeweils eine benannte Theorie beinhalten. Benutzer verwenden meist das auf dem Texteditor *emacs* basierende Frontend *Proof General*, das auch eine Unterstützung zur Eingabe und Darstellung mathematischer Symbole bietet. Die Benutzung erfolgt meist interaktiv, Theorien werden inkrementell eingelesen und verarbeitet. Das bedeutet, dass Einträge definiert werden müssen, bevor sie referenziert und verwendet werden können.

Abbildung 5.1 zeigt den grundsätzlichen Aufbau von Theorien. Jede Theorie hat einen Namen (hier `RobotDSL`) und kann andere Theorien importieren (durch Leerzeichen getrennt). Dabei darf es nicht zu einer zyklischen Abhängigkeit zwischen Theorien kommen. Die Theorie `Main` ist eine Sammlung aller vordefinierten HOL-Theorien und wird immer direkt oder indirekt referenziert. Zwischen den Schlüsselwörtern `begin` und `end` können Konstantendeklarationen, Definitionen, Lemmas und Beweise eingetragen werden. Kommentare werden wie in Zeile 5 gezeigt angegeben.

`RobotDSL`

`Isabelle-Theorie`

```
1  theory RobotDSL
2  imports "Main"
3  begin
4
5  (* Deklarationen, Definitionen und Beweise *)
6
7  end
```

Abbildung 5.1: Grundsätzlicher Aufbau von Isabelle-Theorien

5.2.2 HOL-Konstrukte

Isabelle/HOL ist eine getypte Logik höherer Stufe. In diesem Abschnitt gehen wir auf einige HOL-Konstrukte ein.

- Es stehen Basistypen wie `bool` (Wahrheitswerte) oder `int` (ganze Zahlen) zur Verfügung. Ebenfalls gibt es Typkonstruktoren in Postfixschreibweise, insbesondere `list` (Listen), `set` (Mengen) und `option`. Mit dem Datentyp `option` können partielle Informationen abgebildet werden. Für einen gegebenen Typ `T` besteht der Typ `T option` aus dem ausgezeichneten Element `None` und aus den Elementen `Some t` für jedes Element `t` des Typs `T`. Für einen definierten Wert `v = Some t` erhalten wir mit `the v = t` den Wert zurück.

- Für Listen steht eine große Auswahl an Funktionen und Lemmas zur Verfügung. Darunter befinden sich die aus funktionalen Sprachen bekannten Funktionen höherer Ordnung wie `map` oder `fold`. Die leere Liste wird als `[]` dargestellt. Das Anfügen an eine Liste geschieht mit dem Operator `#`. Listen werden mit `@` konkateniert. `set l` wandelt die Liste `l` in eine Menge desselben Typs um.

- Für Mengen stehen ebenfalls die bekannten Operationen zur Verfügung. `a ∈ A` prüft das Enthaltensein von `a` in `A`, eine Teilmengenbeziehung der Mengen `A` und `B` wird als `A ⊆ B` notiert. Die leere Menge wird als `{}` dargestellt.

- Die Schreibweise für Tupel ist die übliche geklammerte Schreibweise, für ein Paar z.B. `(3,4)`. Der Typ des Tupels wird mit × konstruiert, also z.B. `int × int`.

- Der Funktionstyp ⇒ repräsentiert immer *totale* Funktionen. Partielle Funktionen `f` können syntaktisch durch `f :: A ⇀ B` erzeugt werden. Intern wird eine solche Definition in eine totale Funktion `f :: A ⇒ B option` übersetzt. Für `f` ist `f(a ↦ b)` die Aktualisierung der Funktion an der Stelle `a` auf den Wert `b`.

- Terme werden wie in funktionalen Programmiersprachen gebildet. Beispiele sind Ausdrücke wie
  - `if b then t else s` (`b` hat Typ `bool` und `s`, `t` haben denselben Typ),
  - `λx. x+1` (Lambdaabstraktion für die Funktion, die ihr Argument um eins inkrementiert),
  - `let x = f y in P` (führt lokal in `P` den Namen `x` als Abkürzung für den Ausdruck `f y` ein und hat insgesamt den Typ von `P`).

- Formeln sind Terme des Typs `bool`. Es gibt zwei Konstanten `True` und `False` und die üblichen logischen Konnektiven in fallender Bindungsstärke ¬, ∧, ∨, ⟶ (Implikation) und = (Gleichheit). Die binären Konnektiven sind rechts-assoziativ. Quantoren gibt es in einer beschränkten (z.B. den Allquantor `∀x ∈ A . P`) und einer unbeschränkten Variante (z.B. den Existenzquantor `∃x . P`).

5.2.3 Typaliasse und Datentypen

Abbildung 5.2 zeigt die Definition eines Typalias und eines Datentyps in Isabelle/HOL. In Zeile 5 wird, eingeleitet durch das Schlüsselwort `types`, eine Liste von Zeichen mit `Name` abgekürzt.

Datentypen beginnen mit dem Schlüsselwort `datatype` und ähneln ebenfalls stark den Datentypvereinbarungen aus funktionalen Sprachen. In Zeile 13 wird der Datentyp `Prg` definiert, nach dem Typkonstruktor `Prg` in Zeile 14 folgen eine Reihe von Konstruktorargumenten, hier ein optionaler Bezeichner und eine Liste vom Typ `Cmd`. Dieser Typ ist vorher in Zeile 7 definiert und besitzt vier alternative Konstruktoren, die alle keine Argumente besitzen. Es sind auch wechselseitig rekursive Datentypen definierbar. Diese werden gemeinsam definiert und durch `and` getrennt.

RobotDSL

Isabelle-Theorie

```
theory RobotDSL
imports "Main"
begin

types Name = "char list"

datatype Cmd =
    CmdRIGHT
  | CmdDOWN
  | CmdUP
  | CmdLEFT

datatype Prg =
    Prg "Name option" "Cmd list"

end
```

Abbildung 5.2: Typvereinbarungen in Isabelle/HOL

5.2.4 Konstantendeklarationen

Eine Gruppe von Konstanten wird mit dem Schlüsselwort `consts` durch Angabe des Namens und des Typs deklariert. Mit `defs` werden zuvor eingeführte Konstanten definiert, wobei zusätzlich noch ein Bezeichner zur späteren Referenzierung der Definition angegeben werden muss. Alternativ können beide Schritte mit `constdefs` abgekürzt werden. Dabei kann in vielen Fällen auch die Typangabe aufgrund der Typinferenz sowie der zusätzliche Bezeichner zur späteren Referenzierung (der in diesem Fall automatisch generiert wird) weggelassen werden. Abb. 5.3 zeigt beide Varianten. `prg1` und `prg2` sind jeweils Elemente des Datentyps `Prg` aus Abb. 5.2, wobei Zeichenketten in schräg gestellte Anführungszeichen eingeschlossen werden (Zeile 10). Konstanten können für jeden beliebigen Typ definiert werden, insbesondere auch für Funktionen, deren genaue Definition durch Weglassen des `defs`-Teils unterspezifiziert bleiben kann.

5.2.5 Funktionsdefinitionen und induktive Definitionen

Funktionsdefinitionen können mit dem Schlüsselwort `fun` eingeleitet werden. In Abb. 5.4 ist eine Funktion mit dem Namen `eval` für den Datentyp `Cmd` aus Abb. 5.2 definiert. Die Funktion überführt ein Paar von Ganzzahlen in eine neues Paar, wobei je nach „Kommando" die erste oder zweite Komponente des Tupels verändert wird. Funktionen können alternativ auch mit dem Konstrukt `primrec` gebildet werden. Das Isabelle/HOL-Paket `fun` bietet allerdings eine weitergehende Unterstützung für Terminierungsbeweise oder Vereinfachungsregeln an. Die Funktion `eval` definiert nur die Fälle für zwei der vier alternativen Konstruktoren explizit, trotzdem ist die Funktion total. Das bedeutet, auch für die anderen Fälle ist die Funktion definiert, es wird nur keine Aussage getroffen, wie.

Eine flexible Definition induktiver Prädikate kann mit `inductive` vorgenommen werden. Anders als bei Funktionsdefinitionen werden hier keine Terminierungsbeweise notwendig, eine automatische Funktionsauswertung über mehrere Schritte wie sie für Funktionen mit `fun` zur

RobotDSL

Isabelle-Theorie

```
theory RobotDSL
imports "Main"
begin

(* ... *)

consts prg2 :: "Prg"

defs prg2_def :
 "prg2 == Prg (Some ’’prgName’’) [CmdRIGHT,CmdLEFT]"

constdefs
 "prg1 == Prg None [CmdRIGHT,CmdLEFT]"

end
```

Abbildung 5.3: Konstantendeklaration in Isabelle/HOL

Verfügung stehen, ist allerdings im Allgemeinen nicht möglich. Abb. 5.4 enthält auch ein induktives Prädikat, das ebenfalls die Auswertung des Datentyp `Cmd` beschreibt. Werden Funktionen über wechselseitig rekursive Datentypen aufgebaut, müssen die einzelnen Funktionen ebenfalls zusammen definiert werden, das heißt, durch `and` getrennt notiert werden. Für Funktionsdefinitionen mit `fun` skaliert dieser Ansatz in Isabelle/HOL 2008 nicht, es können dann induktive Definitionen mit den erwähnten Einschränkungen verwendet werden.

RobotDSL

Isabelle-Theorie

```
theory RobotDSL
imports "Main"
begin

(* ... *)

fun eval :: "Cmd ⇒ (int × int) ⇒ (int × int)"
where
  "eval CmdRIGHT (x,y) = (x+1,y)"
| "eval CmdLEFT  (x,y) = (x+-1,y)"

inductive pEval :: "Cmd ⇒ (int × int) ⇒ (int × int) ⇒ bool"
where
  R : "pEval CmdRIGHT (x,y) (x+1,y)"
| L : "pEval CmdLEFT  (x,y) (x+(-1),y)"

end
```

Abbildung 5.4: Funktionsdefinition und induktive Definition in Isabelle/HOL

5.2.6 Beweise

Eine zu beweisende Hypothese wird als Lemma notiert (Schlüsselwort `lemma`, gleichbedeu-

RobotDSL

Isabelle-Theorie

```
theory RobotDSL
imports "Main"
begin

(* ... *)

lemma lr-id : "eval CmdLEFT (eval CmdRIGHT (x,y)) = (x,y)"
apply (simp only: eval.simps)
by simp

end
```

Abbildung 5.5: Beweise in Isabelle/HOL

tend kann auch `theorem` verwendet werden) und dann schrittweise bewiesen. Abb. 5.5 zeigt ein einfaches Lemma. Mit Hilfe von `apply` werden Beweisregeln angewendet, mit `by` wird der letzte Beweisschritt angewendet, der Beweis abgeschlossen und das Lemma als neuer Fakt zur Verfügung gestellt. Obwohl das Lemma von Isabelle mit einen Beweisschritt durch Aufruf des Simplifikators (`simp`) bewiesen werden könnte, ist hier zur Veranschaulichung noch ein Beweisschritt eingefügt, der zunächst die Funktion `eval` auswertet. Der erreichte Beweiszustand wird dann von Isabelle wie folgt ausgegeben:

```
goal (1 subgoal):
 1. (x + 1 + -1, y) = (x, y)
```

Nach Anwendung der ersten Regel existiert also noch ein Zwischenziel, das bewiesen werden muss.

Natürlich ist dies ein sehr einfaches Beweisbeispiel. Konkrete Beweistechniken und umfangreiche Beweise sind nicht Fokus dieser Arbeit. Welche komplexen Beweismöglichkeiten und Taktiken für HOL zur Verfügung stehen, kann im Isabelle/HOL-Tutorial [NPW02] nachgelesen werden.

5.3 Das MontiCore-Framework

In diesem Abschnitt skizzieren wir die Aufgaben des MontiCore-Frameworks [Mon10, KRV07b, KRV08] im Allgemeinen und erklären wie mit seiner Hilfe generative Werkzeuge erstellt werden können. Die Hauptaufgabe des MontiCore-Frameworks besteht darin, MontiCore-Grammatiken einzulesen und Komponenten zur Sprachverarbeitung wie passende Parser und AST-Klassen in Java zu erzeugen. Auf dieser Basis können dann Werkzeuge zur automatisierten Verarbeitung von Modellen entwickelt werden. MontiCore-Grammatiken wurden bereits in Kapitel 2 kurz eingeführt. Ein umfassende Diskussion findet in Kapitel 6 statt, wenn wir die Übersetzung von MontiCore-Grammatiken mit Hilfe des MontiCore-Frameworks nach Isabelle/HOL definieren.

Zur weiteren Verarbeitung von Modellierungssprachen und Modellen wird das *DSLTool-Framework* [Kra10], wie in Abb. 5.6 dargestellt, verwendet. Dies stellt eine Infrastruktur zur Verfügung, die die Erstellung generativer Werkzeuge und damit die automatisierte Verarbeitung

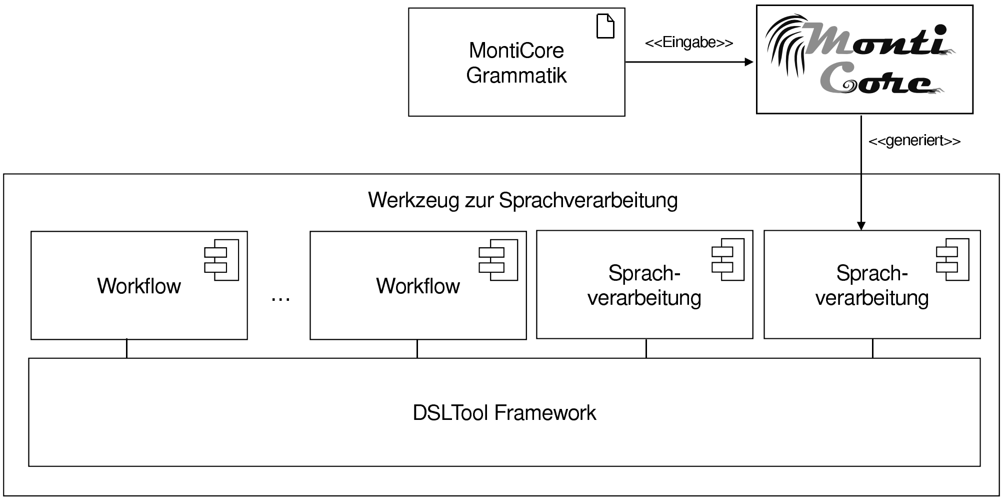


Abbildung 5.6: Erstellung eines generativen Werkzeugs mit MontiCore (angelehnt an [GKR+06])

von Modellen unter Nutzung der durch MontiCore generierten Komponenten unterstützt. Für wiederkehrende technische Fragestellungen werden qualitativ hochwertige Lösungen durch das Framework bereitgestellt. Unter anderem umfasst das DSLTool-Framework folgende Funktionalitäten:

Ablaufsteuerung Nach dem Parsen von Modellen werden weitere Arbeitsschritte durchgeführt. Diese Arbeitsschritte werden als *workflows* implementiert. Ein DSLTool erlaubt die Konfiguration von workflows für unterschiedliche Parametersätze und steuert den Ablauf für konkrete Eingaben.

Traversierung von Datenstrukturen Die Eingangsdaten können konfigurierbar auf Basis einer Erweiterung des Entwurfsmusters Visitor [GHJV95] traversiert werden. Hierbei können zum Beispiel Modelle analysiert oder Code aus ihnen generiert werden.

Modellmanagement Standardmäßig kann ein Paket-Mechanismus ähnlich zum dem in Java für die Verwaltung von Modellen verwendet werden. Über ihren vollqualifizierten Namen sind Modelle dann referenzierbar und Framework-Funktionalität erlaubt das dynamische Nachladen von Modellen bei Bedarf. Damit ein Modell eindeutig wiederzufinden ist, wird automatisch die Einhaltung der Konvention geprüft, dass ein Modell M in Paket A.B im Dateisystem im Verzeichnis A/B unter dem Namen M zu finden ist.

Dateierzeugung Eine häufige Aufgabe für generative Werkzeuge ist die Dateierzeugung. Das DSLTool-Framework bietet einen standardisierten Weg hierfür an. Dabei werden gleiche Dateien nicht erneut erzeugt und überschrieben. Die Dateierzeugung kann für Testfälle ganz unterbunden werden.

Eclipse-Integration MontiCore-Grammatiken lassen sich mit Zusatzinformationen erweitern mit Hilfe derer ein Editor für die integrierte Entwicklungsumgebung Eclipse [Ecl10] generiert werden kann. Konfiguriert werden können unter anderem Syntaxhighlighting, eine Eclipse outline und spezifische Fehlermeldungen.

Attribute MontiCore enthält ein Attributgrammatiksystem [Knu68]. Jede Grammatik kann durch Attribute erweitert werden. Framework-Funktionalität wird für die Aufrufe und Verschaltung der Attributberechnung generiert. Die eigentliche Berechnung eines Attributs wird dann vom Entwickler in Java durch Erweitern einer bestimmten Klasse implementiert.

Der MontiCore-Generator selbst ist mit Hilfe des DSLTool-Framework implementiert. Die Eingangsdaten bzw. Modelle sind MontiCore-Grammatiken, die Parser-Generierung und die Generierung der AST-Klassen sind die wichtigsten workflows. Diese vorkonfigurierten Arbeitsschritte des MontiCore-Generators können leicht um zusätzliche workflows ergänzt werden. Im Zuge dieser Arbeit wurde das MontiCore-Framework um einen weiteren Generator, MontiCore-isabelle, ergänzt. Dieser Generator ist dafür verantwortlich, aus MontiCore-Grammatiken unter anderem einen passenden Isabelle/HOL-Datentyp zu generieren (siehe auch Abschnitte 5.5 und 6.1). Weitere Details zum MontiCore- und DSLTool-Framework können unter anderem in [GKR$^+$06] und [Kra10] nachgelesen werden.

5.4 Feature-Diagramme mit MontiCore

Wir verwenden Feature-Diagramme [CE00], um die Variabilität in einer Sprachdefinition systematisch zu erfassen. Hierfür werden mit MontiCore entwickelte Feature-Diagramme in einer im Vergleich zu [BKRR09] leicht erweiterten Form eingesetzt. Ziel hierbei ist es, Variabilität im Feature-Diagramm zu dokumentieren und für Kontextbedingungen, semantische Abbildungen und das Systemmodell konfigurierte Theorien auf Basis von Konfigurationen der Feature-Diagramme zu generieren.

Textuelle Feature-Diagramme werden anhand eines Beispiels eingeführt. Abb. 5.7 zeigt zwei textuelle Feature-Diagramme. Ein Feature-Diagramm hat einen Namen (Zeile 3), definiert Features und gegebenenfalls Constraints. Wie in anderen mit MontiCore verarbeitbaren Modellen können auch Paketinformationen (Zeile 1) und weitere Informationen zum Modellmanagement standardmäßig angegeben werden. Das Feature-Diagramm `SystemModel` definiert in Zeile 5 ein Feature `SystemModel`, das aus Variationspunkten (`vType`, `VObject`) aufgebaut ist. Die Beziehung zwischen diesen Features ist eine Und-Verknüpfung (`&`). Allerdings sind alle Features optional (`?`). Das Feature `TypeOf` ist ein Blatt im Feature-Baum und bezeichnet damit eine Variante, während `vType` im gleichen Diagramm weiter beschrieben wird und somit einen Variationspunkt darstellt. `*VObject` im ersten Feature-Diagramm referenziert ein weiteres Feature-Diagramm (zweiter Teil der Abbildung), so dass ein modularer Aufbau und eine Wiederverwendung von Feature-Diagrammen möglich ist. Es ist hierbei sicherzustellen, dass die angegebenen Features im Diagramm einen Baum bilden und der Wurzelknoten den Namen des Diagramms trägt, um eine eindeutige Referenzierung von Feature-Bäumen zu gewährleisten. Weitere, im Beispiel nicht benutzte Beziehungen zwischen Features sind Oder-Verknüpfungen (`|`) und Alternativen (`^`), wobei letztere als exklusives Oder zu verstehen sind. Ein Feature

SystemModel

Feature-Diagramm

```
1  package running1;
2
3  featurediagram SystemModel {
4
5    SystemModel = vType? & *VObject; // ...
6
7    vType = TypeOf? & CharAndString?;
8
9    // ...
10
11   constraints {
12     VObject.ValueObjects -> vType.TypeOf;
13   }
14 }
```

VObject

Feature-Diagramm

```
package running1;

featurediagram VObject {

  VObject = LiskovPrinciple? & AntiSymSub? &
            ValueObjects? & SingleInheritance?;
}
```

Abbildung 5.7: Beispiele textueller Feature-Diagramme

kann außerdem mit einem Stereotyp gekennzeichnet werden. Wir werden spezielle Stereotypen für Feature-Diagramm einführen, mit denen die verschiedenen Arten der Variabilität in einer Sprachdefinition unterschieden werden können.

Im ersten Feature-Diagramm ist zusätzlich ein Constraint angegeben (Zeile 11). Constraints werden in Aussagenlogik formuliert und im Wesentlichen dafür verwendet, *requires* oder *excludes* Abhängigkeiten zwischen Features abzubilden. Im Beispiel in Abb. 5.7 bedingt das Feature `ValueObjects` das Feature `TypeOf`. Die Ausdrucksmöglichkeiten zur Angabe von Abhängigkeiten sind neben der Implikation (->) noch eine Oder-Verknüpfung (`|`), eine Und-Verknüpfung (`&`) und die Negation (`!`).

MySM

Konfiguration

```
package running1;

config MySM for running1.SystemModel {

  SystemModel.vType.TypeOf,
  SystemModel.VObject.ValueObjects
}
```

Abbildung 5.8: Textuelle Konfiguration

Die Auswahl konkreter Varianten ist in der Konfigurationen in Abb. 5.8 beispielhaft dargestellt. Eine Konfiguration bezieht sich auf ein Feature-Diagramm (angegeben nach den Schlüsselwort `for`). Es folgt eine einfache Liste der Varianten, die in dieser Konfiguration gewählt wurden. Nicht gezeigt ist, dass auch Konfigurationen andere Konfigurationen referenzieren und damit wiederverwenden können. Syntaktisch wird dies wie in Feature-Diagrammen mit einem vorangestellten Stern markiert. Im Beispiel werden Namen qualifiziert verwendet. Solange Namen eindeutig aufgelöst werden können, können diese auch unqualifiziert verwendet werden. Bei größeren Feature-Diagrammen erleichtert die Qualifikation allerdings zusätzlich das Wiederfinden der Variante im Feature-Diagramm.

Für eine gegebene Konfiguration kann automatisch geprüft werden, ob sie dem angegebenen Feature-Diagramm entspricht. Da eine Konfiguration weitere Konfigurationen referenzieren kann und diese sich wiederum auf dasselbe Feature-Diagramm beziehen können, werden mehrere Konfigurationen für dasselbe Feature-Diagramm automatisch überlagert und die entstehende Gesamtkonfiguration auf Konformität geprüft. Die Konformitätsprüfung berücksichtigt ebenfalls die vorhandenen Constraints im Feature-Diagramm. Diese wird mit Hilfe einer einfachen Evaluation auf Basis des MontiCore-Attributgrammatiksystems durchgeführt.

5.5 Vorgehen zur werkzeugunterstützten Definition einer Modellierungssprache

In diesem Abschnitt beschreiben wir übersichtsartig das Vorgehen bei der werkzeugunterstützten Definition von Syntax und Semantik einer Modellierungssprache. Das Zusammenspiel der einzelnen Artefakte der Sprachdefinition ist in Abb. 5.9 zusammengefasst.

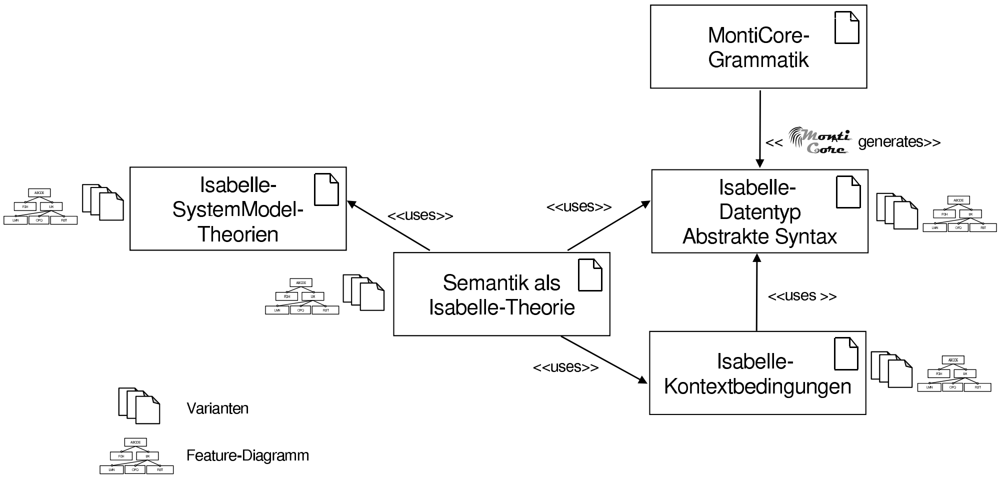

MontiCore-Grammatik
<< Monti Core generates>>
Isabelle-SystemModel-Theorien
<<uses>>
<<uses>>
Isabelle-Datentyp Abstrakte Syntax
Semantik als Isabelle-Theorie
<<uses >>
<<uses>>
Isabelle-Kontextbedingungen
Varianten
Feature-Diagramm

Abbildung 5.9: Zusammenspiel der verschiedenen Artefakte einer werkzeugunterstützten Sprachdefinition

In der vorgeschlagenen Vorgehensweise gilt, die Syntax einer Sprache wird durch MontiCore-Grammatiken festgelegt, abgeleitete abstrakte Syntax, Kontextbedingungen, Systemmodell und semantische Abbildungen stellen mathematische Theorien dar und werden in Isabelle/HOL kodiert. Hierzu wird zunächst das Systemmodell in Isabelle kodiert. Dieser Schritt wird einmalig für alle objektbasierten Modellierungssprachen ausgeführt, da das Systemmodell als einzige gemeinsame semantische Domäne verwendet werden soll. Die Kodierung des Systemmodells in Isabelle wird in Abschnitt 6.4 beschrieben. Eine MontiCore-Grammatik wird in einen Isabelle/HOL-Datentyp übersetzt, der die abstrakte Syntax der Sprache repräsentiert (vgl. Abschnitt 6.1). Hierfür wird der MontiCore-Generator um einen zusätzlichen workflow erweitert, der diese Aufgabe übernimmt und beim Aufruf des MontiCore-Generators über den entsprechenden Parameter `isabelle` gestartet werden kann. Diesen Generator bezeichnen wir im Folgenden mit MontiCore-isabelle.

Für eine Modellierungssprache sind folgende Schritte zu erledigen:

1. Entwicklung einer MontiCore-Grammatik zur Festlegung der Syntax.
2. Anwendung von MontiCore-isabelle zur automatischen Erzeugung passender Datentypen in Isabelle/HOL für die abstrakte Syntax.
3. Entwicklung von (Intra-)Kontextbedingungen in Isabelle/HOL, die für die Modelle gelten sollen.
4. Entwicklung der semantischen Abbildung in Isabelle/HOL, die das kodierte Systemmodell und die generierte abstrakte Syntax in Beziehung setzt.

Begleitend wird die Variabilität der Sprache definiert. Dies beinhaltet zum einen die Dokumentation einer Variante im Feature-Diagramm, zum anderen die Bereitstellung der Variante in Form eines bestimmten Artefakts (z.B. einer Isabelle-Theorie). Die Werkzeugunterstützung ist dabei hauptsächlich auf die Behandlung von Varianten von Kontextbedingungen, semantischen Abbildungen und des Systemmodells ausgelegt.

1. Präsentationsoptionen werden im Feature-Diagramm geeignet dokumentiert. Die jeweiligen Varianten gehören zur Definition der konkreten Syntax und sind im Fall von MontiCore direkt in der Grammatik enthalten.
2. Stereotypen müssen geeignet beschrieben und zulässige Stereotypenmengen im Feature-Diagramm dokumentiert werden. Ihre Einhaltung kann beispielsweise durch syntaktische Überprüfung nach dem Parsen oder ebenfalls als zusätzliche Kontextbedingung geprüft werden.
3. Sprachparameter werden im Feature-Diagramm dokumentiert. Ihre konkrete Konfiguration wird allerdings in MontiCore bereits durch so genannte Sprachdateien geregelt. Um nicht einen zusätzlichen Mechanismus zu schaffen, werden zugehörige Theorien zur Konfiguration der Sprachparameter manuell konsistent gehalten.
4. Strukturelle Veränderungen der abstrakten Syntax und syntaktische Modelltransformationen stehen nicht im Fokus der Arbeit, können aber beispielsweise mit dem Transformationsframework MontiCore-TF [Mah06, The06, Nun09] umgesetzt und im Feature-Diagramm dokumentiert werden. Wir setzen Varianten von Abkürzungen und Sprachein-

schränkungen stets durch Varianten von Kontextbedingungen um und nehmen keine strukturelle Veränderung der abstrakten Syntax vor. Syntaktische Spracherweiterungen werden also nicht betrachtet.

5. Varianten von Kontextbedingungen (für Abkürzungen und Spracheinschränkungen), semantischen Abbildungen und des Systemmodells werden in Form von Isabelle-Theorien bereitgestellt. Die Varianten werden im Feature-Diagramm dokumentiert. Für eine valide Konfiguration der Sprache (konform zu den entsprechenden Feature-Diagrammen) bietet die Werkzeugkette jeweils eine automatisierte Erstellung der Gesamttheorie als Gesamtheit aus Basistheorie und allen gewählten Varianten an.

MLang

Feature-Diagramm

```
<<lang>> featurediagram L {

 L = LPresentation? &
     LSyntax? &
     LSemantics?;

 LPresentation = <<po>> LPresentationOptions? &
                 <<abb>> LAbbreviations;

 LSyntax = <<lparam>> LParameters? &
           <<stereos>> LStereotypes? &
           <<constr>> LConstraints?;

 LSemantics = <<sm>> LSemMapping? &
              <<sd>> ⋆SystemModel?;
}
```

Abbildung 5.10: Feature-Diagramm als Übersicht über Variabilität der Sprache L

Als Hilfestellung und als Vorlage zur Dokumentation der Variabilität in einer Modellierungssprache (hier `L`) kann das generische Feature-Diagramm in Abbildung 5.10 verwendet werden. Es handelt sich um eine textuelle Fassung der graphischen Vorlage aus Abschnitt 3.2.1. Der Stereotyp `<<lang>>` zeigt an, dass mit diesem Feature-Diagramm die Variabilität in einer Sprache modelliert wird. Nicht gezeigt werden konkrete Varianten, da es sich um eine generische Vorlage handelt. Variationspunkte sind typischerweise optional, das stellt den Standardfall dar. Die verwendeten Stereotypen dokumentieren, auf welche Form der Variabilität sich der Variationspunkt bezieht:

po (*presentation options*) Varianten der Präsentationsoptionen.

abb (*abbreviations*) Varianten der Abkürzungen, die als Kontextbedingungen umgesetzt werden.

lparam (*language parameters*) Mögliche Sprachparameter.

stereos (*stereotypes*) Mengen von Stereotypen.

constr (*constraints*) Varianten der Spracheinschränkungen, die als Kontextbedingungen angegeben werden.

sm (*semantic mapping*) Varianten der semantischen Abbildung.

sd (*semantic domain*) Varianten der semantischen Domäne.

Die Angabe der Stereotypen im Feature-Diagramm ist aus technischen Gründen notwendig, um eine eindeutige Zuordnung eines Feature-Diagramm-Teilbaums zu einer Art der Sprachvariabilität zu erreichen. Wie bereits erwähnt werden für Konfigurationen insbesondere die Variationspunkte, die mit `sm`, `sd` bzw. `abb` oder `constr` markiert sind, berücksichtigt, da hierfür integrierte Isabelle-Theorien generiert werden können. Für alle anderen Teile einer Konfiguration wird nur die Konformität zu den Feature-Diagrammen geprüft.

In Abb. 5.9 ist nicht dargestellt, dass sich Kontextbedingungen prinzipiell auch auf Teile anderer, eingebetteter oder gleichzeitig verwendeter Modellierungssprachen beziehen können (Inter-Kontextbedingungen). Hier sind also zwei Fälle zu berücksichtigen:

- Kontextbedingungen eingebetteter Sprachen können wiederverwendet (importiert) werden. Zusätzliche Kontextbedingungen, die sich durch Einbettung einer bestimmten Sprache ergeben können, werden durch eine von der eingebetteten Sprache abhängigen Variante definiert.

- Kontextbedingungen zwischen Modellen eigenständiger Modellierungssprachen werden über einer konzeptuell integrierten Sprache angegeben. Hierfür muss eine weitere Theorie erstellt werden, die beide Sprachen (beispielsweise in einer gemeinsamen Datentypdefinition) integriert.

Nach der Definition der Sprache und ihrer Varianten kann die Sprache für einen Verwendungszweck konfiguriert werden. Hierzu werden Konfigurationen der Feature-Diagramme erstellt. Das Werkzeug *ThyConfig* prüft diese Konfigurationen und erzeugt Isabelle-Theorien, die eine integrierte Sprachdefinition bilden.

Auf Basis der nun vorhandenen formalen, maschinenlesbaren Beschreibung der Modellierungssprache in Isabelle/HOL können Eigenschaften der Syntax (auch ihrer Kontextbedingungen), der semantischen Abbildung, des Systemmodells und ihre Wechselwirkungen für eine Konfiguration untersucht werden.

Die Möglichkeit zur Verifikation von Eigenschaften konkreter Modelle ist noch nicht direkt möglich. Daher wird das Vorgehen erweitert. In Abb. 5.11 ist gezeigt, dass ein zweiter Generator automatisch von MontiCore-isabelle *generiert* wird. Nach einer Parametrisierung dieses Generators auf die gewählte Einbettung, können konkrete Modelle in ihre abstrakte Repräsentation in Isabelle/HOL übersetzt werden. Damit ist auch Verifikation auf konkreten Modellen durch Nachweis von Eigenschaften der Modelle in Isabelle mit diesem Ansatz möglich.

5.5.1 Organisation der Sprachdefinition

Zur Organisation der verschiedenen Artefakte einer Sprachdefinition übernehmen wir die Projektstruktur von MontiCore-Projekten (vgl. [Kra10]). Ein Projekt besteht dabei aus mehreren

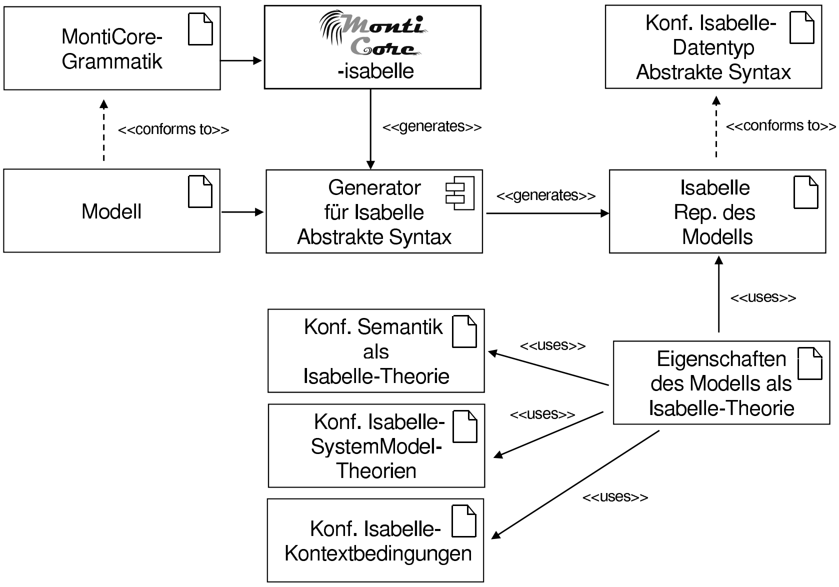

MontiCore-Grammatik
MontiCore
-isabelle
Konf. Isabelle-Datentyp Abstrakte Syntax
<<conforms to>>
<<generates>>
<<conforms to>>
Modell
Generator für Isabelle Abstrakte Syntax
<<generates>>
Isabelle Rep. des Modells
<<uses>>
Konf. Semantik als Isabelle-Theorie
<<uses>>
Eigenschaften des Modells als Isabelle-Theorie
Konf. Isabelle-SystemModel-Theorien
<<uses>>
<<uses>>
Konf. Isabelle-Kontextbedingungen

Abbildung 5.11: Erweiterte Ansicht mit Verwendung eines konkreten Modells

Ordnern, in denen verschiedene Arten von Dateien gespeichert werden. Im Ordner `def` findet sich neben der MontiCore-Grammatik auch das Feature-Diagramm für die gesamte Sprache. In weiteren Unterordnern werden Sprachbestandteile wie Kontextbedingungen oder die semantische Abbildung, mögliche Varianten und die zugehörigen Feature-Diagramme gespeichert. Eine genaue Struktur ist nicht vorgeschrieben, jedoch müssen in Feature-Diagrammen referenzierte Artefakte (referenzierte Feature-Diagramme oder Varianten) eindeutig über einen relativen Pfad auffindbar sein.

In einem separaten Projekt werden die Theorie und das Feature-Diagramm des Systemmodells allen Sprachdefinitionen zur Verfügung gestellt. Jede auf dem Systemmodell basierende Sprachdefinition enthält in ihrem Feature-Diagramm eine Referenz auf das Feature-Diagramm des Systemmodells.

Sprachkonfigurationen gehören logisch nicht zur Definition der Modellierungssprache, sondern sollten in einem separaten Projekt, das die Sprache einsetzt, in einer entsprechenden Ordnerstruktur organisiert werden. Dabei können möglicherweise vorhandene Standardkonfigurationen in der aktuell erstellten Konfiguration referenziert und somit wiederverwendet werden.

Generierte Dateien werden in MontiCore-Projekten im Ordner `gen` abgelegt. Hier finden sich beispielsweise der generierte Parser und die AST-Klassen. Zusätzlich werden in diesem Ordner jetzt auch die generierte abstrakte Syntax als Isabelle-Datentyp (erzeugt durch den workflow `isabelle`) und die ebenfalls generierten Isabelle-Theorien für eine konfigurierte semantische Abbildung oder das integrierte Systemmodell gespeichert (erzeugt durch das Werkzeug *ThyConfig*).

5.5.2 Alternative Vorgehensweisen

Unser Ansatz sieht vor, das Systemmodell als semantische Domäne für objektbasierte Sprachen zu verwenden. Für nicht-objektbasierte oder sehr einfache Sprachen kann gegebenenfalls auf andere semantische Domänen zurückgegriffen werden. Die MontiCore-Grammatik in Abb. 5.12 zeigt beispielsweise die Syntaxdefinition für eine einfache DSL zur Robotersteuerung, angelehnt an die Fallstudie in [WGM08]. Zu dieser Definition passt die Isabelle-Datentypdefinition in Abb. 5.2 auf Seite 87. Die Funktion `eval` aus Abb. 5.4 auf Seite 88 stellt die Semantik für die Sprache dar (Integerpaare bilden die semantische Domäne). Wir erhalten also für die DSL zur Robotersteuerung ebenfalls eine vollständige, werkzeugunterstützte Definition der Sprache, die ohne die Verwendung des Systemmodells auskommt. Generell können beliebige andere, in Isabelle kodierte semantische Domänen gleichermaßen verwendet werden.

RobotDSL

MontiCore-Grammatik

```
package robot;

grammar RobotDSL {

 Prg = Name? "{" Cmd* "}";

 enum Cmd = "left" | "right" | "up" | "down";

}
```

Abbildung 5.12: MontiCore-Grammatik für die RobotDSL

Eine weitere Abwandlung des Ansatzes besteht darin, ein Metamodellierungswerkzeug wie EMF [BSM^{+}03] oder GME [LMB^{+}01] an Stelle von MontiCore-Grammatiken einzusetzen. Der Nachteil ist, dass dann der Generator zur Übersetzung nach Isabelle/HOL nicht verwendet und erneut implementiert werden müsste. Häufig muss auch die konkrete Syntax für die zu entwickelnde Sprache separat definiert werden. Ein solcher Ansatz stellt auch nur eine technologische Variante dar, da sich typische Konzepte der Metamodellierung auch mit MontiCore-Grammatiken nachbilden lassen [KRV07b, Kra10].

Anstatt einer direkten Definition der Semantik basierend auf dem Systemmodell kann als eine weitere Möglichkeit auch eine syntaktische Transformation einer neuen Sprache auf eine bekannte, schon ins Systemmodell abgebildete Sprache, z.B. mittels MontiCore-TF, implementiert werden. Der Vorteil ist die rein syntaktische Übersetzung des Modells in ein anderes Modell. Nachteilig ist die mehrstufige Übersetzung in das Systemmodell, die die Verifikation auf Systemmodellebene erschweren kann.

Flache und tiefe Einbettung der Syntax

Dieser Abschnitt diskutiert den Unterschied zwischen flacher und tiefer Einbettung [WN04] in Isabelle/HOL. In Abb. 5.9 wurde der Ansatz vorgestellt, der eine tiefe Einbettung der Sprache in Isabelle/HOL vornimmt. Das bedeutet, dass die Syntax in HOL explizit repräsentiert wird und die semantische Abbildung vollständig in Isabelle erfolgen kann. Die explizite Repräsentation sowohl der Syntax als auch der semantischen Abbildung hat den Vorteil, dass allgemeine Eigenschaften der Syntax oder der semantischen Abbildung unabhängig von konkreten Modellen in Isabelle untersucht werden können.

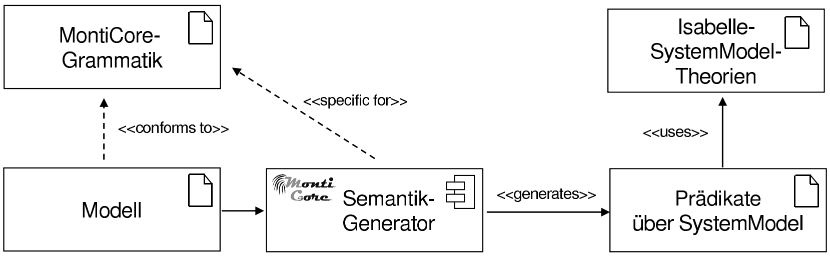


Abbildung 5.13: Alternativer Ansatz: Flache Einbettung

Im Gegensatz dazu zeigt Abb. 5.13 das Vorgehen bei einer flachen Einbettung. Für ein konkretes Modell, das zu einer bestimmten Sprache gehört, wird mit Hilfe eines für die Sprache spezifischen „Generators“ die Semantik eines Modells direkt generiert und beispielsweise als Prädikate über dem Systemmodell kodiert. Es entfällt der Schritt, ein konkretes Modell explizit syntaktisch in Isabelle zu repräsentieren. Der Ansatz bietet den Vorteil bei Beweisen, nicht über den Umweg der abstrakten Syntax und semantischen Abbildung zu den gewünschten Eigenschaften gelangen zu müssen, sondern direkt die generierten Eigenschaften verwenden zu können. Allerdings hat der Ansatz der flachen Einbettung auch gravierende Nachteile. Die semantische Abbildung muss in einem Generator kodiert werden. MontiCore-Generatoren werden in Java programmiert, so dass die semantische Abbildung potentiell schwer verständlich über Java-Klassen verteilt wird. Alternativ müsste eine spezielle Syntax zur einfachen Beschreibung semantischer Abbildungen entworfen werden. Isabelle/HOL hingegen besitzt bereits eine ausgereifte mathematische Syntax, die direkt verwendet werden kann. Wichtiger noch ist, dass bei einer flachen Einbettung nur Beweise auf Basis des Systemmodells möglich sind. Untersuchungen der semantischen Abbildung oder syntaktischer Operationen auf Modellen sind prinzipiell nicht möglich, da diese gar nicht in Isabelle verfügbar sind.

In unserem Ansatz wird ausschließlich eine tiefe Einbettung verfolgt. Zukünftig sind aber auch Mischformen vorstellbar. Insbesondere können hilfreiche Definitionen und Beweise direkt aus konkreten Modellen automatisch generiert werden. Beispielsweise könnten beim Nachweis der Wohlgeformtheit Informationen genutzt werden, die beim Aufbau einer Symboltabelle des Modells bereits verfügbar sind. Variantenbildung ist auch im Fall einer flachen Einbettung problemlos möglich.

5.6 Verwandte Arbeiten

Unsere Werkzeugunterstützung stellt eine so genannte *language workbench* [Fow05] dar, die neben der Syntax- auch die Semantikdefinition und den Umgang mit Sprachvarianten erlaubt. Zur Definition der Semantik wird der Theorembeweiser Isabelle/HOL verwendet. Viele Arbeiten haben gezeigt, dass Theorembeweiser genutzt werden können, die Semantik einer Sprache zu definieren und zu Verifikationszwecken zu verwenden [Ter95, Nip98, Ohe01, Ber07]. In [BW06, BW09] wird OCL als flache Einbettung in Isabelle/HOL realisiert. Es wird unter anderem auch eine komplexe Struktur zur Darstellung von erweiterbaren Klassenhierarchien entwickelt. Die im OCL-Standard vorgesehene dreiwertige Logik wird berücksichtigt.

Der Synthesizer-Generator [RT84] (Cornell University) kann aus einer BNF-Syntax einer Sprache und Attributgrammatik-Definitionen einen syntaxgetriebenen Editor gewinnen. Die Attributgrammatik [Knu68] kann hierbei auch genutzt werden, um die Semantik der Sprache z.B. durch Auswertungsvorschriften für Ausdrücke oder Generierung von ausführbaren Anweisungen zu beschreiben [KJ92]. Ein ganz ähnlicher Ansatz wird mit dem Attributgrammatik-Werkzeug (Aspect)Lisa [RMHV06] verfolgt. Das PSG Programmsystem [BS86] erlaubt die Generierung von Editoren und Interpretern basierend auf einer Spezifikation der Syntax, Kontextbedingungen und (dynamischen) Semantik. Die Semantik wird mit Hilfe einer funktionalen Sprache in einem denotationalen Stil beschrieben. Im IPSEN-Ansatz [KS97] können sowohl textuelle als auch graphische Sprachen erstellt werden. Mit Hilfe des theoretisch fundierten Graphersetzungssystems PROGRES [Sch91] kann auch eine Semantikdefinition z.B. durch Übersetzung in einen weiteren Formalismus erfolgen. Das CENTAUR System [BCD$^+$89] ist eine Werkzeugsammlung in der die operationale Semantik einer Sprache in Typol angegeben und dann nach Prolog übersetzt wird. Mit Hilfe des Prolog-Interpreters können Anfragen ausgewertet werden, um beispielsweise einen Ergebniswert eines Ausdrucks zu berechnen. In [Ter95] wird eine alternative Übersetzung von Typol in den Theorembeweiser Coq diskutiert. Ein Ansatz zur algebraischen Spezifikation wird in ASF+SDF [vdBvDH$^+$01] verfolgt. Die abstrakte Syntax wird als Modul mit einer Signatur und einer Menge von Gleichungen in ASF (Algebraic Specification Formalism) beschrieben. SDF (Syntax Definition Formalism) dient der Spezifikation der konkreten Syntax und kann auf ASF-Spezifikationen abgebildet werden.

Semantic anchoring ist ein neuerer Ansatz, um domänenspezifische Sprachen semantisch zu fundieren [CSAJ05]. Die abstrakte Syntax einer zu behandelnden Sprache wird in die abstrakte Syntax einer bereits bekannten Sprache mit definierter Semantik transformiert. Diese Sprache wird als semantische Einheit (semantic unit) bezeichnet und wird als Abstract State Machine (ASM) in AsmL [Asm10] kodiert. Die syntaktische Transformation erfolgt mit dem Transformationswerkzeug GReAT [GRe10]. Zur Metamodellierung wird das Generic Modeling Environment GME [GME10, LMB$^+$01] verwendet. Ein wie in [CSAJ05] beschriebener Ansatz ist prinzipiell auf andere Frameworks übertragbar, die eine Möglichkeit zur Codegenerierung oder Modelltransformation vorsehen. Explizit diskutiert wird diese Möglichkeit im Kontext anderer aktueller Ansätze wie dem EMF-basierten OpenArchitectureWare (OAW, enthält unter anderem auch xtext) [Ope10] oder MetaEdit+ [KT08] allerdings nicht. In [Wac07] wird eine Implementierung von operationaler Semantik mit Hilfe der OMG-Standards MOF [OMG06a] und QVT [OMG08] vorgeschlagen. Die Transitionsrelationen der operationalen Semantik werden als QVT-Transformationen beschrieben.

Ein vollständig integrierter und werkzeugunterstützter Ansatz, um eine Modellierungssprache formal zu definieren, ist in [KM08] gezeigt. Syntax, Typüberprüfung und operationale Semantik werden in Alloy [Jac02] definiert. Ein Vorteil ist die integrierte Entwicklung in nur einem Werkzeug und einem Formalismus. Alloy verwendet eine Logik erster Stufe und basiert auf der *small scope hypothesis*, die aussagt, dass normalerweise ein begrenzter Suchraum (zum Beispiel im

Zustandsraum) ausreicht, um Gegenbeispiele zu Behauptungen zu finden. Alloy bietet so zwar eine bessere Automatisierung als Isabelle, ist aber auch nicht so allgemein.

Die bisher erwähnten verwandten Arbeiten berücksichtigen keine Variabilität in der Sprachdefinition. In Kapitel 3 haben wir bereits einige Ansätze besprochen für die auch eine Werkzeugunterstützung zur Verfügung steht. Template semantics und darauf basierende Codegeneratoren [NAD03, PADS08] sind geeignet, Variabilität in zustandsbasierten Berechnungsmodelle zu beschreiben. Templatable metamodels [CMTG07b] sind ebenso nur geeignet, um Variabilität in Verhaltensbeschreibungen zu kodieren, genauso wie [CJ05]. In [SASX05] wird eine auf temporaler Logik basierende variable Semantik für Echtzeit-Statecharts angegeben. Die Semantik ist im Theorembeweiser PVS kodiert und mit Varianten parametrierbar. Die Werkzeugunterstützung ist spezifisch für die Statechart-Semantik, so dass auch spezielle Taktiken im Theorembeweiser entwickelt wurden, die eine Beweisführung einfacher machen. Der Ansatz ist spezifisch für Statecharts und definiert kein allgemeines Vorgehen zur Variantenbildung von Modellierungssprachen. Uns ist kein vergleichbares Framework bekannt, das Werkzeugunterstützung für die Definition einer Modellierungssprache anbietet und dabei die Variabilität der Sprache explizit berücksichtigt.

Kapitel 6

Werkzeugunterstützte Definition einer Modellierungssprache

Die Vorgehensweise zur werkzeugunterstützen Definition einer Modellierungssprache wurde in Abschnitt 5.5 eingeführt. In diesem Kapitel werden nun die einzelnen Bestandteile dieser Vorgehensweise im Detail erläutert. Hierzu werden kurz die Vorgehensweise und die hierfür zu schaffenden Voraussetzungen wiederholt, die den Aufbau des Kapitels motivieren.

- Für die Entwicklung einer MontiCore-Grammatik zur Festlegung der Syntax kann die bereits existierende Infrastruktur von MontiCore benutzt werden. Wir gehen davon aus, dass eine Grammatik der Sprache erstellt wurde. Eine Anleitung zum Entwurf von MontiCore-Grammatiken ist beispielsweise in [Kra10, KKP$^+$09] zu finden.
- MontiCore-isabelle wird verwendet, um passende Datentypen in Isabelle/HOL für die abstrakte Syntax der Sprache automatisch zu erzeugen. Die systematische Übersetzung von MontiCore-Grammatikonstrukten nach Isabelle/HOL wird in Abschnitt 6.1 beschrieben.
- Die automatisierte Übersetzung konkreter Modelle in eine Instanz des generierten Datentyps ist in Abschnitt 6.2 erläutert. Es können für beliebige Modellierungssprachen Generatoren automatisch erzeugt werden. Die Übersetzung erlaubt die Arbeit auf konkreten Modellen in Isabelle/HOL.
- Auf Basis des generierten Datentyps zeigen wir in Abschnitt 6.3 wie Kontextbedingungen als Isabelle-Theorie kodiert werden.
- Die Kodierung des Systemmodells und die dabei getroffenen Entwurfsentscheidungen werden in Abschnitt 6.4 erläutert. Da die Kodierung von allen Systemmodell-basierten Modellierungssprachen wiederverwendet wird, ist die Kodierung nicht eigenständiger Teil des Vorgehens und muss nur einmalig erfolgen.
- Hinweise zur Entwicklung einer semantischen Abbildung in Isabelle/HOL, die das kodierte Systemmodell und die generierte abstrakte Syntax in Beziehung setzt, finden sich in Abschnitt 6.5.
- Abschnitt 6.6 behandelt die werkzeugunterstützte Definition und Konfiguration der Variabilität. Neben der Konformitätsprüfung einer Konfiguration ist für Varianten, die als Isabelle-Theorien kodiert werden, die Generierung einer integrierten Theorie durch das Werkzeug ThyConfig möglich.

6.1 Übersetzung der MontiCore-Grammatik nach Isabelle/HOL

MontiCore-Grammatiken können automatisiert in passende Isabelle/HOL-Datentypen übersetzt werden, die die abstrakte Syntax repräsentieren. Wir führen die unterschiedlichen MontiCore-Grammatikkonstrukte und deren Übersetzung schrittweise ein. Wir orientieren uns beim Aufbau dieses Abschnitts und bei der Übersetzung der Konstrukte an [Kra10], wo die einzelnen Elemente der konkreten und abgeleiteten abstrakten Syntax von MontiCore-Grammatiken eingeführt werden. Die systematische Übersetzung wird anhand von Beispielen in konkreter Syntax erklärt. Alternativ wäre auch eine Übersetzung der abstrakten Syntax in Form von Klassendiagrammen (wie sie in [Kra10] beschrieben sind) in Isabelle/HOL möglich. Es wurde die direkte Abbildung der Konstrukte gegenüber der zweistufigen Übersetzung bevorzugt. Es besteht somit auch keine Verwechselungsgefahr mit den später behandelten UML/P Klassendiagrammen.

6.1.1 Lexikalische Syntax

MontiCore verwendet Antlr-basierte Lexer [Ant10], um Tokenklassen durch reguläre Ausdrücke zu definieren. In MontiCore-Grammatiken werden Regeln für den Lexer mit dem Schlüsselwort `token` eingeleitet.

LexProd

MontiCore-Grammatik

```
token CARDINALITY = ('0'..'9')+ | '*' : x -> int :
    {if (x.getText().equals("*")) return -1;
     else return Integer.parseInt(x.getText()); };

token FOO = ('2''b'('0'..'9')+) : x -> lex.Foo :
    { lex.Foo f = new lex.Foo(); f.setText(x.getText());
      return f;
    };
```

LexProdAS

Isabelle-Theorie (gen)

```
types FOO = "char list"

types CARDINALITY = "int"
```

Abbildung 6.1: Lexikalische Syntax

In Abb. 6.1 sehen wir Beispiele für Lexer-Regeln. Die Regel `CARDINALITY` erkennt ganze Zahlen und das Symbol `*`, und ist geeignet, um Kardinalitäten zu spezifizieren. Der Rückgabewert für diese Regel ist vom Typ `int`. Eine Übersetzungsfunktion ist in Java in geschweiften Klammern angegeben. Dabei wird das Symbol $\star$ auf den Wert -1 abgebildet. Eingebaute Rückgabetypen sind neben `int` auch `float` und `String`. Hierfür wird eine Standardkonvertierung vorgenommen, sodass die Übersetzungsfunktion auch entfallen kann. Es können auch beliebige andere Datentypen verwendet werden, wie die Regel `FOO` zeigt. In geschweiften Klammern

CLexTypeConverter

MC-isabelle/Java

```
public class CLexTypeConverter implements LexTypeConverter{

  @Override
  public String translate(STType lextype) {
    String res = LexType.Default;
    if (lextype.getLextype().equals("Foo")) {
      res = LexType.CharList;
    }
    // ...
  }
}
```

Abbildung 6.2: Konvertierung von lexikalischen Produktionen

muss dann wiederum eine Übersetzungsfunktion angegeben werden. In diesem Fall muss eine Übersetzung des Tokens in ein Objekt des Java-Typs `Foo` implementiert werden. Ist kein expliziter Rückgabetyp angegeben werden Token in Objekte vom Typ String übersetzt.

Im zweiten Teil der Abbildung 6.1 wird das Ergebnis der Übersetzung nach Isabelle/HOL gezeigt. Standardmäßig werden Lexer-Regeln in Isabelle zu einer Zeichensequenz (`char list`). Eingebaute Datentypen wie `int` werden in einen passenden HOL-Datentyp übersetzt (`int` nach `int` oder `String` nach `char list`). Das Standardverhalten führt also zu einer Typdefinition `CARDINALITY = "int"`. Dieses Verhalten bei der Übersetzung kann angepasst werden, indem der MontiCore-isabelle-Generator durch Implementieren der Java-Schnittstelle `LexTypeConverter` erweitert wird. Hierzu ist es erforderlich, die Methode `translate` zu implementieren, die einen Lexer-Typ in einen HOL-Typ (als String kodiert) übersetzt. Abbildung 6.2 zeigt, dass der Typ `Foo` aus der Grammatik ebenfalls in eine Zeichensequenz übersetzt werden soll. Natürlich kann für `Foo` auch jeder andere eingebaute oder selbst definierte HOL-Datentyp verwendet werden.

Die in der Grammatik in geschweiften Klammern angegebenen Übersetzungsfunktionen spielen bei der Instantiierung konkreter Modelle eine wichtige Rolle. Ergänzend zu den Übersetzungsfunktionen in der Grammatik, die das Instantiieren der passenden Java-Objekte für den AST erlauben, muss später eine Möglichkeit geschaffen werden, solche Übersetzungsfunktionen auch für Isabelle/HOL zur Verfügung zu stellen, um so einen Wert zu generieren, der zu dem generierten Isabelle/HOL-Typ passt (siehe Abschnitt 6.2).

6.1.2 Kontextfreie Syntax

Die kontextfreie Syntax wird mit Hilfe der Parserproduktionen im MontiCore-Grammatikformat definiert. Es folgt eine Beschreibung der in MontiCore zur Verfügung stehenden Konstrukte und ihrer Umsetzung in Isabelle/HOL-Datentypen. Zur Parser-Generierung wird in MontiCore Antlr [Ant10] eingesetzt und erweitert.

Klassenproduktionen

Herkömmliche Produktionen in MontiCore-Grammatiken werden als Klassenproduktionen bezeichnet, da jede linke Seite zu einer Java-Klasse der AST-Struktur führt. Abbildung 6.3 zeigt eine MontiCore-Grammatik `CDSimp` mit drei Klassenproduktionen.

CDSimp

MontiCore-Grammatik

```
 1  grammar CDSimp {
 2
 3    token CARDINALITY = (’0’..’9’)+ | ’*’ : x -> int :
 4        {if (x.getText().equals("*")) return -1;
 5         else return Integer.parseInt(x.getText()); };
 6
 7    CDDef = "cd" Name "{" (CDClass | CDAssoc)* "}";
 8
 9    CDClass = "class" Name;
10
11    CDAssoc = "assoc" left:Name (cardl:CARDINALITY)?
12                           "--" (cardr:CARDINALITY)? right:Name;
13
14  }
```

CDSimpAS

Isabelle-Theorie (gen)

```
 1  theory CDSimpAS
 2  imports "Main"
 3  begin
 4
 5  types CARDINALITY = "int"
 6
 7  types Name = "char list"
 8
 9  types STRING = "char list"
10
11  datatype CDAssoc =
12      CDAssoc Name "CARDINALITY option" "CARDINALITY option" Name
13
14  datatype CDClass = CDClass Name
15
16  datatype CDDef = CDDef Name "CDAssoc list" "CDClass list"
17
18  end
```

Abbildung 6.3: Klassenproduktionen

In den nachfolgenden Beispielen verwenden wir meist dieses Beispiel einfacher Klassendiagramme, dass je nach darzustellendem Aspekt erweitert, umgebaut oder nur in Auszügen dargestellt wird.

Auf den rechten Seiten von Produktionen werden Terminalsymbole in Anführungszeichen gesetzt (z.B. `"cd"` in Zeile 7). Nichtterminale können mit einem zusätzlichen Namen gefolgt von einem Doppelpunkt versehen werden. Dieser Name dient zur spezifischen Benennung von Attributen im AST (zum Beispiel `left:Name`, Zeile 11). Des Weiteren gibt es die üblichen Konstrukte kontextfreier Grammatiken wie Alternative (`|`), Kleene Stern (`*` und `+`), sowie optionale Elemente (`?`). Standardmäßig verfügbar sind auch Lexer-Regeln für Namen bzw. Identifikatoren (`Name`) und Zeichenketten (`STRING`).

Die resultierende abstrakte Syntax ist im zweiten Teil von Abb. 6.3 zu finden. Für jede Grammatik entsteht eine Isabelle/HOL-Theorie mit demselben Namen und dem Suffix `AS`. Bedingt durch das inkrementelle Parsen der Isabelle-Theorien müssen Datentypvereinbarungen vor der Verwendung deklariert werden. Bei der Generierung findet also eine Abhängigkeitsanalyse statt, um die korrekte Reihenfolge der Deklarationen zu bestimmen. Es entsteht so häufig eine im Vergleich zur Grammatik umgekehrte Reihenfolge der Deklarationen. Die Produktion `CDDef` liefert den Datentyp `CDDef` mit den Konstruktorargumenten, die auch in der Grammatik zu finden sind. Benutzerdefinierte Namen werden nicht übernommen, verschachtelte Verwendung von Grammatikkonstrukten wie Alternativen oder Kleene Stern werden bereits in der MontiCore-Symboltabelle analysiert und eine flache Struktur von Attributen bzw. Argumenten erzeugt. Der Kleene Stern wird auf HOL-Listen abgebildet. Für das Beispiel entstehen für `CDDef` also zwei Listen für die Produktionen `CDClass` und `CDAssoc`. Der Datentyp `CDClass` besteht nur aus einem Bezeichner. Die Argumente eines Konstruktors für Assoziationen `CDAssoc` sind zwei Bezeichner und zwei optionalen Kardinalitäten. Optionale Grammatikelemente verwenden den Isabelle-Typ `option`.

MontiCore-Grammatiken enthalten standardmäßig implizite Lexer-Regeln für Namen und Zeichenketten (Strings). Dies führt zu den generierten Typvereinbarungen `Name` und `STRING` (Zeilen 7 und 9). Weiterhin entsteht die bereits besprochene Übersetzung für die Lexer-Regel `CARDINALITY`. Die Theorie importiert die Standard HOL-Bibliothek `Main`.

Enumerationsproduktionen und Konstanten

MontiCore erlaubt die Definition von Konstanten bzw. Konstantengruppen. Ihre Verwendung wird in Abb. 6.4 demonstriert. In Zeile 3 wird die Konstantengruppe `TP` definiert, wobei eine der angegebenen Alternativen `design` oder `impl` beim Parsen gefunden werden muss. In der abstrakten Syntax wird das so dargestellt, dass ein HOL-Datentyp `CDDefTP` (Zeile 9) generiert wird, der genau diese Alternativen darstellen kann. Dieser Typ wird dann in Zeile 11 verwendet.

Eine einzelne Konstante innerhalb einer Alternative (vgl. Zeile 5) führt in MontiCore normalerweise zu einem bool'schen Attribut, das gesetzt wird, je nachdem, ob die Konstante beim Parsen gefunden wurde, oder nicht. In HOL verzichten wir auf diese Sonderbehandlung und behandeln auch einzelne Konstanten wie Konstantengruppen. So ist statt eines Arguments vom Typ bool noch der Name der Konstante und damit seine intendierte Verwendung erkennbar.

Die Übersetzung von Enumerationsproduktionen in Abb. 6.5 erfolgt analog zur Übersetzung von Konstantengruppen. Basierend auf den intern verwendeten Namen der Alternativen (Namen wie `PUB` oder `PRIV` im Beispiel können explizit vergeben werden) wird ein Datentyp generiert und verwendet, der alle Alternativen als Typkonstruktoren ohne Parameter auflistet, vgl. zweiter Teil von Abb. 6.5, Zeile 4.

CDSimp

MontiCore-Grammatik

```
grammar CDSimp { //...

  CDDef = "cd" TP:["design"|"impl"] Name "{" CDClass* "}";

  CDAssoc =  BIN:["binary"] "assoc" left:Name "--" right:Name
            |               "assoc" Name*;
}
```

CDSimpAS

Isabelle-Theorie (gen)

```
theory CDSimpAS imports "Main"
begin (*...*)

datatype CDAssocBIN = CDAssocBINBINARY

datatype CDAssoc =
    CDAssoc "Name list" "CDAssocBIN option" "Name option" "Name option"

datatype CDDefTP = CDDefTPDESIGN | CDDefTPIMPL

datatype CDDef = CDDef CDDefTP Name "CDAssoc list"

end
```

Abbildung 6.4: Konstanten

CDSimp

MontiCore-Grammatik

```
grammar CDSimp { //...

  CDClass = CDModifier* "class" Name;

  enum CDModifier =   PUB:"public" | PUB:"+"
                    | PRIV:"private" | PRIV:"-";
}
```

CDSimpAS

Isabelle-Theorie (gen)

```
theory CDSimpAS imports "Main"
begin (*...*)

datatype CDModifier = CDModifierPRIV | CDModifierPUB

datatype CDClass = CDClass "CDModifier list" Name

end
```

Abbildung 6.5: Enumerationsproduktionen

Schnittstellen

Abbildung 6.6 zeigt die Verwendung von Schnittstellen (Interfaces) in MontiCore-Grammatiken. Eine Schnittstellenproduktion wird mit `interface` eingeleitet (Zeile 5) und besitzt typischerweise keine rechte Seite. Produktionen, die eine oder mehrere Schnittstellen realisieren, verwenden das Schlüsselwort `implements` gefolgt von den Namen der Schnittstellen (z.B. Zeile 7). Die verwendete Schnittstelle (Zeile 3) steht stellvertretend für alle Produktionen, die sie implementieren. Diese sind als Alternativen beim Parsen erlaubt. Schnittstellen können andere Schnittstellen auch erweitern. Angezeigt wird dies durch das Schlüsselwort `extends`.

CDSimp

MontiCore-Grammatik

```
1   grammar CDSimp {
2
3     CDDef = "cd" Name "{" CDElem* "}";
4
5     interface CDElem;
6
7     CDClass implements CDElem = "class" Name;
8
9     CDAssoc implements CDElem = "assoc" left:Name "--" right:Name;
10  }
```

CDSimpAS

Isabelle-Theorie (gen)

```
1   theory CDSimpAS imports "Main"
2   begin (*...*)
3
4   datatype CDClass = CDClass Name
5
6   datatype CDAssoc = CDAssoc Name Name
7
8   datatype CDElem =
9       CDElemCDClass CDClass
10    | CDElemCDAssoc CDAssoc
11
12  datatype CDDef = CDDef Name "CDElem list"
13
14  end
```

Abbildung 6.6: Schnittstellenproduktionen

Die Umsetzung von Schnittstellen in einen HOL-Datentyp erfolgt durch Berechnung all ihrer implementierenden Produktionen oder sie erweiternden Schnittstellen. Für jede Schnittstelle wird ein Datentyp generiert, der alle gefundenen Alternativen auflistet (vgl. Zeilen 8-10 in Abb. 6.6, zweiter Teil). Hier wird der Unterschied zur herkömmlichen Umsetzung von Schnittstellen in objektorientierten Systemen deutlich. Der Datentyp für die Schnittstelle muss angepasst werden, wenn neue Implementierungen für die Schnittstelle hinzugefügt werden. Innerhalb einer Grammatik stellt dies kein Problem dar, da die Theorie der abstrakten Syntax bei Änderung der Grammatik neu generiert wird. Zugunsten eines modularen Aufbaus der Datentypen für vererbte Sprachen schränken wir aber aus diesem Grund die Grammatikvererbung später so ein, dass zum Beispiel Schnittstellen nicht über Grammatikgrenzen implementiert werden dürfen. Es besteht zudem ein Unterschied zwischen dem MontiCore-AST und dem Isabelle-Datentyp, da „Zwischenknoten“ für Schnittstellen nicht im AST wohl aber im Datentyp vorhanden sind.

Neben `implements` und `extends` können auch `astimplements` und `astextends` verwendet werden. Mit `ast` beginnende Erweiterungen werden nicht zu parsebaren Alternativen, sondern existieren nur in der abstrakten Syntax. Für diese Fälle ist aber keine Sonderbehandlung notwendig, sie werden wie `implements` bzw. `extends` behandelt.

AST-Regeln

CDSimp

MontiCore-Grammatik

```
1  grammar CDSimp { //...
2
3    CDAssoc = "assoc" left:Name "--" right:Name;
4
5    ast CDAssoc = id:/int;
6    ast CDAssoc = method public String getName() {return left+right+id;};
7  }
```

CDSimpAS

Isabelle-Theorie (gen)

```
1  theory CDSimpAS imports "Main"
2  begin (*...*)
3
4  datatype CDAssoc = CDAssoc int Name Name
5
6  end
```

Abbildung 6.7: AST-Regeln

MontiCore-Grammatiken sind dafür geeignet, eine abstrakte Syntax zu spezifizieren, die zusätzliche, nicht direkt Parser-relevante Elemente enthalten kann. Hierfür können mit Hilfe des Schlüsselwortes `ast` zusätzliche Strukturen für die AST-Klassen erzeugt werden. In Abb. 6.7 ist illustriert, wie die Produktion `CDAssoc` durch ein zusätzliches Attribut vom Typ `int` erweitert wird (Zeile 5). Zusätzlich wird eine Java-Methode `getName` in Zeile 6 angegeben.

In HOL muss nur das entsprechende Konstruktorargument `int` (Zeile 4, zweiter Teil der Abb. 6.7) eingefügt werden. Die Übersetzung des Typs `int` aus der Grammatik geschieht analog zur Übersetzung von Lexer-Produktionen. Eine Alternative zur Verwendung von AST-Produktionen, nämlich die Kennzeichnung eines Nichtterminals als handcodiert durch einen Schrägstrich vor der Produktion wurde bei der Isabelle-Übersetzung nicht berücksichtigt. Ebenso wenig werden Methoden der AST-Klassen auf HOL-Funktionen abgebildet.

Assoziationen

Wie bereits erwähnt, lassen sich mit MontiCore-Grammatiken Konzepte aus der Metamodellierung verwenden. So ist es möglich, Assoziationen in einer MontiCore-Grammatik anzugeben.

Assoziationen werden durch `association` eingeleitet. Zwei Beispiele sind in Abb. 6.8 (Zeilen 10, 12) angegeben. Die Assoziationen sollen in einem Statechart die Verbindung zwischen Zuständen und Transitionen herstellen. Rollennamen (`source` und `target`) machen die Assoziationen unterscheidbar.

Übersetzt werden Assoziationen in zusätzliche Attribute des Datentyps für die entsprechende Produktion, die an der Assoziation teilnimmt. Dementsprechend enthält der Datentyp für Transitionen `SCTransition` jetzt zwei weitere Argumente vom Typ `SCState`. In diesem Beispiel ist nur die Spezifikation der Assoziation enthalten. Die Instantiierung bzw. Belegung der aus der Assoziation generierten Attribute ist nicht Teil des Beispiels. Hierfür gibt es viele Umsetzungsmöglichkeiten. Eine einfache Möglichkeit bietet das MontiCore-Konzept `sreference` (Zeile 14, Abb. 6.8, oberer Teil), das für eine flachen Namensraum eine Infrastruktur generiert, die zum Füllen der Assoziationslinks verwendet werden kann.

SCSimp

MontiCore-Grammatik

```
1  grammar SCSimp {
2    SCDef = "sc" Name "{" SCElem* "}";
3
4    interface SCElem;
5
6    SCState implements SCElem = "state" Name;
7
8    SCTransition implements SCElem = from:Name "-->" to:Name;
9
10   association TransitionFrom SCState 1 <- SCTransition.source;
11
12   association TransitionTo   SCState 1 <- SCTransition.target;
13
14   concept sreference {
15     TransitionFrom: SCState.name = SCTransition.from;
16     TransitionTo: SCState.name = SCTransition.to;
17   }
18 }
```

SCSimpAS

Isabelle-Theorie (gen)

```
theory SCSimpAS imports "Main"
begin (*...*)

datatype SCState = SCState Name

datatype SCTransition = SCTransition SCState SCState Name Name

datatype SCElem =
    SCElemSCState SCState
  | SCElemSCTransition SCTransition

datatype SCDef =  SCDef Name "SCElem list"

end
```

Abbildung 6.8: Assoziationen

Optionen

Jede MontiCore-Grammatik kann einen Block enthalten, indem Optionen angegeben werden können.

CDSimp

MontiCore-Grammatik

```
grammar CDSimp { //...

  options {
    compilationunit CDDef
  }

  CDDef = "cd" Name "{" CDElem* "}";
}
```

CDSimpAS

Isabelle-Theorie (gen)

```
theory CDSimpAS imports "Main"
begin (*...*)

datatype CDDef = CDDef Name "CDElem list"

datatype MCCompilationUnit =
    MCCompilationUnit "Name list" "Name list" "STRING option" CDDef
end
```

Abbildung 6.9: Compilation Unit

identrule Der hinter dieser Option angegebene Name bestimmt den Namen der Regel für Identifikatoren. Der Name wird auch in den HOL-Datentypen berücksichtigt. Standardmäßig wird `Name` als Name für die Identifikatorregel verwendet.

noident, nostring Hiermit lassen sich die Generierung der Lexer-Regeln für Identifikatoren und `STRING` verhindern. Diese werden dann auch nicht automatisch als HOL-Typen zur Verfügung gestellt.

nomlcomment, noslcomment Mit diesem Optionen können die Standardkommentare abgeschaltet werden. Diese Option betrifft nicht die HOL-Typen, da Kommentare nicht übernommen werden.

dotident Hier wird eine Identifikatoregel gewählt, bei der Identifikatoren Punkte beinhalten können. Diese ist dann automatisch auch im HOL-Datentyp enthalten.

lookahead, noanything, nocharvocab, header, prettyprinter Diese Optionen sind entweder Parser-, Lexer- oder DSLTool-spezifisch und müssen nicht berücksichtigt werden.

compilationunit Abb. 6.9 zeigt die Verwendung der Option `compilationunit`, mit der eine Startproduktion für die Grammatik festgelegt werden kann. Die Angabe von `CDDef` als Startproduktion führt zu einem Datentyp `MCCompilationUnit` in HOL, wobei eine Liste mit import-Statements, der Paketangabe und eine Versionsnummer (Typ `String`) angegeben werden kann, bevor der Datentyp `CDDef` aufgebaut wird. Wir verwenden die Information der Startproduktion in dieser Option auch, um zu entscheiden, ob es sich bei einem konkreten Modell um ein eigenständiges Modell handelt. Ist dies der Fall, wird zusätzliche der Kopf und Fuß der Isabelle-Theorie generiert.

Prädikate

In MontiCore-Grammatiken können die aus Antlr [Ant10] bekannten syntaktischen und semantischen Prädikate eingesetzt werden. Syntaktische Prädikate sind Hilfestellungen für den Parser und in manchen Fällen notwendig, um eine Ambiguität in der Grammatik aufzulösen. Sie helfen also, die konkrete Syntax zu parsen und sind damit für den HOL-Datentyp nicht relevant. Semantische Prädikate sind allgemeine Java-Ausdrücke, die beim Parsen ausgeführt werden können. Eine mögliche Anwendung ist die Überprüfung von einfachen Kontextbedingungen schon während des Parsens. In unserem Ansatz sollen Kontextbedingungen separat beschrieben werden. Daher spielen semantische Prädikate für uns keine Rolle und sind in ihrer allgemeinen Form auch nicht umsetzbar.

Variablen und Produktionsparameter

CDSimp

MontiCore-Grammatik

```
1  grammar CDSimp {
2
3    CDDef = "cd" Name "{" ("class" n=Name CDClass[|n|])* "}";
4
5    CDClass [|Name|];
6  }
```

CDSimpAS

Isabelle-Theorie (gen)

```
1  theory CDSimpAS imports "Main"
2  begin (*...*)
3
4  datatype CDClass = CDClass "Name option"
5
6  datatype CDDef = CDDef Name "CDClass list"
7
8  end
```

Abbildung 6.10: Variablen und Produktionsparameter

In MontiCore-Grammatiken können innerhalb von Produktionen Variablen zugewiesen und ihr Wert an andere Produktionen weitergegeben werden. Dies ist in Abb. 6.10 dargestellt. In Zeile 3 (ersten Teil der Abbildung) wird nach dem Terminalsymbol `class` ein Name geparst und dieser Wert der Variablen `n` zugewiesen. Hier wird der zweite `Name` nicht Teil der abstrakten Syntax von `CDDef`. Anschließend wird der Wert an die Produktion `CDClass` übergeben, erkennbar an der speziellen Klammerung. Die Produktion `CDClass` erhält den Wert, der sich auch in ihrer abstrakten Syntax als optionales Element manifestiert, vgl. dazu Abb. 6.10, zweiter Teil, Zeile 4.

CDSimp

MontiCore-Grammatik

```
1   grammar CDSimp {
2     CDDef = "cd" Name ComplexElem*;
3
4     ComplexElem returns CDElem =
5       ret=CDElem | astscript{!();} "{" CDElem ("," CDElem)* "}";
6
7     interface CDElem;
8     CDClass implements CDElem = "class" Name;
9     CDAssoc implements CDElem = "assoc" left:Name "--" right:Name;
10  }
```

CDSimpAS

Isabelle-Theorie (gen)

```
1   theory CDSimpAS imports "Main"
2   begin (*...*)
3
4   datatype CDClass = CDClass Name
5
6   datatype CDAssoc = CDAssoc Name Name
7
8   datatype CDElem =
9       CDElemComplexElem ComplexElem
10    | CDElemCDClass CDClass
11    | CDElemCDAssoc CDAssoc
12  and
13  ComplexElem = ComplexElem "CDElem list"
14
15  datatype CDDef = CDDef Name "CDElem list"
16
17  end
```

Abbildung 6.11: Manipulation des Rückgabetyps einer Produktion

In Abb. 6.11 führen wir die Produktion `ComplexElem` ein, die eine Zusammenfassung verschiedener Elemente darstellen soll: Entweder handelt es sich um ein einzelnes Element oder einen Block von Elementen. Für viele einzelne Elemente erhalten wir so nicht auch viele `ComplexElem`, die jeweils nur aus den einzelnen Elementen bestehen, sondern direkt die Elemente `CDClass` oder `CDAssoc`. In Zeile 5 wird hierfür das Rückgabeobjekt der Produktion mit Hilfe von `ret` so manipuliert, dass sich der Rückgabewert direkt aus dem Wert der Produktion `CDElem` ergibt, wenn diese Alternative gewählt wurde. Durch das Schlüsselwort `returns` wird dann der Rückgabetyp der Produktion auf `CDElem` festgelegt.

In der abstrakten Syntax ist zu erkennen, dass `CDDef` tatsächlich aus `CDElem` und nicht aus `ComplexElem` aufgebaut wird und das Interface `CDElem` um `ComplexElem` erweitert wurde. Hier tritt auch eine rekursive Abhängigkeit der Definitionen von `ComplexElem` und `CDElem` auf. Dies hat in HOL die Konsequenz, dass beide Datentypen gemeinsam definiert und dabei durch `and` (Zeile 12) verbunden werden müssen. Die Analyse von zyklischen Abhängigkeiten in Definitionen geschieht automatisch durch Berechnung von starken Zusammenhangskomponenten mit Hilfe von Tarjans Algorithmus [Tar72].

Aktionen

Mit Hilfe des Schlüsselwortes `astscript` können Aktionen zu Regeln angegeben werden, die nach dem Parser der entsprechenden Stelle eines Modells ausgeführt werden. Es gibt einige vordefinierte Aktionen, beispielsweise zur Konstruktion neuer Objekte. Ansonsten ist die Angabe beliebigen Java-Codes möglich. Da die Ausführung der Aktionen bei der Verarbeitung konkreter Modelle geschieht und die angelegten Datenstrukturen nicht geändert werden, haben Aktionen keine Auswirkung auf die Struktur der abstrakten Syntax und brauchen nicht berücksichtigt werden.

Mehrteilige Klassenproduktionen

Normalerweise führt eine Klassenproduktion zu einem Datentyp in der abstrakten Syntax. Es ist aber auch möglich, die Datentypdefinition über mehrere Produktionen zu verteilen. In Abb. 6.12 wird dies demonstriert. In Zeile 3 (oberer Teil) werden die beiden Produktionen `CDElem` und `CDInterface` verwendet. Bei der Definition von `CDInterface` wird ihr Rückgabetyp nach

CDSimp

MontiCore-Grammatik

```
1  grammar CDSimp { //...
2
3    CDDef = "cd" Name "{" (all:CDElem|all:CDInterface)* "}";
4
5    CDElem = "class" Name;
6
7    CDInterface:CDElem = IsInterface:["interface"] Name;
8  }
```

CDSimpAS

Isabelle-Theorie (gen)

```
1  theory CDSimpAS imports "Main"
2  begin (*...*)
3
4  datatype CDElemIsInterface = CDElemIsInterfaceINTERFACE
5
6  datatype CDElem = CDElem Name "CDElemIsInterface option"
7
8  datatype CDDef = CDDef Name "CDElem list"
9
10 end
```

Abbildung 6.12: Mehrteilige Klassenproduktionen

dem Doppelpunkt allerdings auf `CDElem` festgelegt (Zeile 7). In der abstrakten Syntax wird folglich nur ein Datentyp `CDElem` generiert und verwendet. Die Attribute dieses Datentyps entstehen aus der Vereinigung der einzelnen Definitionen und werden dabei optional, wenn sie nur in einer der Alternativen vorkommen, vgl. Zeile 6 der HOL-Definition. Attribute mit demselben Namen und demselben Typ in unterschiedlichen Produktionen werden also als gleich angesehen.

Ausdrücke

Expression

MontiCore-Grammatik

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

Abbildung 6.13: MontiCore-Grammatik für Ausdrücke

Ausdrücke unterscheiden sich von anderen Teilen einer Sprache meist durch die Existenz einer Vielzahl von AST-Elementen, die fast beliebig zu einer Baumstruktur zusammengesetzt werden können. Die spezifische Realisierung von Ausdrucksstrukturen mit MontiCore verfolgt zwei Ziele. Zum einen soll die Baumstruktur nach dem Parsen möglichst flach sein und nur die wirklich aufgetretenen Operatoren beinhalten. Künstliche Zwischenknoten, die durch die Beachtung der Bindungsstärke von Operatoren entstehen können, sollen vermieden werden. Zum anderen sollen Operatoren gleicher Bindungsstärke in einer Produktion definierbar sein, um so die Anzahl der benötigten Produktionen (und folglich Datentypen in der abstrakten Syntax) möglichst gering zu halten.

Abb. 6.13 zeigt die Definition einer einfachen Ausdruckssprache mit Rechenoperationen für natürliche Zahlen. Die Operatoren + und − haben dabei wie gewohnt eine niedrigere Priorität als `*` und `/`. Dies wird dadurch erreicht, dass in der Regel `AddSub` (Zeile 8) zunächst versucht wird, mit der Produktion `MultDiv` fortzufahren. Die flache Baumstruktur entsteht tatsächlich. Wenn beispielsweise eine Multiplikation geparst wird, wird der Rückgabetyp mittels `ret` auf diesen Typ gesetzt. Weitere Erklärungen zum systematischen Aufbau von Ausdruckssprachen

ExpressionAS

Isabelle-Theorie (gen)

```
1  theory ExpressionAS imports "Main"
2  begin (*...*)
3
4  types NUMBER = "int"
5
6  datatype NumLiteral = NumLiteral NUMBER
7
8  datatype Variable = Variable Name
9
10 datatype Literal =
11     LiteralNumLiteral NumLiteral
12   | LiteralVariable Variable
13
14 datatype InfixExpOp =
15     InfixExpopPLUS
16   | InfixExpopMINUS
17   | InfixExpopDIV
18   | InfixExpopMULT
19
20 datatype InfixExp = InfixExp Expression Expression InfixExpOp
21 and
22 Expression =
23     ExpressionInfixExp InfixExp
24   | ExpressionLiteral Literal
25
26 end
```

Abbildung 6.14: Isabelle-Theorie für Ausdrücke

sind in [Kra10] beschrieben.

Die abstrakte Syntax in der Theorie in Abb. 6.14 entspricht der angestrebten flachen Baumstruktur. Ein Ausdruck ist entweder eine Infix-Expression oder ein Literal (Zeile 22). Eine Infix-Expression hat eine linke und eine rechte Seite, die mit einem Operator verknüpft sind (Zeile 20). Die Verknüpfung der beiden HOL-Datentypen mit `and` ist aufgrund der typischen rekursiven Struktur der Ausdruckssprache notwendig.

6.1.3 Grammatikvererbung

MontiCore-Grammatiken können von anderen MontiCore-Grammatiken erben und dabei neue Produktionen hinzufügen oder existierende Produktionen ersetzen oder erweitern. Mehrfachvererbung ist möglich. Die MontiCore-Parser-Generierung erzeugt für jede Untergrammatik einen neuen Parser, der auch die Produktionen der Obergrammatiken enthält. Bei der Erzeugung der AST-Klassen wird die Erweiterbarkeit der Java-Klassen ausgenutzt, so dass diese wiederverwendet werden können.

Die Grammatikvererbung kann bei der Abbildung auf HOL-Datenstrukturen nicht uneingeschränkt verwendet werden. Sie wird in unserem Ansatz nur im Sinne einer Erweiterung der Sprache unter Verwendung der Obersprachen verstanden. Modifikationen an bestehenden Definitionen sind nicht erlaubt. Das liegt an der beschränkten Möglichkeit von HOL-Datentypen, erweiterbare Objektstrukturen darstellen zu können. Als Konsequenz ist es beispielsweise nicht

Common

MontiCore-Grammatik

```
package inh.common;

grammar Common {

  Stereotype =  "<<" values:STRING* ">>";
}
```

CDSimp

MontiCore-Grammatik

```
grammar CDSimp extends inh.common.Common { //...

  CDDef= Stereotype? "cd" Name "{" CDElem* "}";
}
```

Abbildung 6.15: MontiCore-Grammatikvererbung

erlaubt, ein Interface einer Obergrammatik in einer Untergrammatik zu implementieren, da hierdurch der Datentyp der Obergrammatik für das Interface angepasst werden müsste.

Drei Alternativen zur gewählten Umsetzungen sind denkbar:

- Grammatikvererbung wird aufgelöst, indem eine einzige Grammatik gebildet wird, die alle Produktionen der gesamten Hierarchie enthält. Damit würde aber die Möglichkeit eingebüßt, eine Sprache und insbesondere ihre Semantik modular, das heißt Ober- und Untergrammatik getrennt, zu entwickeln oder die Semantik einer Obergrammatik bei der Definition der Semantik der Untergrammatik einfach wiederverwenden zu können.

- Für eine erweiternde Produktion in einer Untergrammatik wird auch die erweiterte Definition der Obergrammatik in der Untergrammatik erneut definiert und mit Alternativen für die bisherige und die neue Definition ausgestattet. Alle Verwendungsstellen des ursprünglichen Datentyps müssen gefunden und durch diesen neuen Datentyp ersetzt werden. Wie in der ersten Umsetzung findet so effektiv eine Auflösung der Grammatikvererbung statt, die allerdings nur lokale Auswirkungen auf die redefinierten Datentypen hat.

- Jeder Datentyp wird erweiterbar definiert. Dies ist in HOL jedoch nur unter erheblichem Aufwand möglich. Eine solche erweiterbare Hierarchie von Datentypen, die objektorientierten Klassenhierarchien ähnelt, wird in [BW06] entworfen. Der Nachteil ist, dass die Verständlichkeit der Definitionen erheblich leidet.

Grammatikvererbung wird in den betrachteten Fallstudien so verwendet, dass die erwähnte Einschränkung für uns keine Auswirkung hat und wir davon profitieren können, jede Grammatik separat übersetzen und verwenden zu können.

Abbildungen 6.15 und 6.16 zeigen das generelle Vorgehen bei der Umsetzung von Grammatikvererbung in Isabelle. Grammatik `CDSimp` erweitert die Grammatik `Common` unter Angabe des vollqualifizierten Namens. Die generierte Theorie für die Untergrammatik `CDSimp` (Abbildung 6.16) importiert die Theorien der Obergrammatiken. Dabei wird eine projektspezifische

CommonAS

Isabelle-Theorie (gen)

```
theory CommonAS imports "Main"
begin (*...*)

datatype Stereotype =
    Stereotype "STRING list"
end
```

CDSimpAS

Isabelle-Theorie (gen)

```
theory CDSimpAS
imports "$EXT/gen/inh/common/CommonAS"
begin (*...*)

datatype CDDef =
    CDDef "Stereotype option" Name "CDElem list"
end
```

Abbildung 6.16: Umsetzung der Grammatikvererbung in Isabelle-Theorien

Umgebungsvariable (in unserem Beispiel `$EXT`) vor den Pfad zur generierten Theorie der Obergrammatik angefügt, um Sprachen projektübergreifend wiederverwenden zu können und einen eindeutigen Pfad bereitzustellen. Diese Umgebungsvariable ist ein Konfigurationsparameter des MontiCore-isabelle-Generators.

6.1.4 Einbettung

Oft ist es sinnvoll, mehrere Sprachen miteinander zu kombinieren. Ein bekanntes Beispiel aus der UML ist, das Verhalten einer Klassen mit Hilfe eines Statechart zu beschreiben und Invarianten in Zuständen oder logische Vorbedingungen von Transitionen mittels OCL auszudrücken. Hierbei wird OCL in Statecharts als Sprachparameter verwendet, da eine Festlegung auf OCL nicht notwendig ist und prinzipiell auch andere Constraint-Sprachen verwendet werden könnten. Technisch erlaubt es das MontiCore-Grammatikformat, Nichtterminale durch das Schlüsselwort `external` zu kennzeichnen. An dieses Nichtterminal können dann Produktionen anderer Sprachen gebunden werden. Dieser Mechanismus wird in MontiCore als Einbettung bezeichnet.

In Abb. 6.17 werden zwei verschiedene Formen der Spracheinbettung verwendet. Zunächst werden zwei externe Nichtterminale (Zeilen 3 und 13) definiert. Das erste Nichtterminal für Invarianten `Invariant` wird in der Produktion `CDDef` verwendet. Das zweite Nichtterminal für eine Block von Anweisungen `StmtBlock` wird in der Produktion `CDCode` mit einem Parameter `tp` vom Typ `Name` parametrisiert, erkennbar an den spitzen Klammern hinter `StmtBlock`. Beim Parsen kann so abhängig vom Wert des Attributs `tp` ein Parser ausgewählt werden, der unter diesem Namen registriert ist.

In der abstrakten Syntax (vgl. zweiter Teil der Abb. 6.17) wird Einbettung über den Import einer speziellen Theorie `ExternalCDSimpAS` gelöst. Wir nehmen für den Moment an, dass in dieser Theorie die fehlenden HOL-Datenstrukturen `StmtBlock` und `Invariant` definiert

CDSimp

MontiCore-Grammatik

```
grammar CDSimp { //...

  external Invariant;

  CDDef = "cd" Name ("[" Invariant "]")? "{" CDElem* "}";

  interface CDElem;

  CDClass implements CDElem = "class" Name "{" CDAttr* "}";

  CDAttr = (type:Name)? Name ";";

  external StmtBlock;

  CDCode implements CDElem = tp:Name "code" StmtBlock<tp>;
}
```

CDSimpAS

Isabelle-Theorie (gen)

```
theory CDSimpAS
imports "$EXT/def/emb/ExternalCDSimpAS"
begin (*...*)

datatype CDAttr = CDAttr "Name option" Name

datatype CDClass = CDClass Name "CDAttr list"

datatype CDCode = CDCode Name StmtBlock

datatype CDElem =
    CDElemCDClass CDClass
  | CDElemCDCode CDCode

datatype CDDef = CDDef Name "Invariant option" "CDElem list"

end
```

Abbildung 6.17: Spracheinbettung in parametrisierter und unparametrisierter Form

CDSimp

MontiCore-Sprachdatei

```
1  package emb;
2
3  language CDSimp {
4
5   root CDSimpRoot<MCCompilationUnit>;
6
7   rootfactory CDSimpRootFactory for CDSimpRoot<MCCompilationUnit> {
8     CDSimp.MCCompilationUnit cd <<start>>;
9
10    emb.ocl.OCL.Invariant inv in cd.Invariant;
11
12    emb.java.Java.JBlockStatement s1 in cd.StmtBlock(java);
13    emb.cpp.CPP.CBlockStatement s2 in cd.StmtBlock(cpp);
14  }
15
16  parsingworkflow CDSimpParsingWorkflow for CDSimpRoot<MCCompilationUnit>;
17 }
```

ExternalCDSimpAS

Isabelle-Theorie

```
1  theory ExternalCDSimpAS
2  imports "$EXT/gen/emb/ocl/OCLAS"
3          "$EXT/gen/emb/java/JavaAS"
4          "$EXT/gen/emb/cpp/CPPAS"
5  begin
6
7  types Invariant = OCLAS.Invariant
8
9  datatype StmtBlock = StatementJava JavaAS.JBlockStatement
10                    | StatementCPP  CPPAS.CBlockStatement
11 end
```

Abbildung 6.18: Kombination von Sprachen

sind. Zunächst ist auch kein Unterschied zwischen den beiden Formen der Einbettung sichtbar.

In MontiCore können einzelne Sprachteile, die auch Fragmente genannt werden, über so genannten Sprachdateien kombiniert werden. Abb. 6.18 enthält eine Sprachdatei für die Grammatik aus Abb. 6.17. Hierin wird unter anderem festgelegt, dass die Produktion `Invariant` aus dem Klassendiagramm mit der Produktion `Invariant` der Sprache `OCL` kombiniert wird (Zeile 10). In den Zeilen 12 und 13 wird definiert, dass abhängig davon, ob der Parameter `tp` aus der Grammatik in Abb. 6.18 dem Wert `java` oder dem Wert `cpp` entspricht, entweder die Produktion `JBlockStatement` der Sprache `Java` oder die Produktion `CBlockStatement` der Sprache `CPP` für Statements in der Grammatik eingesetzt werden soll. Diese Konfigurationsmöglichkeit wird genau durch die bereits erwähnte importierte Theorie `ExternalCDSimpAS` bereit gestellt. Sie ist im zweiten Teil der Abb. 6.18 dargestellt. Diese Theorie wird nicht automatisch generiert, sondern muss für die gewählte Einbettung manuell kodiert werden.

Analog zur Sprachdatei wird festgelegt, dass der Typ `Invariant` dem Typ `Invariant` aus der Theorie `OCLAS` entsprechen soll. Für den Typ `StmtBlock` stehen zwei Alternativen zur Auswahl, entweder Statements der HOL-Typen in `JavaAS` oder `CPPAS`. Beim Aufbau der

Datenstrukturen für ein konkretes Modell ist darauf zu achten, dass hier die richtige Alternative entsprechend dem Wert des Attributs `tp` gewählt wird. Wie dies erreicht wird, wird unter anderem in Abschnitt 6.2 erläutert.

6.1.5 Zusammenfassung

Abbildung 6.19 illustriert die Übersetzung einer MontiCore-Grammatik `L1.mc`, die von einer Grammatik `L2.mc` erbt und auch Einbettung verwendet.

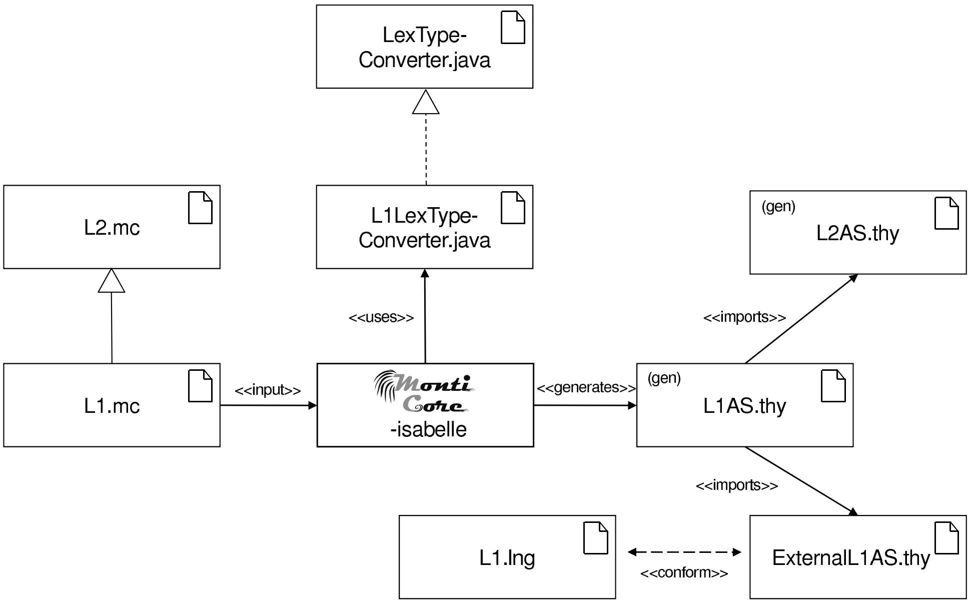


Abbildung 6.19: Übersetzungsprozess für eine MontiCore-Grammatik

- MontiCore-isabelle wird mit der handkodierten Klasse `L1LexTypeConverter` parametrisiert, die die Schnittstelle `LexTypeConverter` implementiert, um die Typen von Lexer-Regeln in HOL-Typen zu übersetzen.

- Die generierte Theorie `L1AS.thy` enthält die Repräsentation der abstrakten Syntax von `L1` in Isabelle/HOL und importiert die separat generierte Theorie der abstrakten Syntax für die Obersprache `L2AS.thy`.

- Weiterhin wird die Theorie `ExternalL1AS.thy` importiert, die die Spracheinbettung, also die noch nicht definierten Bestandteile der abstrakten Syntax von `L1`, belegt. Dies muss konform zur MontiCore-Sprachdatei `L1.lng` geschehen und manuell sicher gestellt werden.

6.2 Übersetzung konkreter Modelle

Das bisher beschriebene Verfahren bietet die Möglichkeit, die abstrakte Syntax einer MontiCore-Grammatik mit Hilfe des Generators MontiCore-isabelle in einen Isabelle/HOL-Datentyp zu übersetzen. In diesem Abschnitt wird beschrieben, dass MontiCore-isabelle zusätzlich in der Lage ist, zum generierten HOL-Datentyp einen sprachspezifischen Generator zu erzeugen, der aus konkreten Modellen eine Instanz des Isabelle/HOL-Datentyps erstellt. Das hat den Vorteil, dass automatisch auch konkrete Modelle direkt in Isabelle repräsentiert werden können. Die Erweiterung von MontiCore-isabelle ist in Abb. 6.20 veranschaulicht.

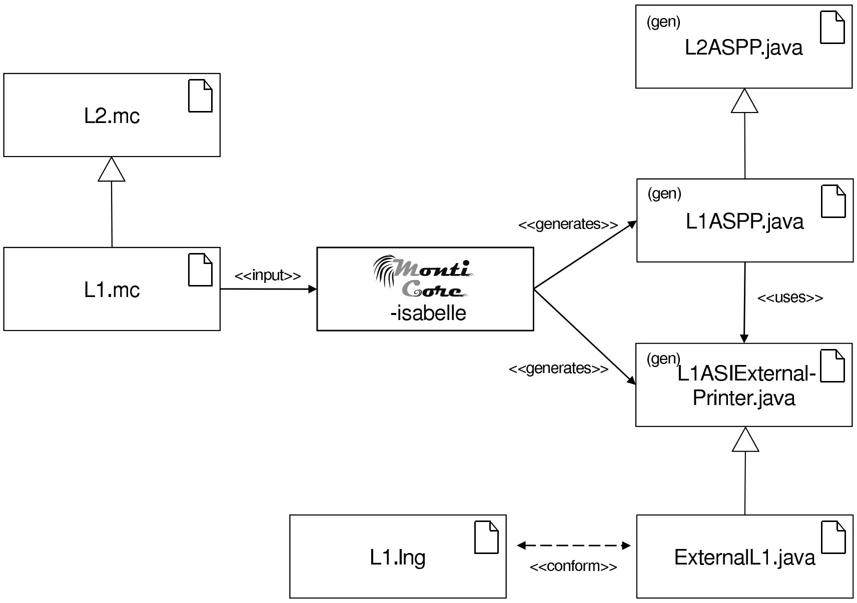


Abbildung 6.20: Generierung des Übersetzers `L1ASPP` für Modelle von `L1`

Für jede Grammatik `L1` wird eine Java-Klasse `L1ASPP` generiert, die für alle Regeln der Grammatik `print`-Methoden anbietet, um eine Java-Objekt der abstrakten Syntax in dessen Isabelle-Repräsentation zu überführen. `L1ASPP` nutzt den generierten Übersetzer `L2ASPP` für die Obersprache `L2`, um ererbte Regeln zu behandeln. Die automatische Übersetzung konkreter Modelle ist nur möglich, wenn maximal eine Obergrammatik (also keine Mehrfachvererbung von Grammatiken) verwendet wird. `L1ASIExternalPrinter` ist eine ebenfalls generierte Java-Schnittstelle, die alle Methoden auflistet, die zur Übersetzung von Instanzen von Lexer-Typen in HOL-Werte und zur Übersetzung von Modellteilen aus eingebetteten Sprachen in HOL-Konstanten noch implementiert werden müssen (hier durch die Klasse `ExternalL1`). `L1ASPP` delegiert solche Elemente der abstrakten Syntax an diese Schnittstelle. Die Implementierung der Schnittstelle muss sich nach der Konfiguration der Spracheinbettung in `L1.lng` richten. Dies ist manuell sicherzustellen. Der generierte Übersetzer für Modelle `L1ASPP` wird verwendet, indem er in einem generativen Werkzeug `L1Tool` (siehe Abb. 6.21) auf Basis des DSLTool-Framework durch einen workflow eingebunden wird.

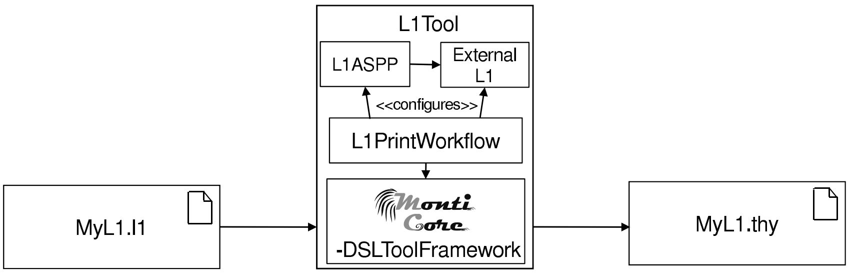


Abbildung 6.21: Einbindung des Übersetzers für Modelle in eine Werkzeug für die Sprache `L1`

Abb. 6.22 zeigt ein Beispiel für die Übersetzung eines konkreten Modells. Das Modell im ersten Teil der Abbildung ist konform zur Grammatik in Abb. 6.17. Nachdem es geparst wurde, wird der generierte Übersetzer `CDSimpASPP` verwendet, um das Modell in die darunter stehende Isabelle/HOL-Definitionen zu übersetzen.

Die generierte Theorie importiert die abstrakte Syntax aus Abb. 6.17. Es folgen eine Reihe nummerierter Definitionen. Von einer Generierung einer sprechenderen Namensgebung für die Konstanten wurde abgesehen, da diese Konstantennamen durch einfache Simplifikation in Isabelle eliminiert werden können und somit überflüssig werden. Zudem besitzen viele der generierten Elemente keinen eindeutigen Namen. Im Beispiel sind gerade vier der 19 Definitionen benannt. Das generierte Modell ist nicht robust gegenüber Änderungen. Werden Modellelemente hinzugefügt, kann sich die Nummerierung vollständig ändern. Da bei Isabelle-Beweisen die Konstantennamen eine untergeordnete Rolle spielen, hat diese Tatsache keine großen Auswirkung. Die Lesbarkeit des generierten Modells ist noch akzeptabel, da die Verweise innerhalb des Modells durch die durchgängige Nummerierung leicht nachvollzogen werden können. Die Theorie ist von unten nach oben zu lesen. Zunächst wird eine Compilation Unit konstruiert, wobei import-Statements, Paket und Version leer bleiben und sie nur die Referenz auf die Definition des Klassendiagramms in `e18` enthält. Ein Klassendiagramm besitzt einen Namen, eine Invariante, die in `e1` definiert wird (Invarianten sind in unserem einfachen Fall nur Identifikatoren) und eine Liste von `CDElem` (Zeile 20). Beide Varianten der Einbettung werden verwendet, siehe Zeilen 12-13 und 16-17.

Es ist auffällig, dass für jeden Datentyp eine eigene Deklaration eingeführt wird. Es wäre auch eine alternative Übersetzung möglich, bei der nur eine Deklaration entsteht und in der statt einem Verweis auf andere Deklarationen direkt die rechten Seiten der Deklarationen eingesetzt wird. Die gewählte Strategie zur Übersetzung eines Modells begründet sich aus der Verwendung des übersetzten Modells in Isabelle. Der modulare Aufbau erlaubt eine selektive Betrachtung und Entfaltung des Modells. Dies ist insbesondere bei großen Modellen hilfreich. Nicht gezeigt ist der Umgang mit Grammatikvererbung. Der generierte Generator delegiert für Produktionen aus der Obergrammatik die Arbeit an den genauso generierten Generator für diese Obergrammatik.

Das generierte Java-Interface `CDSimpASIExternalPrinter`, das für die Übersetzung von Modellen zusätzlich zu implementieren ist, ist in Abb. 6.23 angegeben. Es enthält zum einen Methoden, die die vom Lexer erzeugten Token in ihre Isabelle-Repräsentation umwandeln sollen. Ein Beispiel ist die Methode `convertCARDINALITY` (Zeile 4), die ein Token

MyCD10

Klassendiagramm

```
cd MyCD10 [true]  {

  class A { int a; }
  class B { }

  java code { someJCode }
  cpp code { someCppCode}
}
```

MyCD10

Isabelle-Theorie (gen)

```
theory MyCD10
imports "$EXT/gen/emb/CDSimpAS"
begin
constdefs "e1 == Invariant  ''true''  "
constdefs "e2 == CDAttr  (Some ''int'' ) ''a''  "
constdefs "e3 == (e2#[])"
constdefs "e4 == CDClass  ''A''  e3"
constdefs "e5 == CDElemCDClass e4"
constdefs "e6 == []"
constdefs "e7 == CDClass  ''B''  e6"
constdefs "e8 == CDElemCDClass e7"
constdefs "e9 == JBlockStatement  ''someJCode''  "
constdefs "e10 == StatementJava e9"
constdefs "e11 == CDCode  ''java''  e10"
constdefs "e12 == CDElemCDCode e11"
constdefs "e13 == CBlockStatement  ''someCppCode''  "
constdefs "e14 == StatementCPP e13"
constdefs "e15 == CDCode  ''cpp''  e14"
constdefs "e16 == CDElemCDCode e15"
constdefs "e17 ==  (e16#(e12#(e8#(e5#[]))))"
constdefs "e18 == CDDef  ''MyCD10''  (Some e1) e17"
constdefs "e19 == MCCompilationUnit [''''] []  None e18"
end
```

Abbildung 6.22: Übersetzung eines konkreten Modells nach Isabelle/HOL

CDSimpASIExternalPrinter

CDSimp-Tool/Java (gen)

```
public interface CDSimpASIExternalPrinter{

 public int printNode(mc.ast.ASTNode a, int c, mc.helper.IndentPrinter p) ;
 public String convertCARDINALITY(int a) ;
 public String convertName(String a) ;
 public String convertSTRING(String a) ;
}
```

Abbildung 6.23: Java-Interface zur Übersetzung von Teilen eines Modells nach Isabelle/HOL

ExternalCDSimp

CDSimp-Tool/Java

```
public class ExternalCDSimp extends ExternalAll
                              implements CDSimpASIExternalPrinter{
  CDSimpRoot root;

  public ExternalCDSimp(CDSimpRoot dslroot) {
    this.root = dslroot;
  }

  public int printNode(ASTNode a, int c, IndentPrinter p) {
    if (a instanceof emb.ocl._ast.ASTInvariant) {
      OCLASPP oclpp = new OCLASPP(root,p);
      return oclpp.printInvariant((emb.ocl._ast.ASTInvariant)a, c);
    }
  }
}
```

Abbildung 6.24: Implementierung der Schnittstelle

der lexikalischen Produktion `CARDINALITY` (vgl. Abb. 6.1, Seite 104), das vom Typ `int` ist, umwandeln muss. Zum anderen enthält das Interface Methoden, die genutzt werden, um bei Spracheinbettung den passenden Generator für die eingebettete Sprache aufzurufen. Der Methode `printNode` wird vom aufrufenden Generator ein AST-Knoten übergeben. Weitere Parameter sind die aktuelle Nummer der Konstantendefinition und der zu verwendende Printer. Nach Aufruf eines passenden Generators wird die dann aktuelle Nummer der Konstantendefinition zurückgeliefert und vom aufrufenden Generator weiterverwendet.

Abb. 6.24 zeigt einen Ausschnitt der Implementierung der Methode `printNode`. Die Implementierung behandelt den Fall, dass eine eingebettete OCL-Invariante übersetzt werden muss. Der passende Generator `OCLASPP` wird instantiiert und die entsprechende Methode zur Verarbeitung einer Invariante aufgerufen. Es ist manuell sicherzustellen, dass für die Konfiguration der Einbettung, wie sie in Abschnitt 6.1.4 beschrieben ist, der richtige Generator aufgerufen wird.

6.3 Kodierung von Kontextbedingungen

Durch die tiefe Einbettung der abstrakten Syntax einer Modellierungssprache ist es möglich, ihre Kontextbedingungen in einer Isabelle/HOL-Theorie zu kodieren. Abb. 6.25 verdeutlicht zunächst die Beziehung zwischen den benötigen Theorien, wobei die Kontextbedingungen einer Sprache `L1` in der Theorie `L1CC-base.thy` erfasst werden.

- `L1AS.thy` wurde automatisch aus einer MontiCore-Grammatik erzeugt.
- Kontextbedingungen einer Obersprache `L2` können importiert werden. Sie werden im Zuge der Definition und Konfiguration der Sprache `L2` erstellt. Die Theorie `L2CC` beinhaltet bereits alle gültigen Kontextbedingungen einer Sprachvariante von `L2` (vgl. Abschnitt 6.6).

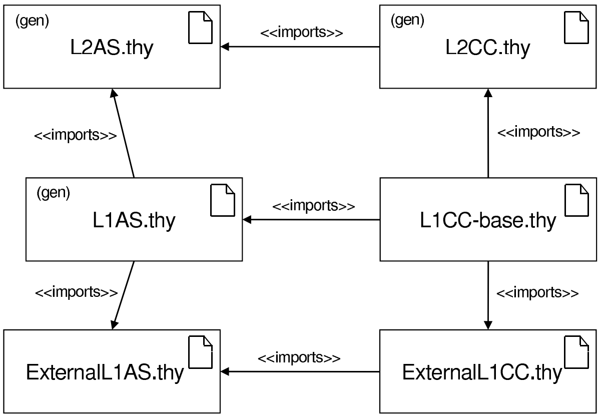


Abbildung 6.25: Theorie `L1CC-base` für Kontextbedingungen mit Abhängigkeiten

- Kontextbedingungen für Sprachparameter sind manuell in die Theorie `ExternalL1CC` zu importieren. Die Theorie muss selbst die abstrakte Syntax der Sprachparameter (in der Theorie `ExternalL1AS`) importieren und ist damit automatisch konform zur Konfiguration der Einbettung durch die Sprachdatei `L1.lng`.

- Auf Basis der wiederverwendbaren Kontextbedingungen der Obergrammatik und eingebetteter Sprachen werden die notwendigen Kontextbedingungen für `L1` in der Theorie `L1CC-base` kodiert.

- Abschließend muss das Prädikat `wellformed-base` in Theorie `L1CC-base` definiert werden, das alle kodierten Kontextbedingungen zusammenfasst. Wir legen als Namenskonvention fest, dass jede Theorie für Kontextbedingungen ein Prädikat dieses Namens bereitstellen muss, sofern es sich um eine eigenständige Sprache handelt. Hierdurch kann später eine Theorie mit Kontextbedingungen generiert werden, die die notwendigen Kontextbedingungen (aus der Funktion `wellformed-base`) sowie alle ausgewählten Varianten der Kontextbedingungen umfasst.

Kontextbedingungen müssen nicht zwangsläufig auch in Isabelle formal repräsentiert werden. Wenn möglich ist eine syntaktische Überprüfung nach dem Parsen des Modells auf Basis einer Symboltabelle vorzuziehen. In bestimmten Fällen ist jedoch eine Kodierung der Kontextbedingungen in Isabelle sinnvoll. Folgend sind Verwendungsmöglichkeiten der kodierten Kontextbedingungen (als Prädikat wellformed) in Isabelle/HOL aufgelistet.

- Für ein konkretes Modell m kann manuell bewiesen werden, dass es wohlgeformt ist, indem wellformed m nachgewiesen wird.

- Anstatt den Beweis tatsächlich durchzuführen, kann per Axiom die Gültigkeit behauptet werden. Das ist dann sinnvoll, wenn anderweitig sichergestellt wird (zum Beispiel beim Aufbau und Prüfung einer Symboltabelle), dass die angegebenen Bedingungen erfüllt sind.

CDSimpCC-base

Isabelle-Theorie

```
1   theory CDSimpCC-base
2   imports "$EXT/gen/running1/CDSimpAS"
3           "$EXT/def/running1/syntax/ExternalCDSimpCC"
4   begin
5
6   fun nameOf :: "CDClass ⇒ Name"
7   where
8    "nameOf (CDClass n l1 l2) = n"
9
10  fun classesOf :: "CDElem list ⇒ CDClass set"
11  where
12   "classesOf elems = { cl | cl . ∃ i . elems!i = CDElemCDClass cl }"
13
14  fun cNameUnique :: "CDClass set ⇒ bool"
15  where
16   "cNameUnique classes =
17      (∀ c1 ∈ classes . ∀ c2 ∈ classes . nameOf c1 = nameOf c2 ⟶ c1 = c2)"
18
19  fun wellformed-base :: "CDDia ⇒ bool"
20  where
21   "wellformed-base (CDDia n invars elems) = (
22      (∀ i ∈ set invars . wellformed-Invariant  i) ∧
23      (cNameUnique (classesOf elems))
24    )"
25
26  end
```

Abbildung 6.26: Kodierung einer Kontextbedingung in Isabelle/HOL

- wellformed kann als zusätzliche Annahme in Beweisen verwendet werden, wenn sich hieraus nützliche Eigenschaften entnehmen lassen. Dabei ist bei Bezug auf konkrete Modelle darauf zu achten, dass die Wohlgeformtheit tatsächlich gegeben ist, da sonst beliebige Aussagen beweisbar sind. Dies trifft auch für die als Axiom definierte Wohlgeformtheit zu.
- Bestimmte Wohlgeformtheitseigenschaften lassen sich nicht automatisiert während einer syntaktischen Prüfung feststellen. Als Beispiel sei hier die Kontextbedingung in UML-StateMachines erwähnt, nach der immer mindestens eine ausgehende Transition von transienten Zuständen bereit (das heißt ihre Vorbedingung erfüllt) sein muss [OMG09b, Abschnitt 15.3.8]. Erfüllbarkeit von allgemeinen Bedingungen ist nicht entscheidbar, kann aber gegebenenfalls manuell bewiesen werden. Da für den Beweis die Semantik der Vorbedingungen herangezogen werden muss, wäre es allerdings sinnvoller, wenn der UML-Standard die Bedingung nicht als Teil der Wohlgeformtheit, sondern als Teil der Semantik definierte.

Als Beispiel zeigt Abb. 6.26 eine Theorie mit einer Kontextbedingung für die Modellierungssprache `CDSimp` wie sie in der Grammatik aus Definition 2.3 definiert ist. Die Kontextbedingung `cNameUnqique` (Zeile 14) besagt, dass alle Klassennamen diagrammweit eindeutig sein müssen. Andere Kontextbedingungen für diese Sprache werden auf ähnliche Weise kodiert. Die Theorie in Abb. 6.26 importiert nicht nur die abstrakte Syntax der Sprache aus der Theorie

UMLPCC

Isabelle-Theorie

```
theory UMLPCC
imports CDAS ODAS SCAS SDAS
begin

datatype UMLP =
   CD CDDef |OD ODDef
  |SC SCDef |SD SDDef

fun wellformedUMLP :: "UMLP set ⇒ bool"
where
 "wellformedUMLP docs = (
     True (* ∧ Inter-Kontextbedingungen *)
  )"

end
```

Abbildung 6.27: Beispiel zum Aufbau von Inter-Kontextbedingungen

`CDSimpAS`, sondern auch Kontextbedingungen für die eingebetteten Sprachen aus der Theorie `ExternalCDSimpCC`. Je nachdem welcher Kontext für die Überprüfung von Kontextbedingungen für eingebettete Sprachen notwendig ist, sind die Prädikate in der importierten Theorie angegeben und müssen dementsprechend angesprochen werden. In unserem einfachen Beispiel wird das Prädikat `wellformed-Invariant` aus der importierten Theorie für alle Invarianten des Klassendiagramms überprüft (Zeile 22). Sprachvererbung ist im Beispiel nicht gezeigt.

Mit dem beschriebenen Verfahren lassen sich Kontextbedingungen kodieren, die sich auf Elemente der Sprache und eingebetteter Sprachbestandteile beziehen. Neben diesen Intra-Kontextbedingungen existieren noch Kontextbedingungen zwischen unabhängigen Sprachen (Inter-Kontextbedingungen). Für solche Kontextbedingungen ist keine weitere Werkzeugunterstützung implementiert. Es ist jedoch ohne weiteres ein Verfahren möglich, bei dem zunächst eine Integration der Teilsprachen über die Angabe eines Datentyps mit entsprechenden Alternativen für jede Teilsprache erfolgt. Über einer Menge von Modellen dieses Datentyps sind nun Inter-Kontextbedingungen formulierbar. Abb. 6.27 illustriert den prinzipiellen Aufbau von Inter-Kontextbedingungen. Dabei wird vorausgesetzt, dass die einzelnen Sprachen (`CDDef`, `ODDef` usw.) bereits in den importierten Theorien (`CDAS`, `ODAS` etc.) definiert sind.

6.4 Kodierung des Systemmodells

Eine Kodierung des Systemmodells in Isabelle/HOL erlaubt es uns, zusammen mit der expliziten Repräsentation der abstrakten Syntax der betrachteten Modellierungssprache in Isabelle/HOL, semantische Abbildungen später direkt in Isabelle/HOL zu definieren. Hauptziel der Kodierung des Systemmodells ist eine möglichst genaue Umsetzung der Definitionen aus Kapitel 4, die gleichzeitig verständlich und nachvollziehbar ist. Verschiedene Umsetzungsmöglichkeiten sind denkbar. Wir beschreiben den gewählten Ansatz und diskutieren dabei mögliche Alternativen.

- Der Aufbau der Theorien in Isabelle (siehe Abb. 6.28) entspricht genau dem Aufbau der

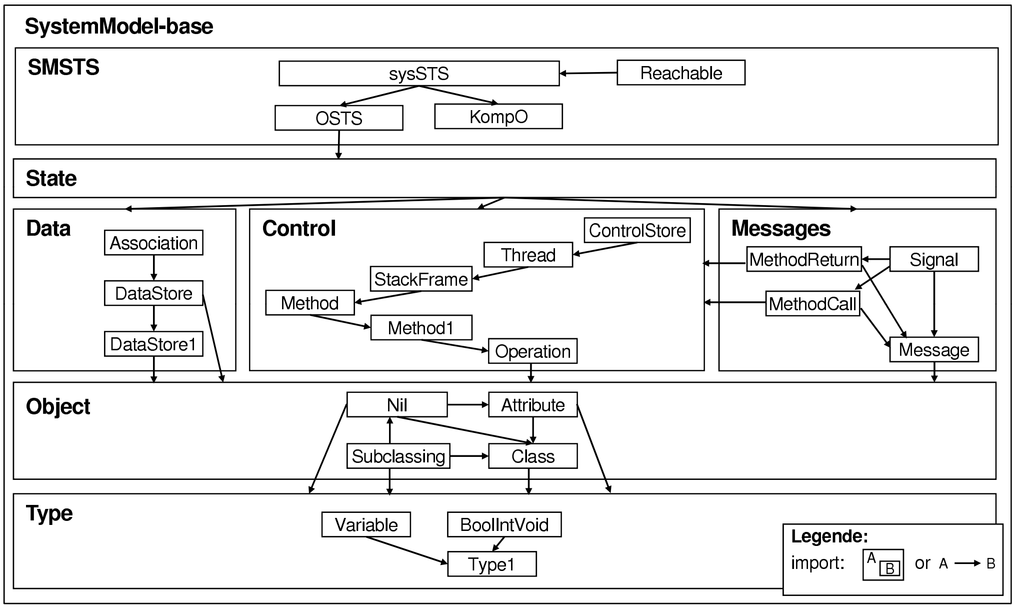

SystemModel-base
SMSTS
sysSTS
Reachable
OSTS
KompO
State
Data
Association
DataStore
DataStore1
Control
ControlStore
Thread
StackFrame
Method
Method1
Operation
Messages
MethodReturn
Signal
MethodCall
Message
Object
Nil
Attribute
Subclassing
Class
Type
Variable
BoolIntVoid
Type1
Legende:
import:
A
B
or A → B

Abbildung 6.28: Schichtenbild der Systemmodelltheorien in Isabelle/HOL

Definitionen aus Kapitel 4 (vgl. Abb. 4.1). Der Unterschied ist, dass in Isabelle nur ein einfacher import-Mechanismus zur Verfügung steht und die Unterscheidung zwischen use- und extend-Beziehungen zwischen Theorien nicht möglich ist. Die Umsetzung enthält alle Definitionen und Varianten. Zusätzlich sind in der Isabelle-Kodierung noch einige Hilfsfunktionen enthalten. Die vollständigen Isabelle-Theorien sind in Anhang B gelistet.

- Die Definitionen in Kapitel 4 sind „aus Sicht" eines einzelnen Systems formuliert. So ist zum Beispiel das Universum UTYPE als Menge von Typnamen spezifisch für ein System. Dies wird zu Beginn der Definitionen einmalig erläutert und gilt entsprechend für alle folgenden Definitionen. In Isabelle/HOL muss dieser Kontext bzw. diese Abhängigkeit von UTYPE von einem konkreten System explizit dargestellt werden. In der Kodierung wird dies so gelöst, dass ein Typ `SystemModel` deklariert wird und dann alle Bestandteile eines Systems als Selektorfunktionen für ein konkretes System (ein Element des Typs `SystemModel`) eingeführt werden. Wie in Abb. 6.29 gezeigt, ist daher `UTYPE` beispielsweise eine Funktion, die für ein System eine Menge von Typnamen (`iTYPE set`) liefert.

  Eine Alternative hierzu ist die Kodierung des Systemmodells direkt als Tupel, indem alle Bestandteile aufgezählt werden. Dadurch würde aber keine einfach erweiterbare Definition erreicht werden. In Isabelle/HOL besteht weiterhin die Möglichkeit, Records zur Definition des Systemmodells zu verwenden. Hierbei treten allerdings ebenfalls Probleme bei der Erweiterbarkeit in Bezug auf Varianten auf [Rin08].

- Die Definitionen im Systemmodell sind meist nicht konstruktiv. Entsprechend werden Elemente in Isabelle ebenfalls nur deklariert. Eigenschaften der Elemente werden dann

Type1

Isabelle-Theorie

```
1   theory Type1
2   imports Base
3   begin
4
5   consts
6    UTYPE :: "SystemModel ⇒ iTYPE set"
7    UVAL  :: "SystemModel ⇒ iVAL set"
8    CAR :: "SystemModel ⇒ (iTYPE ⇒ iVAL set)"
9
10  fun pCARNotEmpty :: "SystemModel ⇒ bool"
11  where
12   "pCARNotEmpty sm = (∀ u ∈ UTYPE sm . CAR sm u ≠ {})"
13
14  fun valid-Type1 :: "SystemModel ⇒ bool"
15  where
16   "valid-Type1 sm = pCARNotEmpty sm"
17
18  end
```

Abbildung 6.29: Die kodierte Theorie Type1 des Systemmodells

mit Prädikaten über dem Systemmodell angegeben. `pCARNotEmpty` (Abb. 6.29, Zeile 10) beispielsweise fordert, dass für alle Typnamen eine nichtleere Trägermenge existiert. Jede Isabelle-Theorie definiert zusätzlich ein Prädikat `valid-`*NameDerTheorie*, in dem alle Eigenschaften zusammengeführt werden (im Beispiel gibt es nur die eine Eigenschaft). Nachdem alle Theorien kodiert sind, werden alle Eigenschaften, die ein System im Systemmodell besitzen soll, in der Theorie `SystemModel-base` zum Prädikat `valid-base` gebündelt (siehe Abschnitt B.9 auf Seite 275).

- HOL ist eine getypte Logik. Das bedeutet, dass zum Beispiel der Menge von Typnamen, die mit `UTYPE` selektiert werden kann, ein HOL-Typ zugewiesen werden muss. Das führt dazu, dass wir eine „maximale Menge“ von Typnamen definieren, aus der Typnamen für `UTYPE` entnommen werden können. Dieser HOL-Typ ist `iTYPE` (vgl. Theorie `Base` in Abschnitt B.1, Seite 245) und umfasst zunächst alle im Systemmodell vorkommenden Typnamen (auch die Typnamen aus optionalen Varianten). Der Typ ist eingeschränkt erweiterbar durch den Typkonstruktor `TExt Name "iTYPE list"`, womit neue Typnamen eingeführt werden können, die auf existierenden Typnamen aufbauen. Ist dieser Konstruktor nicht ausreichend, muss die Definition von `iTYPE` direkt verändert werden. Das Prinzip der Einführung eines „maximalen Universums“ wird nicht nur für `UTYPE`, sondern für alle Universen (z.B. `UVAL`) angewendet. Es ist klar, dass `iTYPE` selbst nicht als `UTYPE` verwendet werden kann, da dann alle Systeme dieselben Typnamen besäßen und `iTYPE` auch keine konsistente Menge von Typnamen definiert. Zum Beispiel sind in `iTYPE` über den Konstruktor `TExt` auch Typnamen mit gleichen Namen aber unterschiedlichen Argumenten enthalten.

  Da das beschriebene Phänomen auf alle Universen und Mengen zutrifft, werden in der Theorie `Base` jeweils ihre maximal möglichen Ausprägungen eingeführt und zur Unterscheidung von den eigentlichen Universen mit dem Präfix `i` gekennzeichnet. Daneben enthält die Theorie `Base` noch rudimentäre Kodierungen eines Stack und eines Puffers.

- Die Verwendung von HOL führt weiterhin dazu, dass in Funktionsdefinitionen ebenfalls der `i`-Typ verwendet werden muss. Zum Beispiel ist die Signatur der Funktion zur Bestimmung des Typs einer Variable

  `vtype :: SystemModel ⇒ (iVAR ⇒ iTYPE)`

  Statt `iVAR` und `iTYPE` wären hier sicherlich `UVAR` und `UTYPE` wünschenswert. Da es sich dabei aber um konkrete Mengen und nicht Typen handelt, ist dies in HOL nicht gestattet. Zudem müssen einige Nebenbedingungen kodiert werden, die in der Formalisierung in Kapitel 4 nicht notwendig sind. Es ist nämlich beispielsweise sicherzustellen, dass `vtype` für Variablen aus `UVAR` auch Typnamen aus `UTYPE` liefert. Diese Bedingung ist in der Theorie `Variable` als Prädikat `pUVAR` kodiert (Definition B.5, Seite 248). Solche Nebenbedingungen könnten vermieden werden, wenn Universen direkt als Isabelle/HOL-Typ kodiert würden. Da dieser Typ jeweils abhängig von einer Instanz des Typs `SystemModel` ist, kommen hier als Kodierung nur Typparameter für den Typ `SystemModel` in Frage. Dies führt allerdings zu einer sehr schwer lesbaren und schlecht erweiterbaren Kodierung des Systemmodells, da alle Typparameter immer explizit angegeben werden müssten und Erweiterungen um einen zusätzlichen Typparameter sich auf alle Verwendungsstellen von `SystemModel` auswirkten. Anstatt HOL könnte auch eine ungetypte Logik verwendet werden. Allerdings ist die Infrastruktur (vorhandene Theorien, Beweisverfahren) in Isabelle nur für die Logik HOL sehr gut ausgebaut.

- Die im Systemmodell enthaltene Unterspezifikation kann genau nachgebildet werden. Universen und andere unterspezifizierte Elemente des Systemmodells werden deklariert und einige ihrer Eigenschaften in nachfolgenden Prädikaten charakterisiert. Einige exakt spezifizierte Elemente des Systemmodells werden ebenso als konkrete Funktionsdefinitionen kodiert. So ist beispielsweise die Menge der erlaubten Datenzustände genau festgelegt. Die Definition ist in der Theorie `DataStore1` in Definition B.18 angegeben. Die Funktion `DataStore` liefert für ein System die Menge aller möglichen Datenzustände, nämlich alle die Elemente des Typs `iDataStore`, die die angegebenen Bedingungen erfüllen.

- Die mathematische Notation in Kapitel 4 ist insgesamt etwas flexibler. Die Kodierung in Isabelle wirkt daher zum Teil etwas umständlich. Auf eine weitere syntaktische Anpassung der Isabelle-Kodierung, die durch Syntaxannotationen [NPW02] möglich wäre, wurde verzichtet. Der Stil der Isabelle-Kodierung entspricht im Wesentlichen funktionaler Programmierung, die mit logischen Ausdrücken durchmischt ist.

6.5 Kodierung semantischer Abbildungen

Die abstrakte Syntax einer Sprache und die Definitionen des Systemmodells liegen in Form von Isabelle-Theorien vor. Die semantische Abbildung kann also vollständig in Isabelle/HOL kodiert werden. Abb. 6.30 zeigt das Zusammenspiel der Theorien für eine Sprache `L1`.

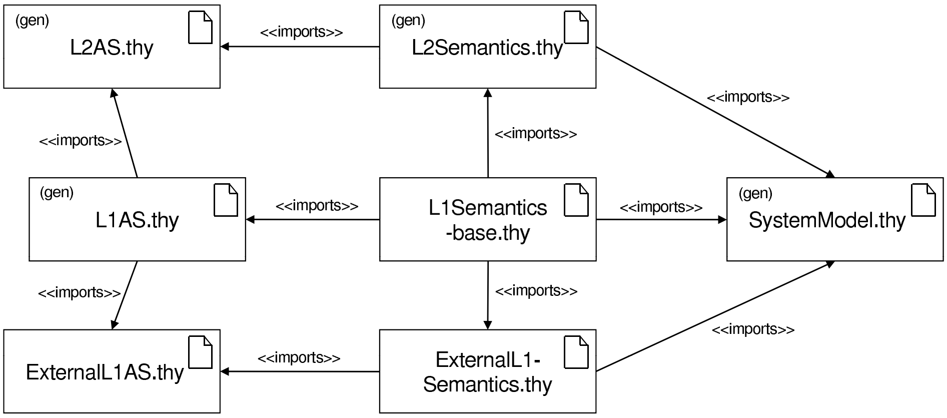


Abbildung 6.30: Theorie `L1Semantics-base` für die semantische Abbildung und ihre Abhängigkeiten zu anderen Theorien

- `L1AS.thy` wurde automatisch aus einer MontiCore-Grammatik erzeugt.
- Die semantische Abbildung einer Obersprache `L2` kann importiert werden. Sie wurde im Zuge der Definition und Konfiguration der Sprache `L2` erstellt. `L2Semantics` beinhaltet bereits die konfigurierte semantische Abbildung einer Sprachvariante von `L2` (vgl. Abschnitt 6.6).
- Die semantischen Abbildungen für Sprachparameter sind manuell in die Isabelle-Theorie `ExternalL1Semantics` zu importieren. Die Theorie muss selbst die abstrakte Syntax der Sprachparameter `ExternalL1AS` importieren und ist damit automatisch konform zur Konfiguration der Einbettung durch die Sprachdatei `L1.lng`.
- Die Theorie des Systemmodells wird importiert. Es handelt sich um ein bereits gemäß Abschnitt 6.6 konfiguriertes Systemmodell, in dem beispielsweise die für die semantische Abbildung benötigen Varianten enthalten sind. Die Theorie wird auch von allen anderen gleichzeitig verwendeten Sprachen importiert.
- Auf Basis der wiederverwendbaren semantischen Abbildungen von Obergrammatik und eingebetteten Sprachen wird in Theorie `L1Semantisc-base` die semantische Abbildung für `L1` kodiert.

Ein gekürztes Beispiel, das nur die Signatur der einzelnen Funktionen der semantischen Abbildung für die Sprache `CDSimp` enthält, geben wir in Abb. 6.31 an. Es empfiehlt sich, soweit möglich, die Definition der semantischen Abbildung entlang der abstrakten Syntax vorzunehmen, um so einen modularen Aufbau der Abbildung zu erreichen. Ist für die Semantik eines Konstrukts keine Variabilität vorgesehen, ist eine direkte Funktionsdefinition (alternativ, eine induktive Definition) sinnvoll. Sollen dagegen unterschiedliche Varianten der Abbildung erlaubt werden, wird nur eine Deklaration benötigt. In unserem Beispiel ist dies der Fall für

CDSimpSemantics-base

Isabelle-Theorie

```
theory CDSimpSemantics-base
imports "$EXT/gen/running1/CDSimpAS"
        "$EXT/def/running1/semantics/ExternalCDSimpSemantics"
        "$EXT/gen/SystemModel"
begin

consts mSuperClasses :: "Name list ⇒ iCLASS ⇒ SystemModel ⇒ bool"

fun mCDAttr :: "CDAttr ⇒ iCLASS ⇒ SystemModel ⇒ bool"
where ..

fun mCDAssoc :: "CDAssoc ⇒ SystemModel ⇒ bool"
where ..

fun mCDClass :: "CDClass ⇒ SystemModel ⇒ bool"
where ..

fun mCDInvariant :: "Invariant ⇒ SystemModel ⇒ bool"
where ..

fun mCDElem :: "CDElem ⇒ SystemModel ⇒ bool"
where ..

fun mCDDia :: "CDDia ⇒ SystemModel ⇒ bool"
where ..

end
```

Abbildung 6.31: Kodierung der semantischen Abbildung in Isabelle/HOL

`mSuperClasses` zur semantischen Abbildung der Vererbung im Klassendiagramm. Alle anderen Teile der Abbildung sind Funktionsdefinitionen ohne Variabilität.

Die Semantik für eingebettete Sprachteile wird ebenfalls importiert. Da die Einbettung variabel ist, wird nicht eine konkrete Sprache importiert, sondern die Konfiguration der Einbettung, die im Beispiel in der Theorie `ExternalCDSimpSemantics` festgelegt ist. Diese Theorie ist in unserem Beispiel also dafür verantwortlich, eine gemäß der Einbettung passende Semantik für Invarianten anzubieten. Hierbei muss sichergestellt werden, dass neben einer kompatiblen Signatur auch der notwendige Kontext von der äußeren Sprache bereitgestellt werden kann (z.B. Variablenbelegungen, Stereotypwerte, o.ä.). Ausführliche Beispiele finden sich in Kapitel 7.

6.6 Definition und Konfiguration von Variabilität

Jegliche Variabilität einer Modellierungssprache (vgl. Kapitel 3) lässt sich zunächst in textuellen Feature-Diagrammen dokumentieren, die analog zu der Vorlage in Abb. 5.10 auf Seite 95 strukturiert sind. Im Zuge dieser Arbeit wurde das Werkzeug ThyConfig implementiert, mit dem sich textuelle Konfigurationen automatisiert auf Konformität zu den Feature-Diagrammen prüfen lassen. Für Varianten, die als mathematische Theorien in Isabelle kodiert werden, generiert ThyConfig zusätzlich jeweils eine integrierte Theorie, die die Basisdefinitionen und die gewählten Varianten umfasst. Dies betrifft Varianten von Kontextbedingungen (aus Abkürzungen oder Spracheinschränkungen), des Systemmodells und semantischer Abbildungen. Für andere Formen der Variabilität (Präsentationsoptionen, Stereotypen, Sprachparameter und syntaktische Spracherweiterungen) ist keine automatische Konfiguration der Sprachen vorgesehen.

6.6.1 Präsentationsvariabilität

Varianten von Präsentationsoptionen

Durch die Verwendung eines textuellen Ansatzes mit MontiCore und der automatisch aus der konkreten Syntax abgeleiteten abstrakten Syntax sind Präsentationsoptionen nur sehr begrenzt definierbar. Es ist möglich, verschiedene Terminalsymbole zur Darstellung desselben Sachverhalts (beispielsweise `public` oder + für einen Modifikator, der den öffentlichen Zugriff anzeigt) einzuführen. Terminalsymbole können auch ganz entfernt oder neu eingeführt werden, ohne die abstrakte Syntax zu beeinflussen. Lexikalische Produktionen können ebenfalls verändert werden, solange der Rückgabetyp gleich bleibt. Bei derartigen Änderungen ist aber darauf zu achten, die Parsefähigkeit der Sprache nicht zu verlieren oder die Ausdrucksmächtigkeit der Sprache nicht einzuschränken. Vielseitigere Präsentationsoptionen sind für graphische Notationen möglich. Der UML-Standard definiert beispielsweise viele dieser Varianten explizit.

Die notwendigen strukturellen Veränderungen einer Grammatik bei Auswahl einer Präsentationsoption (und später auch bei Auswahl einer Abkürzung oder syntaktischen Spracherweiterung) sind nicht Fokus dieser Arbeit. Automatisierte Grammatikanpassungen in MontiCore werden beispielsweise in [Tor06] untersucht.

Varianten von Abkürzungen

Varianten von Abkürzungen sind im Feature-Diagramm unter einem Knoten mit dem Stereotyp `<<abb>>` dokumentiert. Wir setzen voraus, dass für jede im Feature-Diagramm aufgeführte Abkürzung eine Theorie mit demselben Namen implementiert ist, die das Vorhandensein der Abkürzung in einem Modell durch eine entsprechende Kontextbedingung ausschließt. Wir nehmen also an, dass die Abkürzung bereits durch eine geeignete syntaktische Transformation des Modells in andere Elemente übersetzt wurde und prüfen dies anhand der Kontextbedingung. Die syntaktische Transformation selbst ist nicht Fokus dieser Arbeit.

Abb. 6.32 zeigt beispielhaft ein Feature-Diagramm mit der syntaktischen Erweiterung bzw. Abkürzung `Hierarchy` für Statecharts, die die hierarchische Schachtelung von Zuständen erlaubt. Weiter Abkürzungen und die zugehörigen Transformationen werden ausführlich zum Beispiel in [CGR08a, Rum04b] dargestellt. Die konforme Konfiguration dieses Teils der Sprache im zweiten Teil der Abbildung führt nach Anwendung von ThyConfig zur generierten Theorie in Abb. 6.33. Per Namenskonvention legen wir also fest, dass alle Varianten von Abkürzungen ein Prädikat des Namens `wellformed-`*NameDerVariante* definieren. Hierdurch lässt sich in der generierten Theorie das Prädikat `wellformed` erzeugen, das alle Varianten miteinander verknüpft.

Zur syntaktischen Transformationen von Modellen kann MontiCore-TF [Nun09], ein Transformationsframework, verwendet werden. Die Semantik der hierbei transformierten Konstrukte

SCPresentation

Feature-Diagramm

```
package running1.presentation;

featurediagram SCPresentation {

   SCPresentation = <<abb>> SCAbbreviation;

   SCAbbreviation = Hierarchy;
}
```

SCPresentation1

Konfiguration

```
package running1.presentation;

config SCPresentation1 for running1.presentation.SCPresentation {

  Hierarchy

}
```

Abbildung 6.32: Feature-Diagramm und Konfiguration für Statechart-Abkürzungen

SCAbbreviation

Isabelle-Theorie (gen)

```
theory SCAbbreviation
imports "$EXT/def/running1/presentation/Hierarchy"

begin
fun wellformed where
 "wellformed m = (wellformed-Hierarchy m)"
end
```

Abbildung 6.33: Generierte Theorie mit Abkürzungen

ist somit über diese in Java implementierten Transformationsregeln festgelegt. Anschließend wird für die vereinfachte Sprache die Semantik auf Basis des Systemmodells wie beschrieben in Isabelle/HOL angegeben. Als Implementierungsgrundlage für die Transformationsregeln kann vorher eine Formalisierung erfolgen. Hierfür schlagen wir das Transformationsschema aus [CGR08b] vor. In [CGR08b] ist auch gezeigt, wie sich komplexe Transformationen, nämlich die Vereinfachung von Statecharts gemäß [Rum04b], beschreiben lassen. Die dort eingeführten Regeln wurden später auch in MontiCore-TF-Regeln überführt [Sch08].

Das Verfahren zur syntaktischen Transformation ist in Abb. 6.34 skizziert. Findet keine strukturelle Veränderung der Sprache statt, sind beide MontiCore-Grammatiken identisch und die Transformation erfolgt *in-place*.

Ist es gewünscht, dass Transformationsregeln auch mathematisch präzise in maschinenlesbarer Form vorliegen, ist auch ein erweitertes Vorgehen möglich. Die komplexe Sprache und die vereinfachte Sprache werden in Isabelle/HOL-Datentypen übersetzt und die Transformationsregeln als Relation oder Funktion zwischen diesen Datentypen direkt in Isabelle/HOL angegeben.

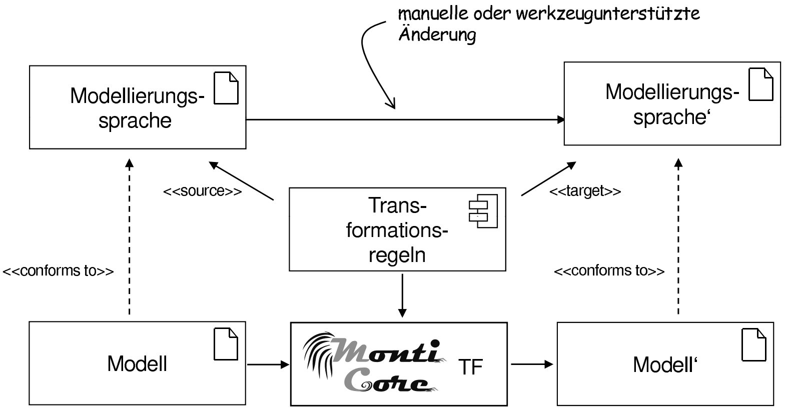


Abbildung 6.34: Syntaktische Transformation für eine einfachere Semantikdefinition

So sind auch die Eigenschaften der Transformation weiteren Analysen in Isabelle zugänglich. Sind die komplexe und vereinfachte abstrakte Syntax in Isabelle/HOL vorhanden, bietet sich sogar die Möglichkeit der Verifikation der Korrektheit von Transformationen. Es besteht die Möglichkeit zu prüfen, ob die angegebenen Transformationsregeln tatsächlich semantikerhaltend sind. Voraussetzung hierfür ist allerdings, dass für beide Versionen der Sprache die semantische Abbildung definiert worden ist. Der ursprüngliche Gedanke bei der syntaktischen Transformation ist allerdings, eine vereinfachte semantische Abbildung zu erreichen. Beide Ziele, Korrektheitsbeweise der Transformationen in Isabelle und einfache semantische Abbildung, sind unvereinbar.

6.6.2 Syntaktische Variabilität

Gemäß unserer allgemeinen Klassifikation aus Kapitel 3 betrachten wir Varianten für Sprachparameter, Spracheinschränkungen und Stereotypen. Erfasst wird syntaktische Variabilität in einem textuellen Feature-Diagramm wie es exemplarisch in Abb. 6.35 für vereinfachte Klassendiagramme gezeigt ist. Über den Stereotyp `<<lparam>>` wird angezeigt, dass sich der nachfolgende Variationspunkt auf Sprachparameter bezieht, analog werden `<<constr>>` und `<<stereos>>` verwendet, um Variationspunkte für spracheinschränkende Kontextbedingungen und Stereotypen zu kennzeichnen.

Es ist zu beachten, dass Feature-Diagramme beliebig modularisiert werden können. Im Beispiel sind alle syntaktischen Variationspunkte in einem Diagramm zusammengefasst. Sie lassen sich aber ebenso gut in einzelne Feature-Diagramme aufteilen. Das Beispiel zeigt nur einen Teil der Variabilität. Zusätzlich kann das Feature-Diagramm auch semantische Varianten oder Präsentationsvarianten enthalten. Wir können also die gesamte Variabilität einer Sprache in einem Feature-Diagramm modellieren oder jeden Variationspunkt in einem eigenen Feature-Diagramm separat beschreiben.

CDSimpSyntax

Feature-Diagramm

```
1   package running1.syntax;
2
3   featurediagram CDSimpSyntax {
4
5      CDSimpSyntax = <<lparam>> LangParam? &
6                     <<constr>> CDSimpConstr? &
7                     <<stereos>> Stereos?;
8
9      LangParam = Invariant?;
10
11     Invariant = OCLInvariant? ^ JavaExpression?;
12
13     Stereos = StandardStereoSet? ^ CustomStereoSet?;
14
15     CDSimpConstr = CCSingleInheritance? & CCAttrTypeDefined?;
16  }
```

Abbildung 6.35: Feature-Diagramm zur Definition syntaktischer Variabilität

Stereotypen

Stereotypen werden im Feature-Diagramm zu Dokumentationszwecken erfasst, damit wir eine vollständige Beschreibung der Varianten der Sprache erhalten. Eine weitere Generierung, beispielsweise von Code zur Überprüfung, ob ein Modell tatsächlich nur die Stereotypen einer ausgewählten Variante enthält, findet nicht statt. Wir gehen davon aus, dass ähnlich der Darstellung in [Rum04b] eine Beschreibung der Stereotypen für eine Modellierungssprache vorliegt. Diese Beschreibung enthält Informationen darüber, auf welche Modellelemente sich Stereotypen beziehen, warum der Stereotyp eingeführt wird, welche Rahmenbedingungen existieren und welche Wirkung der Stereotyp erzielen soll. Für eine Modellierungssprache wird dann festgelegt, welche Menge von Stereotypen verwendet werden darf. Diese Menge kann je nach Einsatzzweck der Modellierungssprache variiert werden. So entstehen Varianten in Stereotypen. Im Beispiel in Abb. 6.35 können die (fiktiven) Stereotypmengen `StandardStereoSet` und `CustomStereoSet` exklusiv verwendet werden (Zeile 13).

Für eine Konfiguration der Stereotypen wird durch das Werkzeug ThyConfig geprüft, ob die Auswahl konform zum Feature-Diagramm erfolgt ist. Häufig resultieren aus der Auswahl einer Menge von Stereotypen weitere Abhängigkeiten, da eine entsprechende Variante der semantischen Abbildung auszuwählen ist, die die Stereotypen semantisch behandelt.

Sprachparameter

Die Verwendung von Sprachparametern durch Einbettung, wie sie in MontiCore-Grammatiken erlaubt ist, wurde bereits im Abschnitt 6.1.4 ausführlich erläutert. Ebenso wie die Variabilität in Stereotypen dient die Definition der Varianten für Sprachparameter eher Dokumentationszwecken, um eine vollständige Sprachdefinition zu erreichen. Der mit dem Stereotyp `<<lparam>>` gekennzeichnete Variationspunkt zeigt, welche Sprachen für welche externen Produktionen eingebettet werden können. Im Beispiel in Abb. 6.35 ist `Invariant` (Zeile 11)

CDSimpConstr

Isabelle-Theorie (gen)

```
theory CDSimpConstr
imports "$EXT/def/running1/syntax/CCSingleInheritance"
        "$EXT/def/running1/syntax/CCAttrTypeDefined"

begin
fun wellformed where
 "wellformed m = (wellformed-CCSingleInheritance m
              ∧ wellformed-CCAttrTypeDefined m)"
end
```

Abbildung 6.36: Generierte Theorie mit Kontextbedingungen für Spracheinschränkungen

die einzige definierte Einbettung. Hierfür können Ausdrücke in OCL (`OCLInvariant`) bzw. Java (`JavaExpression`) eingesetzt werden.

Für eine Konfiguration der Sprachparameter überprüft ThyConfig, ob die Auswahl konform zum Feature-Diagramm erfolgt ist. Eine weitergehende Werkzeugunterstützung bei der Umsetzung der Konfiguration erfolgt nicht. Der Grund hierfür ist, dass bereits mit MontiCore-Sprachdateien eine Möglichkeit besteht, Parser für die Einbettung zu konfigurieren und damit die Auswahl der Sprachparameter festzulegen. Die Überprüfung, ob eine Konfiguration korrekt umgesetzt ist, muss manuell anhand der MontiCore-Sprachdateien nachvollzogen werden. Für zukünftige Versionen wäre aber denkbar, die Sprachdateien durch eine erweiterte Form der Feature-Diagramme und Konfigurationen mit dem Ziel zu ersetzen, weitere Automatisierung bezüglich der Sprachkonfiguration zu erreichen.

Spracheinschränkungen

Spracheinschränkungen als Verallgemeinerung von Abkürzungen sind technisch ebenfalls als strukturelle Sprachveränderungen oder zusätzliche Kontextbedingungen realisierbar. Wir verwenden in dieser Arbeit nur die zweite Möglichkeit und den Stereotyp `<<constr>>`, um Varianten von Spracheinschränkungen im Feature-Diagramm zu kennzeichnen. Das Feature-Diagramm in Abb. 6.35 enthält hierzu zwei optionale Varianten hinter denen in Isabelle/HOL kodierte Theorien stehen. Mit Hilfe des Werkzeugs ThyConfig wird aus einer Konfiguration, in der beide Varianten ausgewählt werden, die Theorie `CDSimpConstr` in Abb. 6.36 erzeugt. Per Namenskonvention müssen alle Varianten ein Prädikat `wellformed-`*NameDerVariante* definieren. Hierdurch lässt sich in der generierten Theorie das Prädikat `wellformed` erzeugen, das alle Varianten miteinander verknüpft.

Die Gesamtheit aller Kontextbedingungen ergibt sich aus Kontextbedingungen für Abkürzungen, Spracheinschränkungen und den notwendigen Kontextbedingungen. Hierfür generiert ThyConfig für eine Sprache, in diesem Fall `CDSimp`, eine Theorie, die alle Theorien mit Kontextbedingungen importiert, so dass Kontextbedingungen einfach verwendet werden können, siehe Abb. 6.37. Abkürzungen sind in diesem Beispiel nicht verwendet. Die Generierung des zusammenfassenden Prädikats `wellformed` ist sowohl für Spracheinschränkungen als auch für die integrierte Theorie standardmäßig unterdrückt, da es für Sprachen, die nur eine lose Sammlung von Konstrukten ohne definierte Startproduktion nicht sinnvoll generiert werden kann. Die

CDSimpCC

Isabelle-Theorie (gen)

```
theory CDSimpCC
imports "$EXT/def/running1/syntax/CDSimpCC-base"
        "$EXT/gen/running1/syntax/CDSimpConstr"
begin
fun wellformed where
 "wellformed m = (wellformed-base m  ∧ CDSimpConstr.wellformed  m)"
end
```

Abbildung 6.37: Generierte Theorie mit allen Kontextbedingungen für eine Sprachkonfiguration

Generierung wird durch zusätzliche Angabe des Stereotyps `<<cu>>` (*compilation unit*) für eigenständige Sprachen aktiviert.

6.6.3 Variabilität des Systemmodells

Alle in Kapitel 4 eingeführten Varianten des Systemmodells sind in Isabelle kodiert. Jede Variante ist in einer eigenen Theorie enthalten. In Varianten können zum einen einfache zusätzliche Bedingungen definiert werden, beispielsweise eine Teilmengenbeziehung zwischen Attributen von Klassen in Vererbungsbeziehung oder die Antisymmetrie der Subklassenbeziehung (vgl. Varianten B.13 und B.14, Seite 252). Zum anderen können auch neue Funktionen oder Mengen eingeführt werden, die das Systemmodell strukturell erweitern, wie es zum Beispiel in Variante B.23 (Seite 256) in der Theorie `BinaryAssociation` mit der Definition von `binaryAssoc` oder `binaryRelOf` geschieht.

Ähnlich wie bei Varianten für Kontextbedingungen besteht auch hier eine Namenskonvention. Jede Isabelle-Theorie mit einer Systemmodellvariante enthält ein Prädikat mit dem Namen `valid-`*NameDerVariante*. Alle Variationspunkte und Varianten sind in einem textuellen Feature-Diagramm enthalten, das genau dem Feature-Diagramm in Abb. 4.61 auf Seite 80 entspricht (vgl. Definition B.56, Seite 276). Die Namen der Varianten im Feature-Diagramm korrelieren mit den Namen der Theorien, die die Varianten kodieren.

Die Variabilität einer Sprache beinhaltet die Variabilität der semantischen Domäne, also des Systemmodells. Das vollständige Feature-Diagramm für eine Sprache enthält also eine Referenz auf das Feature-Diagramm des Systemmodells. Wir markieren den Verweis auf die Variabilität der semantischen Domäne mit dem Stereotyp `<<sd>>`, da prinzipiell auch andere semantische Domänen zum Einsatz kommen können. Bei der Konfiguration der Sprache können Varianten des Systemmodells ausgewählt werden. Das Werkzeug ThyConfig generiert hieraus die Theorie `SystemModel.thy`, wobei die Theorie `SystemModel-base.thy` und alle gewählten Varianten importiert werden. Die generierte Theorie enthält das Prädikat `valid`, das aus `valid-base` (aus der Theorie `SystemModel-base`, vgl. Definition B.55 auf Seite 275) und allen Prädikaten der Varianten zusammengesetzt wird. Wir erhalten mit `valid` dann eine Zusammenfassung aller Eigenschaften des konfigurierten Systemmodells. Ein Beispiel ist in Abb. 6.38 dargestellt. Die Theorie entsteht bei Auswahl der Varianten `LiskovPrinciple` und `AntiSymSub`.

Werden mehrere Sprachen konfiguriert, die alle das Systemmodell als semantische Domäne

SystemModel

Isabelle-Theorie (gen)

```
theory SystemModel
imports "$SYSTEMMODEL/SystemModel-base"
        "$SYSTEMMODEL/vObject/LiskovPrinciple"
        "$SYSTEMMODEL/vObject/AntiSymSub"

begin
constdefs "valid sm == valid-base sm
                  ∧ valid-LiskovPrinciple sm
                  ∧ valid-AntiSymSub sm"
end
```

Abbildung 6.38: Generierte Theorie des Systemmodells

referenzieren, entsteht trotzdem nur eine gemeinsam genutzte Systemmodell-Theorie, die aus der Überlagerung aller einzelnen Konfigurationen resultiert. Nach der Überlagerung wird die Konformität der Gesamtkonfiguration zum Feature-Diagramm durch ThyConfig geprüft, bevor die Theorie `SystemModel.thy` generiert wird.

6.6.4 Variabilität semantischer Abbildungen

Varianten einer semantischen Abbildung sind meist alternative Realisierungen eines Teils der Abbildung. Alle in der Kodierung als `consts` deklarierten Teile lassen Variantenbildung zu. Soll eine Variante für eine fest definierte Funktion im Sinne einer alternativen Realisierung gebildet werden, muss zunächst die Definition in eine Deklaration umgewandelt und beide Varianten zur Verfügung gestellt werden. Jede Variante der semantischen Abbildung wird in einer separaten Theorie kodiert. Als Beispiel nehmen wir an, dass im Zuge der Definition der Semantik von Klassendiagrammen eine Theorie `MapSuperClassDirect` erstellt wurde, die die Funktion `mapSuperClassDirect` enthält, um Mehrfachvererbung semantisch abzubilden. Eine offen gelassene Definition wie die entsprechende Deklaration `mSuperClass` aus Abb. 6.31 von Seite 134 wird mit der Definition der Variante `mapSuperClassDirect` versehen, indem die Variante die Zeile

```
defs mSuperClass-def : "mSuperClass == mSuperClassDirect"
```

enthält.

Wie üblich werden Varianten in einem Feature-Diagramm modelliert. Mit dem Stereotyp `<<sm>>` werden Variationspunkte der semantischen Abbildung gekennzeichnet. Aus einer zum Feature-Diagramm konformen Konfiguration der semantischen Abbildung generiert ThyConfig eine Theorie, die zum einen die Basisversion der Abbildung und zum anderen alle ausgewählten Varianten importiert. Wird die generierte Theorie mit der Semantik der Sprache geladen, werden durch den Isabelle import-Mechanismus die Definitionen aus den Varianten automatisch an die Deklarationen der Basisversion gebunden. Zahlreiche Beispiel zur Definition von Varianten semantischer Abbildungen finden sich auch in Kapitel 7.

6.6.5 Zusammenfassung

Das Werkzeug ThyConfig verarbeitet Sprachkonfigurationen und Feature-Diagramme für eine oder mehrere Sprachen. Abb. 6.39 veranschaulicht die Aufgaben. Für eine einzelne Konfiguration wird zunächst die Konformität der ausgewählten Varianten mit dem zugehörigen Feature-Diagramm geprüft. Sprachparameter werden nicht weiter bearbeitet, da die Auswahl manuell mit MontiCore-Sprachdateien abzugleichen ist. Für Stereotypen und Präsentationsoptionen ist ebenfalls nur eine Konformitätsprüfung möglich.

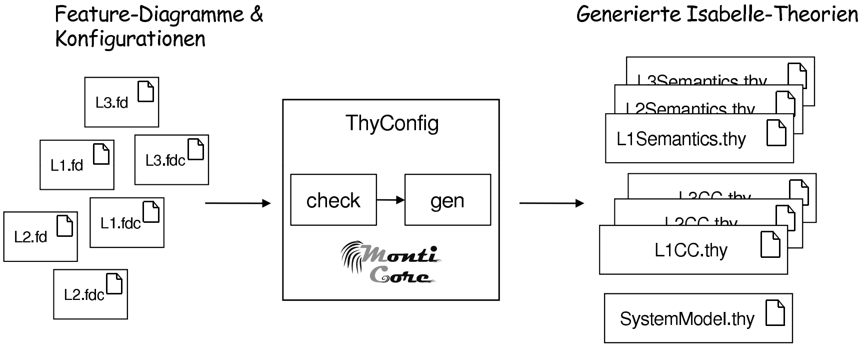

Feature-Diagramme &
Konfigurationen
L3.fd
L1.fd
L3.fdc
L2.fd
L1.fdc
L2.fdc
ThyConfig
check
gen
Monti
Core
Generierte Isabelle-Theorien
L3Semantics.thy
L2Semantics.thy
L1Semantics.thy
L3CC.thy
L2CC.thy
L1CC.thy
SystemModel.thy

Abbildung 6.39: ThyConfig zur Generierung von Isabelle/HOL-Theorien

Aus einer Auswahl von Varianten für Abkürzungen und Spracheinschränkungen für eine Sprache `L1` werden Theorien mit Kontextbedingungen generiert. Es entsteht auch eine Theorie `L1CC` mit sämtlichen Kontextbedingungen zur einfachen Verwendung. ThyConfig generiert weiterhin die Theorie der semantischen Abbildung `L1Semantics` und für alle betrachteten Sprachen eine gemeinsame Theorie des Systemmodells.

Teil III

Fallstudien

Kapitel 7

UML/P

Dieses Kapitel beschäftigt sich mit der Definition der Semantik für UML/P. Eine ausführliche Einführung in die Sprache und ihren methodischen Einsatz in der modellbasierten Softwareentwicklung kann dieses Kapitel nicht leisten. Hierfür wird auf [Rum04a] und [Rum04b] verwiesen.

7.1 Übersicht

Eine vollständige Definition einer Modellierungssprache umfasst die Syntax und die Semantik. Der Fokus dieser Arbeit und speziell dieses Kapitels liegt auf der werkzeugunterstützten Definition der semantischen Abbildung von UML/P in das Systemmodell, wobei auch Varianten berücksichtigt werden. Die Definition der (konkreten) Syntax, der Kontextbedingungen und syntaktischer Varianten spielen daher eine untergeordnete Rolle.

Wir betrachten alle in [Rum04b] eingeführten UML/P-Diagrammarten. Diese sind Klassendiagramme (CD), Objektdiagramme (OD), Statecharts (SC) und Sequenzdiagramme (SD). Zudem werden einige semantische Varianten dieser Sprachen betrachtet, wobei eine Behandlung aller denkbaren Varianten nicht möglich ist, da sie häufig aus einem konkreten Verwendungszweck heraus definiert werden. Zusätzliche syntaktische oder semantische Varianten können aber leicht eingefügt werden.

In [Rum04b] werden auch Java und OCL eingeführt. Eine vollständige Behandlung von Java als Aktionssprache oder OCL als Constraint-Sprache würde den Rahmen der Arbeit sprengen. Wir beschränken uns auf sehr vereinfachte Versionen dieser beiden Sprachen, die zur Einbettung in UML/P-Diagrammarten verwendet werden können. Die semantischen Abbildungen für Java und OCL zeigen, wie prinzipiell Programmier- und Constraint-Sprachen auf das Systemmodell abgebildet werden können. Sie werden gerade so umfangreich definiert, dass mit ihnen einfache, sinnvolle Invarianten oder Anweisungen für die Beispiele in Kapitel 8 angegeben werden können. Abbildung 7.1 zeigt eine Übersicht über die betrachteten Sprachen. Neben den eigenständigen UML/P-Diagrammarten, Java und OCL existieren noch gemeinsam genutzte Sprachteile, die mittels Sprachvererbung verwendet werden. Es handelt sich um Fragmente zur Definition von Literalen, Typen und weiteren gemeinsamen Definitionen (Common).

Es wird das in Abschnitt 5.5 vorgeschlagene und in Kapitel 6 ausführlich beschriebene Verfahren zur werkzeugunterstützten Definition von Modellierungssprachen verwendet. Die hierfür notwendigen MontiCore-Grammatiken und die hieraus generierte abstrakte Syntax in Isabelle/HOL sind in Anhang C angegeben. Ebenfalls in Anhang C sind in Isabelle kodierte Kontext-

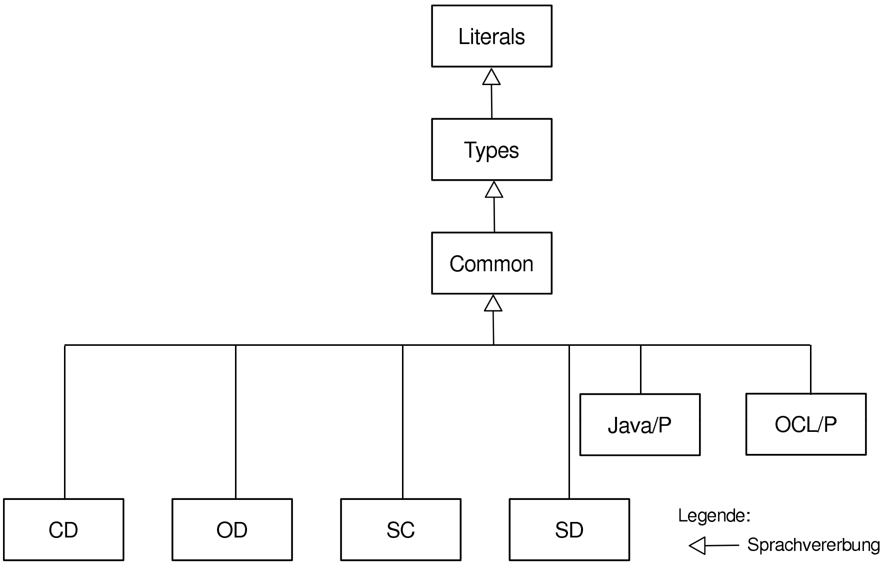


Abbildung 7.1: Übersicht über die betrachteten Sprachen

bedingungen, sowie jeweils ein Feature-Diagramm für die syntaktische Variabilität zu finden. Wir konzentrieren uns in diesem Kapitel daher vor allem auf die semantischen Abbildungen und deren Varianten. Die Isabelle-Theorien werden hierfür ganz oder in Auszügen dargestellt.

Eine MontiCore-Grammatik soll als zentrale Sprachreferenz dienen [Kra10]. Sie ist deshalb relativ ausführlich kommentiert. Bei Verständnisschwierigkeiten der abstrakten Syntax kann daher immer die zugehörige Grammatik konsultiert werden. Die hier verwendeten UML/P-Grammatiken in der Version 1.0a sind nicht im Zuge dieser Arbeit entstanden, sondern weitgehend aus der Version 1.0 aus [Sch09] übernommen. Auf Änderungen bezüglich der Version 1.0 der UML/P-Grammatiken wird ebenfalls in Anhang C hingewiesen. Die rudimentären Grammatiken für Java und OCL sind aus den Grammatiken aus [Pin07](OCL) und [FM07](Java) hervorgegangen.

Der Rest des Kapitels orientiert sich am Aufbau der UML/P und ist wie folgt strukturiert. In Abschnitt 7.2 wird die Semantik der gemeinsamen Sprachteile Literals, Types und Common definiert. In Abschnitt 7.3 betrachten wir die Semantik von Java und OCL. Klassendiagramme werden in Abschnitt 7.4 behandelt. Abschnitt 7.5 enthält eine Semantik für Objektdiagramme während in den Abschnitten 7.6 und 7.7 Semantiken von Statecharts und Sequenzdiagramme definiert werden. In Abschnitt 7.8 zeigen wir beispielhaft wie eine Konfiguration der Sprachen von UML/P durchgeführt werden kann. Verwandte Arbeiten im Bereich der Einzelsemantiken für die UML/P-Diagrammarten werden im jeweiligen Abschnitt kurz umrissen.

7.2 Gemeinsam genutzte Sprachteile

7.2.1 Literals

In allen UML/P-Teilsprachen wird eine gemeinsame Definition von Literalen verwendet. Die zugehörige MontiCore-Grammatik ist in Anhang C.1.1, Definition C.1 ab Seite 277 angegeben. Sie enthält eine Java-kompatible Definition von Literalen für Ganz- und Fließkommazahlen, Wahrheitswerte, Namen, Zeichen und Zeichenketten. Während die MontiCore-Grammatik aufgrund der Lexer-Regeln relativ komplex ist, ist die resultierende abstrakte Syntax in Definition C.2 ab Seite 282 kompakt. Um eine eher aufwendige Kodierung von Fließkommazahlen in Isabelle ohne großen Mehrwert zu vermeiden, verzichten wir auf die Behandlung von Fließkommazahlen. Mit der Kontextbedingung in Variante C.3 gibt es daher die Möglichkeit, die Verwendung von Fließkommazahlen auszuschließen. Die Übersetzung des Typs der Lexer-Regeln für `NUM-FLOAT`, `NUM-DOUBLE` und `NUMBERS` können ignoriert werden.

Die semantische Abbildung ist in Definition 7.2 angegeben. Werte in der abstrakten Syntax werden weitgehend eins zu eins auf semantische Werte abgebildet. So wird beispielsweise ein Integer-Wert `x` (Isabelle-Typ `int`, vgl. Definition C.2) mit der Funktion `mNUM-INT` auf `VInt x` (Zeile 8) abgebildet, so dass dieser Wert einen Wert des Werteuniversums `UVAL` bilden kann. Es ist mit der semantischen Abbildungen aber noch nicht ausgesagt, dass ein bestimmter Wert im Systemmodell-Universum `UVAL` enthalten ist, sondern lediglich wie die syntaktische Einheit in eine semantische Einheit übersetzt wird. Dies dient der strikten Trennung von Syntax und Semantik. Obwohl möglich, werden Syntax und Semantik selbst einfacher Einheiten wie Literale zugunsten einer systematischen Semantikbildung nicht identifiziert.

Definition SemAbb 7.2 (Literals)

LiteralsSemantics-base

Isabelle-Theorie

```
1   theory LiteralsSemantics-base
2   imports "$UMLP/gen/mc/literals/LiteralsAS"
3           "$UMLP/gen/SystemModel"
4   begin
5
6   fun mNUM-INT :: "NUM-INT ⇒ iVAL"
7   where
8    "mNUM-INT x = VInt x"
9
10  fun mNUM-LONG :: "NUM-LONG ⇒ iVAL"
11  where
12   "mNUM-LONG x = VInt x"
13
14  fun mCHAR :: "CHAR ⇒ iVAL"
15  where
16   "mCHAR c = VChar c"
17
18  fun mSTRING :: "STRING ⇒ iVAL"
19  where
20   "mSTRING s = VString s"
21
22  fun mNullLiteral :: "NullLiteral ⇒ iVAL"
23  where
24   "mNullLiteral NullLiteral = VOid VNil"
25
26  fun mCharLiteral :: "CharLiteral ⇒ iVAL"
```

```
where
 "mCharLiteral (CharLiteral c) = mCHAR c"

fun mBooleanLiteral :: "BooleanLiteral ⇒ iVAL"
where
  "mBooleanLiteral (BooleanLiteral BooleanLiteralsourceFALSE)
     = VBool False"
| "mBooleanLiteral (BooleanLiteral BooleanLiteralsourceTRUE)
     = VBool True"

fun mLongLiteral :: "LongLiteral ⇒ iVAL"
where
 "mLongLiteral (LongLiteral l) = mNUM-LONG l"

fun mStringLiteral :: "StringLiteral ⇒ iVAL"
where
 "mStringLiteral (StringLiteral s) = mSTRING s"

fun mIntLiteral :: "IntLiteral ⇒ iVAL"
where
 "mIntLiteral (IntLiteral i) = mNUM-INT i"

fun mNumericLiteral :: "NumericLiteral ⇒ iVAL"
where
  "mNumericLiteral (NumericLiteralIntLiteral i) = mIntLiteral i"
| "mNumericLiteral (NumericLiteralLongLiteral l) = mLongLiteral l"

fun mLiteral :: "Literal ⇒ iVAL"
where
  "mLiteral (LiteralNullLiteral n) = mNullLiteral n"
| "mLiteral (LiteralBooleanLiteral b) = mBooleanLiteral b"
| "mLiteral (LiteralStringLiteral s) = mStringLiteral s"
| "mLiteral (LiteralNumericLiteral n) = mNumericLiteral n"

end
```

In Definition 7.2 fehlt die semantische Abbildung für Namen bzw. Identifikatoren. Eine mögliche Variante, nämlich eine eins zu eins Umsetzung auf Namen im Systemmodell ist in Variante 7.3 angegeben. Um Verständnisschwierigkeiten vorzubeugen ist, wo sinnvoll, die Definitionsstelle einer Funktion oder eines Datentyps durch den qualifizierten Namen angegeben. So ist `Name` (Zeile 6) in der Systemmodell-Theorie `Base` (vgl. Definition B.1) bzw. in der abstrakten Syntax für Literale (siehe Definition C.2) definiert. Die Variante ist im Feature-Diagramm für die Variabilität der semantischen Abbildung in Definition 7.4 erfasst. Zusätzliche Bedingungen sorgen zum einen dafür, dass Fließkommawerte syntaktisch ausgeschlossen werden, zum anderen wird die Systemmodellvariante `CharAndString` gefordert, um Zeichen und Zeichenketten direkt und einfach abbilden zu können.

Variante SemAbb 7.3 (Direkte Umsetzung von Namen)

Name1to1

Isabelle-Theorie

```
1  theory Name1to1
2  imports "$UMLP/gen/mc/literals/LiteralsAS"
3          "$UMLP/gen/SystemModel"
4  begin
5
```

```
fun mName :: "LiteralsAS.Name ⇒ Base.Name"
where
 "mName x = x"

end
```

Definition Variabilität 7.4 (Semantik Literals)

LiteralsSemantics

Feature-Diagramm

```
package mc.literals.semantics;

featurediagram LiteralsSemantics {

 LiteralsSemantics = TranslateNames?;

 TranslateNames = Name1to1;

 constraints {
   LiteralsSemantics -> LiteralsConstr.RestrictFloatTypes;
   LiteralsSemantics -> SystemModel.vType.CharAndString;
 }
}
```

Die gesamte Variabilität des Sprachfragments zeigt Definition 7.5. Es enthält einen Verweis auf das Systemmodell als semantische Domäne, auf die Variabilität in Spracheinschränkungen (definiert in Anhang C.1.1, Definition C.4) und die Variabilität der semantischen Abbildung. Abkürzungen, Sprachparameter und Stereotypen wurden nicht definiert.

Definition Variabilität 7.5 (Literals)

Literals

Feature-Diagramm

```
package mc.literals;

<<lang>> featurediagram Literals {

 Literals = <<constr>> *mc.literals.syntax.LiteralsConstr &
            <<sm>> *mc.literals.semantics.LiteralsSemantics &
            <<sd>> *systemmodel.SystemModel;

}
```

7.2.2 Types

Das Sprachfragment Types stellt eine gemeinsame Definition von Typen zur Verfügung, die in allen UML/P-Teilsprachen verwendet wird. Es handelt sich hierbei um eine Java-kompatible Grammatik über die Struktur von Typen. Die MontiCore-Grammatik ist in Anhang C.1.2 in Definition C.5 ab Seite 284 angegeben. Aufgrund der gewünschten Konformität zu Java ist die Grammatik komplex, da neben primitiven Typen wie `boolean`, `int` oder (qualifizierten) Namen auch Arraytypen oder generische Typen darstellbar sein sollen. Die generierte abstrakte Syntax ist in Definition C.6 ab Seite 289 gelistet. In dieser Arbeit wollen wir aus Einfachheitsgründen nur eine eingeschränkte Menge von Typdefinitionen oder -verwendungen zulassen. Wir betrachten in der semantischen Abbildung die primitiven Typen `char`, `boolean`, `int` und `void` sowie den einfachen Referenztyp ohne Typargumente. Diese Einschränkungen sind in der optionalen Kontextbedingung C.7 kodiert und als syntaktische Variabilität im Feature-Diagramm in Definition C.8 dokumentiert.

Die semantische Abbildung in Definition 7.6 zeigt die eins zu eins Umsetzung der primitiven Typen sowie des Typs `void`. Qualifizierte Namen werden mit Hilfe von `mNameList` in Namen im Systemmodell übersetzt. Eine einfache Fassung dieser Übersetzung ist in Variante 7.7 angegeben, in der eine Liste von Identifikatoren auf einen Namen, der durch Punkte getrennt ist, überführt wird.

Definition SemAbb 7.6 (Types)

`TypesSemantics-base`

`Isabelle-Theorie`

```
theory TypesSemantics-base
imports "$UMLP/gen/mc/types/TypesAS"
        "$UMLP/gen/SystemModel"
        "$UMLP/gen/mc/literals/semantics/LiteralsSemantics"
begin

fun mPrimitiveType :: "PrimitiveType ⇒ iTYPE"
where
  "mPrimitiveType (PrimitiveType (Some PrimitiveTypeprimitiveCHAR)) = TChar"
| "mPrimitiveType (PrimitiveType (Some PrimitiveTypeprimitiveBOOLEAN)) = TBool"
| "mPrimitiveType (PrimitiveType (Some PrimitiveTypeprimitiveINT)) = TInt"

fun mVoidType :: "VoidType ⇒ iTYPE"
where
 "mVoidType v = TVoid"

consts mNameList :: "LiteralsAS.Name list ⇒ Base.Name"

fun mQualifiedName :: "TypesAS.QualifiedName ⇒ Base.Name"
where
 "mQualifiedName (QualifiedName ids) = mNameList ids"

end
```

Variante SemAbb 7.7 (Abbildung Typnamen)

`MapName`

`Isabelle-Theorie`

```
theory MapName
imports "$UMLP/gen/mc/types/TypesAS"
        "$UMLP/def/mc/types/semantics/TypesSemantics-base"
begin

fun vmNameList :: "LiteralsAS.Name list ⇒ Base.Name"
where
  "vmNameList (x#[]) = mName x"
| "vmNameList (x#xs) = (mName x)@’’.’’@(vmNameList xs)"

defs mNameList-def : "mNameList == vmNameList"

end
```

In der zweiten Variante 7.8 ist die semantische Abbildung für Typnamen insgesamt dargestellt. Es werden nur primitive oder einfache Referenztypen in der semantischen Abbildung berücksichtigt. Referenztypen werden direkt auf Klassen abgebildet und mit dem Typkonstruktor `TO` in einen Typ überführt. Die Übersetzung syntaktischer Typen (und Werte) benötigt nicht den Kontext Systemmodell. Es ist mit diesen semantischen Abbildungen wiederum nicht ausgesagt, dass ein bestimmter Typ im Systemmodell-Universum `UTYPE` enthalten ist, sondern lediglich wie die syntaktische Einheit in eine semantische Einheit übersetzt wird.

Variante SemAbb 7.8 (Abbildung Type)

SimpleTypeMapping

Isabelle-Theorie

```
theory SimpleTypeMapping
imports "$UMLP/gen/mc/types/TypesAS"
        "$UMLP/def/mc/types/semantics/TypesSemantics-base"
begin

fun mSimpleReferenceType :: "SimpleReferenceType ⇒ iTYPE"
where
"mSimpleReferenceType (SimpleReferenceType ids None)
      = TO (Class (mNameList ids))"

fun mReferenceType :: "ReferenceType ⇒ iTYPE"
where
 "mReferenceType (ReferenceTypeSimpleReferenceType srefT)
      = mSimpleReferenceType srefT"

fun mType :: "Type ⇒ iTYPE"
where
  "mType (TypePrimitiveType p) = mPrimitiveType p"
| "mType (TypeReferenceType refT) = mReferenceType refT"

fun mReturnType :: "ReturnType ⇒ iTYPE"
where
  "mReturnType (ReturnTypeVoidType v) = mVoidType v"
| "mReturnType (ReturnTypeType t) = mType t"

end
```

Die semantische Variabilität ist im Feature-Diagramm in Definition 7.9 zusammengefasst, wobei die Variante `SimpleTypeMapping` den syntaktischen Ausschluss komplizierterer Typnamen per Kontextbedingung `RestrictTypes` voraussetzt. Die Forderung der Auswahl der Variante `CharAndString` des Systemmodells ist zwar richtig, dient aber eher Dokumentationszwecken, da über die Semantik für Literale die Verwendung bereits sichergestellt ist. Die gesamte Variabilität des Sprachfragments `Types` ist im Feature-Diagramm in Definition 7.10 definiert.

Definition Variabilität 7.9 (Semantik Types)

TypesSemantics

Feature-Diagramm

```
package mc.types.semantics;

featurediagram TypesSemantics {

  TypesSemantics = TypeMapping? & vMapName?;

  TypeMapping = SimpleTypeMapping;

  vMapName = MapName;

  constraints {
    TypesSemantics -> SystemModel.vType.CharAndString;
    TypeMapping.SimpleTypeMapping -> TypesConstr.RestrictTypes;
  }
}
```

Definition Variabilität 7.10 (Types)

Types

Feature-Diagramm

```
package mc.types;

<<lang>> featurediagram Types {

 Types = <<constr>> *mc.types.syntax.TypesConstr &
         <<sm>> *mc.types.semantics.TypesSemantics &
         <<sd>> *systemmodel.SystemModel;

}
```

7.2.3 Common

Als letztes gemeinsam genutztes Sprachfragment bietet Common verschiedene wiederverwendbare Konstrukte an. In der MontiCore-Grammatik in Definition C.9 im Anhang C.1.3 werden Stereotypen, Kardinalitäten und Modifier (inkl. Sichtbarkeiten) syntaktisch definiert. Ebenfalls eingeführt werden Qualifier, die bei der Definition von qualifizierten Assoziationen in Klassendiagrammen eine Rolle spielen und die aus [Rum04b] bekannte Notation `Completeness` zur Angabe, wie vollständig ein Modell ist. Die abstrakte Syntax ist in Definition C.10 zu finden. In Variante C.11 auf Seite 296 definieren wir eine Kontextbedingung, die sicherstellt, dass immer höchstens eine Sichtbarkeit durch einen Modifier definiert wird. Die syntaktische Variabilität ist im Feature-Diagramm in Definition C.12 zu finden.

Die Semantik für die Konstrukte des Sprachfragments Common ist meist spezifisch für eine Modellierungssprache, insbesondere die Bedeutung von Stereotypen oder bestimmter Modifier. Sichtbarkeiten werden generell nicht semantisch behandelt, da hier eine syntaktische Prüfung ausreichend ist. Wir gehen davon aus, dass diese Überprüfung stattfindet und definieren sie nicht.

`Completeness`-Angaben für ein Modellelement beziehen sich darauf, dass ein Modellelement bezüglich eines Gesamtmodells des Systems vollständig definiert ist oder nicht. Die Angabe bezieht sich nicht auf das zugrunde liegende System [Rum04b]. Das bedeutet, dass `Completeness` allenfalls als (Inter-)Kontextbedingung zu prüfen ist und semantisch keine Rolle spielt. `Completeness` wird daher in folgenden Semantiken stets ignoriert.

Kardinalitäten werden verwendet, um auszudrücken, wie oft ein bestimmtes Element im System (zum Beispiel Objekte einer Klasse) mindestens oder höchstens auftreten darf. Für Kardinalitäten schlagen wird eine semantische Abbildung in Variante 7.11 vor. Hiernach bestimmt eine Kardinalität einen Teilbereich der natürlichen Zahlen, der auch unendlich sein kann.

Variante SemAbb 7.11 (Abbildung Kardinalitäten)

MapCardinality

Isabelle-Theorie

```
theory MapCardinality
imports "$UMLP/gen/mc/umlp/common/CommonAS"
begin

fun mCardinality :: "Cardinality ⇒ nat set"
where
  "mCardinality (Cardinality (Some n) None (Some m) None)
       = {x | x . x >= nat n ∧ x <= nat m}"
| "mCardinality
    (Cardinality (Some n) (Some CardinalitynoUpperLimitSTAR)  None None)
       = {x | x . x >= nat n}"
| "mCardinality (Cardinality None None None (Some CardinalitymanySTAR))
      = {x | x . True}"

fun mCardinalityOpt :: "Cardinality option ⇒ nat set"
where
  "mCardinalityOpt None = {x | x . True}"
| "mCardinalityOpt (Some c) = mCardinality c"

end
```

Qualifier werden verwendet, um anzuzeigen, dass auf ein Element im System mit Hilfe eines Wertes eindeutig zugegriffen werden kann. Eine semantische Abbildung für Qualifier ist in Variante 7.12 angegeben. Ein Qualifier beschreibt eine Menge erlaubter Werte. Handelt es sich um einen Typnamen, schränkt sich diese Menge auf die Trägermenge des Typs ein. Ansonsten können allein aus der Definition des Qualifier keine weiteren Informationen gewonnen werden und alle Werte (`UVAL`) sind möglich.

Variante SemAbb 7.12 (Abbildung Qualifier)

MapQualifier

Isabelle-Theorie

```
theory MapQualifier
imports "$UMLP/gen/mc/umlp/common/CommonAS"
        "$UMLP/gen/mc/types/semantics/TypesSemantics"
begin

fun mQualifier  :: "Qualifier ⇒ SystemModel ⇒ iVAL set"
where
  "mQualifier (Qualifier typeorName) sm =
     (if (mType typeorName ∈ UTYPE sm)
      then CAR sm (mType typeorName)
      else UVAL sm)"

end
```

Die semantische Abbildung für das Sprachfragment Common enthält nur Varianten, die im Feature-Diagramm in Definition 7.13 dargestellt sind. Die gesamte Variabilität des Sprachfragments ist in Definition 7.14 zu finden.

Definition Variabilität 7.13 (Semantik Common)

CommonSemantics

Feature-Diagramm

```
package mc.umlp.common.semantics;

featurediagram CommonSemantics {

  CommonSemantics = MapCardinality? & MapQualifier?;

}
```

Definition Variabilität 7.14 (Common)

Common

Feature-Diagramm

```
package mc.umlp.common;

<<lang>> featurediagram Common {

 Common = <<constr>> *mc.umlp.common.syntax.CommonConstr &
          <<sm>> *mc.umlp.common.semantics.CommonSemantics &
          <<sd>> *systemmodel.SystemModel;
}
```

7.3 Java/P und OCL/P

Bei den in dieser Arbeit angegebenen Versionen von Java/P und OCL/P handelt es sich um rudimentäre Definitionen der Sprachen. Sie können nicht eigenständig verwendet werden und sind nur zur Einbettung in andere UML/P-Diagrammarten geeignet.

7.3.1 Java/P

Die in Definition C.13 in Anhang C.2.1 ab Seite 296 aufgeführte Grammatik stellt einen kleinen Teil von Java-Ausdrücken (Expressions) und -Anweisungen (Statements) zur Verfügung. Zudem fehlen alle Konstrukte, mit denen beispielsweise die Struktur einer Java-Klasse definiert werden könnte. Syntaktisch definiert werden nur einige wenige logische und arithmetische Ausdrücke, sowie Anweisungen für Methodenrückgaben (Return), Variablendeklarationen, Zuweisungen sowie Blockanweisungen. Die resultierende abstrakte Syntax ist auf Seite 298 in Definition C.14 dargestellt. Die Kontextbedingung in Definition C.15 stellt unter Verwendung der entsprechenden Kontextbedingung für Literale im Wesentlichen sicher, dass Literale keine Fließkommawerte sind und dass ein Anweisungsblock vereinfachend nur aus einer Anweisung bestehen darf.

In den Definitionen 7.15 bis 7.22 wird die Java/P-Semantik sukzessive definiert. Eine Funktion zur Auswertung von Infixoperationen ist in Definition 7.15 enthalten. Arithmetische Operationen liefern einen ganzzahligen Wert, logische Operatoren einen Wahrheitswert. Vergleiche funktionieren für alle Elemente in `UVAL`, denn sie basieren auf der Vergleichbarkeit der Elemente des Datentyps `iVAL`. Bei arithmetischen und anderen logischen Operatoren hingegen werden die Operanden zunächst in den zugrunde liegenden Isabelle-Typ umgewandelt. Das Verhalten bei Typfehlern ist nicht definiert (vgl. Definition B.1).

Definition SemAbb 7.15 (JavaP Semantik, Infix-Ausdrücke)

EvalInfix

Isabelle-Theorie

```
fun evalInfix ::
  "iVAL ⇒ iVAL ⇒ InfixExpressionOperator ⇒ iVAL"
where
  "evalInfix v1 v2 InfixExpressionOperatorEQUALSEQUALS
      = VBool (v1 = v2)"
| "evalInfix v1 v2 InfixExpressionOperatorGT
      = VBool (toInt v1 > toInt v2)"
| "evalInfix v1 v2 InfixExpressionOperatorLT
      = VBool (toInt v1 < toInt v2)"
| "evalInfix v1 v2 InfixExpressionOperatorPLUS
      = VInt (toInt v1 + toInt v2)"
| "evalInfix v1 v2 InfixExpressionOperatorMINUS
      = VInt (toInt v1 - toInt v2)"
| "evalInfix v1 v2 InfixExpressionOperatorEXCLAMATIONMARKEQUALS
      = VBool (v1 ≠ v2)"
| "evalInfix v1 v2  InfixExpressionOperatorPIPEPIPE
      = VBool (toBool v1 ∨ toBool v2)"
| "evalInfix v1 v2 InfixExpressionOperatorANDAND
      = VBool (toBool v1 ∧ toBool v2)"
```

In Definition 7.16 werden Literale und Namen behandelt. Für die semantische Abbildung von Literalen wird auf die semantische Abbildung aus Definition 7.2 zurückgegriffen. Sie liefert direkt einen Wert. Namen (zum Beispiel der Name einer lokalen Variable oder eines Methodenparameters) liefern ebenfalls einen Wert. Jedoch wird hier der Kontext zum Auflösen des Namens benötigt. Handelt es sich um ein Attribut einer Klasse, ist der Wert im Datenzustand des übergebenen Zustands unter einem Objektidentifikator abgelegt. Lokale Variablen oder Parameter werden im Kontrollzustand gespeichert, wobei für den Zugriff hierauf zusätzlich ein Thread benötigt wird. Alternativ kann der Wert auch „außerhalb“ des Systems in der übergebenen Variablenbelegung abgelegt sein. Die Auflösung des Namens ist ein Variationspunkt, weshalb die Funktion `lookup` nur deklariert und nicht endgültig definiert wird.

Definition SemAbb 7.16 (JavaP Semantik, Literale)

mLiteral

Isabelle-Theorie

```
consts lookup ::
  "LiteralsAS.Name list ⇒ iSTATE ⇒ iVarAssign ⇒
     iOID ⇒ iTHREAD ⇒ SystemModel ⇒ iVAL"

fun mJLiteral :: "JLiteral ⇒ iVAL"
where
 "mJLiteral (JLiteral lit) = mLiteral lit"

fun mJName ::
  "JName ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ iVAL"
where
 "mJName (JName (QualifiedName ln)) s va oid th sm = lookup ln s va oid th sm"
```

Eine einfache Realisierung von `lookup` ist in Variante 7.17 dargelegt. Die Funktionsvariante `slookup` prüft, ob ein qualifizierter Name vorliegt. Ist dies der Fall, identifiziert die erste Namenskomponente eine Objektreferenz, die in der Variablenbelegung gespeichert ist. Danach wird im Datenzustand dieses Objekts nachgesehen, wobei weitere Namenskomponenten (bis auf die letzte) rekursiv zu weiteren Objektreferenzen führen. Die letzte Namenskomponente identifiziert dann einen Attributwert des zuvor gefundenen Objekts. Namen werden in dieser Variante also zuerst in der Variablenbelegung gesucht, danach wird im Datenzustand eines Objekts nachgesehen. Der Kontrollzustand wird in dieser Variante nicht untersucht.

Variante SemAbb 7.17 (Namensauflösung)

SimpleNameLookup

Isabelle-Theorie

```
theory SimpleNameLookUp
imports "$UMLP/def/mc/javap/semantics/JavaPSemantics-base"
begin

fun lookupDS :: "LiteralsAS.Name list ⇒ iSTATE ⇒ iOID ⇒ iVAL"
where
  "lookupDS (x#[]) s oid = (the (getAttr ((the ((dsOf s) oid)), mName x)))"
| "lookupDS (x#xs) s oid =
     lookupDS xs s (toOid (the (getAttr ((the ((dsOf s) oid)), mName x))))"

fun slookup ::
  "LiteralsAS.Name list ⇒ iSTATE ⇒ iVarAssign ⇒
    iOID ⇒ iTHREAD ⇒ SystemModel ⇒ iVAL"
where
 "slookup n s va oid th sm = (
    if (length n > 1)
    then (lookupDS (tl n) s (toOid (the (va(mName(hd n))))))
    else (if (va(mNameList n) ≠ None)
          then (the (va(mNameList n)))
          else (the (getAttr ((the ((dsOf s) oid)), mNameList n)))))"

defs lookup-def : "lookup == slookup"

end
```

Definition 7.18 gibt die semantische Abbildung für Java/P-Ausdrücke an. Ausdrücke sind entweder Namen, Literale oder Infix-Operationen. Bei Infix-Operationen werden erst die linke und rechte Seite des Ausdrucks ausgewertet und danach der Operator mit Hilfe der Funktion `evalInfix` angewendet. Wird eine deutlich umfangreichere Sprache für Ausdrücke benötigt, sollte die Definition der semantischen Abbildung als induktive Definition erfolgen, da eine höhere Anzahl wechselseitig rekursiver Funktionsaufrufe in der verwendeten Isabelle-Version nicht skaliert. Der Vorteil einer Funktionsdefinition ist, dass eine Auswertung eines Ausdrucks mit konkreten Werten automatisch möglich ist. Dies ist bei induktiven Definition so nicht der Fall.

Definition SemAbb 7.18 (JavaP Semantik, Ausdrücke)

mJExpression

Isabelle-Theorie

```
fun
 mExpression ::
   "JavaPAS.Expression ⇒
```

```
      iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ iVAL"
and
 mInfixExpression ::
   "JavaPAS.InfixExpression ⇒
      iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ iVAL"
where
  "mExpression (ExpressionJName expr) s va oid th sm =
      mJName expr s va oid th sm"
| "mExpression (ExpressionJLiteral expr) s va oid th sm =
      mJLiteral expr"
| "mExpression (ExpressionInfixExpression expr) s va oid th sm =
      mInfixExpression expr s va oid th sm"
| "mInfixExpression (InfixExpression expr1 expr2 opr) s va oid th sm =
      evalInfix
        (mExpression expr1 s va oid th sm)
        (mExpression expr2 s va oid th sm) opr"
```

Die Semantik einer Methodenrückgabe ist in Definition 7.19 beschrieben. Generell werden für die Semantik von Anweisungen neben dem Objekt, das die Anweisung „enthält“ und einem Thread, der die Anweisung ausführt, zwei Zustände benötigt. Der erste Zustand ist der Ausgangszustand und der zweite Zustand ist ein Folgezustand, der den Effekt der Anweisung widerspiegelt. Im Fall einer Methodenrückgabe erhält das aufrufende Objekt eine Nachricht, die die Rückgabe der Methode darstellt. Der Aufrufer und der Methodenname werden aus dem aktiven Frame des Kontrollzustands für das angegebene Objekt und dem angegebenen Thread identifiziert. Der Rückgabewert ist entweder der Wert *void* oder ergibt sich durch Auswertung des angegebenen Ausdrucks.

Definition SemAbb 7.19 (JavaP Semantik, Return-Anweisung)

mReturnStatement

Isabelle-Theorie

```
fun mReturnStatement ::
  "ReturnStatement ⇒
     iSTATE ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
 "mReturnStatement (ReturnStatement exprOpt) s s′ va oid th sm =
  (let n = StackFrame.methodNameOf (topF((the(the((csOf s) oid) th)))) in
   let c = StackFrame.caller       (topF((the(the((csOf s) oid) th)))) in
   let ret =  (if (exprOpt = None)
               then [VVoid]
               else ([mExpression (the exprOpt) s va oid th sm])) in
                 State.hasMsg s′ c (c, n, ret, oid, (Some th)))"
```

Definition 7.20 enthält die semantische Abbildung der Definition einer lokalen Variablen. Die Variable wird im aktiven Frame des Kontrollzustands des Folgezustands `s′` (identifiziert durch Objekt und Thread) gespeichert. Ist ein Ausdruck angegeben, nimmt die lokale Variable den Wert des Ausdrucks an, auf jeden Fall aber stammt der Wert aus der Trägermenge des Typs der Variable. Weitere Änderungen am aktuellen Frame, beispielsweise das „Hochzählen“ des Programmzählers bleiben unterspezifiziert.

Definition SemAbb 7.20 (JavaP Semantik, Variablendeklaration)

mVariableDeclaration

Isabelle-Theorie

```
fun mVariableDeclarationStatement ::
  "VariableDeclarationStatement ⇒
     iSTATE ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
 "mVariableDeclarationStatement
   (VariableDeclarationStatement tp n exprOpt) s s' va oid th sm =
     (let val = (if (exprOpt = None) then arbitrary
                 else (mExpression (the exprOpt) s va oid th sm)) in
      let frame = (topF((the(the((csOf s') oid) th)))) in
        localVarAssign frame (mName n) ≠ None ∧
        (the ((localVarAssign frame) (mName n))) = val ∧
         val ∈ CAR sm (mType tp)
     )"
```

Bei einer Zuweisung (Definition 7.21) wird zunächst der angegebene Ausdruck ausgewertet. Falls eine lokale Variable des angegebenen Namens vorhanden ist, wird ihr der Wert zugewiesen. Ansonsten handelt es sich um ein Attribut der Klasse und der Datenzustand wird entsprechend aktualisiert. Da wir im Systemmodell von global eindeutigen Variablennamen ausgehen, findet keine Verschattung statt und der Zugriff auf eine Variable ist eindeutig.

Definition SemAbb 7.21 (JavaP Semantik, Zuweisung)

mAssignmentStatement

Isabelle-Theorie

```
fun mAssignmentStatement ::
  "AssignmentStatement ⇒
     iSTATE ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
 "mAssignmentStatement (AssignmentStatement n expr) s s' va oid th sm =
   (let vl = mExpression expr s va oid th sm in
     let frame = (topF((the(the((csOf s) oid) th)))) in
      if ((localVarAssign frame) (mName n) ≠ None)
      then (let frame' = (topF((the(the((csOf s') oid) th)))) in
           (the ((localVarAssign frame) (mName n))) = vl)
      else (the (val sm (dsOf s', oid, (mName n))) = vl)
   )"
```

Bei der semantischen Abbildung für Anweisungen insgesamt in Definition 7.22 wird auf die Abbildungen für Variablendeklaration, Zuweisung und Methodenrückgabe zurückgegriffen.

Zusätzlich ist die Bedeutung von Anweisungsblöcken definiert, Wir beschränken uns bei Anweisungsblöcke auf nur eine Anweisung und vermeiden dadurch eine aufwendigere Kodierung von Zwischenzuständen (Betrachung eines Subtrace).

Definition SemAbb 7.22 (JavaP Semantik, Anweisungen)

mJStatement

Isabelle-Theorie

```
fun mStatement ::
  "JavaPAS.Statement ⇒
     iSTATE ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
and
```

```
  mBlockStatement ::
   "JavaPAS.BlockStatement ⇒
      iSTATE ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
 where
  "mStatement (StatementVariableDeclarationStatement  stmt) s s’ va oid th sm =
      mVariableDeclarationStatement stmt s s’ va oid th sm"
 |"mStatement (StatementAssignmentStatement stmt) s s’ va oid th sm =
      mAssignmentStatement stmt s s’ va oid th sm"
 |"mStatement (StatementReturnStatement stmt) s s’ va oid th sm =
      mReturnStatement stmt s s’ va oid th sm"
 |"mStatement (StatementBlockStatement b) s s’ va oid th sm =
      mBlockStatement b s s’ va oid th sm"
 |"mBlockStatement (BlockStatement [x]) s s’ va oid th sm =
      mStatement x s s’ va oid th sm"
```

Die Variabilität der semantischen Abbildung beschränkt sich auf die Namensauflösung und ist in Definition 7.23 aufgeführt. Die gesamte Variabilität von Java/P enthält Definition 7.24.

Definition Variabilität 7.23 (Semantik JavaP)

JavaPSemantics

Feature-Diagramm

```
package mc.javap.semantics;

featurediagram JavaPSemantics {

 JavaPSemantics = vNameLookUp?;

 vNameLookUp = SimpleNameLookUp;

}
```

Definition Variabilität 7.24 (JavaP)

JavaP

Feature-Diagramm

```
package mc.javap;

<<lang>> featurediagram JavaP {

 JavaP = <<constr>> *mc.javap.syntax.JavaPConstr &
         <<sm>> *mc.javap.semantics.JavaPSemantics &
         <<sd>> *systemmodel.SystemModel;

}
```

Wir verwenden gemäß [Rum04b] Java/P als Aktionssprache zur Einbettung in andere UML/P-Diagrammarten, betrachten aber nur einen kleinen Teil einer echten Programmiersprache wie Java, da eine ausführliche Semantik den Rahmen dieser Arbeit sprengen würde. Im Kontext von Isabelle/HOL ist das Projekt Jinja [KN06] erwähnenswert, in dem eine in Isabelle kodierte Java-Teilsemantik definiert wird. Neben einfachen Ausdrücken und Anweisungen werden auch Methodenaufrufe und Ausnahmebehandlung formalisiert. Eine weiterführende Arbeit betrachtet auch Nebenläufigkeit durch Threads [Loc08]. Die Semantik ist als operationale Semantik angegeben. Die Syntax wird stark vereinfacht manuell in Isabelle/HOL kodiert. Im OMG-Standard für UML [OMG09b] wird statt einer existierende Programmiersprache eine spezielle *action language* definiert. Eine konkrete Syntax hierfür ist nicht festgelegt. Es ist prinzipiell möglich, die Semantik für Aktionen und Aktivitäten von Crane [Cra09], die auch auf dem Systemmodell basiert, in Isabelle zu kodieren und an Stelle von Java/P zu verwenden.

7.3.2 OCL/P

OCL/P ist eine syntaktisch an Java angelehnte Fassung der OCL [OMG06b]. Syntax und Semantik werden ausführlich in [Rum04b] beschrieben. Im Rahmen dieser Arbeit betrachten wir wieder nur einen sehr kleinen Ausschnitt dieser Sprache. Die konkrete und abstrakte Syntax finden sich in Anhang C.2.2. Die in Definition C.19 definierten Kontextbedingungen sichern wiederum nur, dass keine Fließkommawerte verwendet werden. Ansonsten werden keine syntaktischen Varianten definiert. Die Ausdrucksstärke ist wie bei Java/P eingeschränkt, Ausdrücke für `forall`, `any` etc. sowie Methodenvor- und Nachbedingungen werden nicht berücksichtigt. Betrachtet werden im Folgenden die Semantik einfacher Ausdrücke und Invarianten.

Definition 7.25 definiert die Berechnung von Infixoperationen. Hier sind im Vergleich zu Java/P zwei weitere logische Operatoren, nämlich Implikation und Äquivalenz, berücksichtigt.

Definition SemAbb 7.25 (OCL Semantik, Infix-Ausdrücke)

OCLInfix

Isabelle-Theorie

```
fun oclEval ::
  "iVAL ⇒ iVAL ⇒ OCLInfixExpressionOperator ⇒ iVAL"
where
  "oclEval v1 v2 OCLInfixExpressionOperatorEQUALSEQUALS
      = VBool (v1 = v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorEXCLAMATIONMARKEQUALS
      = VBool (v1 ≠ v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorGT
      = VBool (toInt v1 > toInt v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorLT
      = VBool (toInt v1 < toInt v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorPLUS
      = VInt (toInt v1 + toInt v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorMINUS
      = VInt (toInt v1 - toInt v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorIMPLIES
      = VBool (toBool v1 ⟶ toBool v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorLTEQUALSGT
      = VBool (toBool v1 ⟷ toBool v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorPIPEPIPE
      = VBool (toBool v1 ∨ toBool v2)"
| "oclEval v1 v2 OCLInfixExpressionOperatorANDAND
      = VBool (toBool v1 ∧ toBool v2)"
```

Literale werden gemäß Definition 7.26 direkt auf Werte abgebildet, während die semantische Abbildung von Namen einen Kontext zur Namensauflösung für die Funktion `oclLookup` benötigt. An der Signatur dieser Deklaration (und auch an noch folgenden Signaturen) ist erkennbar, dass im Vergleich zu Java/P der Objektidentifikator und Thread fehlen. Das liegt daran, dass OCL-Ausdrücke allgemein nicht wie Java-Ausdrücke im Kontext einer Berechnung (z.B.

Methodenausführung) in einem Objekt durch einen Thread ausgewertet werden. Jegliche Informationen über beteiligte Objekte sind in der Variablenbelegung (die auch Objektreferenzen speichern kann) zu finden.

Definition SemAbb 7.26 (OCL Semantik, Literale und Namen)

OCLLiteral

Isabelle-Theorie

```
consts oclLookup ::
  "LiteralsAS.Name list ⇒ iSTATE ⇒ iVarAssign ⇒ iVAL"

fun mOCLName :: "OCLName ⇒ iSTATE ⇒ iVarAssign ⇒ iVAL"
where
 "mOCLName (OCLName (QualifiedName ln)) st vars = oclLookup ln st vars"

fun mOCLLiteral :: "OCLLiteral ⇒ iVAL"
where
 "mOCLLiteral (OCLLiteral l) = mLiteral l"
```

Variante 7.27 definiert entsprechend eine einfache Namensauflösung, die erwartet, dass die erste Komponente eines qualifizierten Namens ein Objektidentifikator ist, der in der Variablenbelegung zu finden ist. Mit diesem Identifikator wird dann der Datenzustand nach einem Attributwert (ggf. rekursiv) abgefragt.

Variante SemAbb 7.27 (OCL Namensauflösung)

OCLSimpleNameLookUp

Isabelle-Theorie

```
theory OCLSimpleNameLookUp
imports "$UMLP/def/mc/umlp/ocl/semantics/OCLSemantics-base"
begin

fun lookupDS :: "LiteralsAS.Name list ⇒ iSTATE ⇒ iOID ⇒ iVAL"
where
  "lookupDS (x#[]) s oid = (the (getAttr ((the ((dsOf s) oid)), mName x)))"
| "lookupDS (x#xs) s oid =
     lookupDS xs s (toOid (the (getAttr ((the ((dsOf s) oid)), mName x))))"

fun soclLookup :: "LiteralsAS.Name list ⇒ iSTATE ⇒ iVarAssign ⇒ iVAL"
where
 "soclLookup (n#xs) s va =
     lookupDS xs s (toOid (the (va((mName n)))))"

defs oclLookup-def : "oclLookup == soclLookup"

end
```

In Definition 7.28 werden OCL-Ausdrücke auf Werte abgebildet. Neben der bereits besprochenen Abbildung für Infixausdrücke über `oclEval`, Literale und Namen ist die Semantik eines OCL-Methodenaufrufs (ab Zeile 22) angegeben. Wir gehen davon aus, dass unter dem Namen der Methode eine Abfrage (Query, siehe Systemmodellvariante B.47 auf Seite 271) zu finden ist. Argumente dieser Abfrage sind der Zustand, eine Variablenbelegung, ein Objektidentifikator und ein Thread. Der angegebene Thread spielt in unserem Fall keine Rolle, da wir keine Informationen über aktuell laufende Berechnungen abfragen. Da eine Abfrage zustandslos ist, müssen auch keine Ausführungsinformationen in einem Folgezustand gespeichert werden. Zuletzt erhält die Query noch eine Liste von Parameterwerten, die erst ausgewertet werden bevor sie übergeben werden.

Definition SemAbb 7.28 (OCL Semantik, OCL-Ausdrücke)

OCLExpression

Isabelle-Theorie

```
fun
 mOCLExpression ::
    "OCLExpression ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ iVAL"
and
 mOCLInfixExpression ::
    "OCLInfixExpression ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ iVAL"
and
 mOCLMethodInvocation ::
    "OCLMethodInvocation ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ iVAL"
where
  "mOCLExpression (OCLExpressionOCLInfixExpression iexpr) st vars sm =
     mOCLInfixExpression iexpr st vars sm"
| "mOCLExpression (OCLExpressionOCLLiteral l) st vars sm =
     mOCLLiteral l"
| "mOCLExpression (OCLExpressionOCLMethodInvocation mi) st vars sm  =
       mOCLMethodInvocation mi st vars sm"
| "mOCLExpression (OCLExpressionOCLName n) st vars sm =
      mOCLName n st vars"
| "mOCLInfixExpression (OCLInfixExpression expl expr ope) st vars sm =
     oclEval (mOCLExpression expl st vars sm)
             (mOCLExpression expr st vars sm) ope"
| "mOCLMethodInvocation (OCLMethodInvocation qmname exprL) st vars sm =
    ((query sm (mName (last qmname))) st vars
        (toOid (oclLookup (butlast(qmname)) st vars)) (Thread (mName ''oclC''))
          (map (λ expr . mOCLExpression expr st vars sm) exprL) sm)"
```

Schließlich wird die Semantik einer OCL-Invariante in Definition 7.29 bestimmt. Aus dem Kontext der Invariante wird eine Variablenbelegung erstellt, die Namen auf Objekte abbildet (vgl. Zeile 7). Wie diese Abbildung stattfindet, klärt der Variationspunkt `mapObject`, für den eine einfache Realisierung in Variante 7.30 dargestellt ist. Für alle erreichbaren Zustände im Systemmodell, die einen gültigen Kontext aufweisen (das heißt, die entsprechenden Objekte existent sind und den richtigen Typ besitzen) muss der angegebene OCL-Ausdruck wahr sein (Zeilen 21-23). Wir gehen vereinfachend davon aus, dass der Kontext eindeutig bestimmt ist und es damit nur eine gültige Variablenbelegung gibt. Eine Verallgemeinerung dieser Definition ist jedoch leicht möglich.

Definition SemAbb 7.29 (OCL Semantik, OCL-Invarianten)

OCLInvariant

Isabelle-Theorie

```
1  consts mapObject :: "LiteralsAS.Name ⇒ iOID"
2
3  fun makeContext :: "OCLContextDefinition list ⇒ iVarAssign ⇒ iVarAssign"
4  where
5    "makeContext [] vars = vars"
6  | "makeContext ((OCLContextDefinition className (Some name))#xs) vars =
7       makeContext xs (vars(mName name↦ VOid (mapObject name)))"
8
```

```
fun validContext :: "OCLContextDefinition list ⇒ iSTATE ⇒ SystemModel ⇒ bool"
where
  "validContext [] s sm = True"
| "validContext ((OCLContextDefinition className (Some name))#xs) s sm =
        ( (mapObject name) ∈ oids s ∧
          VOid (mapObject name) ∈ CAR sm (mType className) ∧
          validContext xs s sm )"

fun mOCLInvariant :: "OCLInvariant ⇒ SystemModel ⇒ bool"
where
 "mOCLInvariant (OCLInvariant contextList nameOpt expr) sm =
     (let vars =  makeContext contextList empty in
        ∀ s ∈ reachable sm .
               validContext contextList s sm ⟶
                    mOCLExpression expr s vars sm = (VBool True))"
```

Variante SemAbb 7.30 (Abbildung von Namen auf Objekte)

OCLMapObject

Isabelle-Theorie

```
theory OCLMapObject
imports "$UMLP/def/mc/umlp/ocl/semantics/OCLSemantics-base"
begin

fun vmapObject :: "LiteralsAS.Name ⇒ iOID"
where
 "vmapObject n = Oid (mName n)"

defs mapObject-def : "mapObject == vmapObject"

end
```

Die semantische Variabilität ist in Definition 7.31 zusammengefasst, während Definition 7.32 die gesamte Variabilität von OCL/P in einem Feature-Diagramm modelliert. Im Kontext von Isabelle/HOL gibt es einige interessante Arbeiten zur Umsetzung von OCL in Isabelle [BW06, BW09]. Bei der Kodierung handelt es sich um eine flache Einbettung der OCL. Das heißt, dass die Syntax nicht in Isabelle repräsentiert ist. Die OMG-Version der OCL basiert auf einer dreiwertigen Logik, undefinierte Werte werden explizit kodiert. OCL/P verzichtet auf diese etwas kompliziertere Formulierung und kommt mit einer klassischen zweiwertigen Logik aus. In dieser Arbeit werten wir OCL-Ausdrücke im Kontext des Systemmodells aus. In [BW06] geschieht die Auswertung im Kontext einer speziellen erweiterbaren Objektstruktur. Diese hat den Vorteil der nachträglichen Erweiterbarkeit, erfordert aber eine deutlich komplexere Kodierung, was die Verständlichkeit reduziert.

Definition Variabilität 7.31 (Semantik OCL)

OCLSemantics

Feature-Diagramm

```
package mc.umlp.ocl.semantics;

featurediagram OCLSemantics {

  OCLSemantics = vNameLookUp? & vMapObject?;

  vNameLookUp = OCLSimpleNameLookUp;

  vMapObject = OCLMapObject;

}
```

Definition Variabilität 7.32 (OCL)

OCL

Feature-Diagramm

```
package mc.umlp.ocl;

<<lang>> featurediagram OCL {

 OCL   = <<constr>> *mc.umlp.ocl.syntax.OCLConstr &
         <<sm>> *mc.umlp.ocl.semantics.OCLSemantics &
         <<sd>> *systemmodel.SystemModel;

}
```

7.4 Klassendiagramme

UML/P Klassendiagramme werden hauptsächlich verwendet, um die Struktur eines objektorientierten Softwaresystems zu beschreiben. Mit einem Klassendiagramm beschreiben wir also Elemente wie Klassen, Beziehungen zwischen Klassen, Methoden, Attribute und Konstruktoren von Klassen oder Schnittstellen, die ein System mindestens aufweisen muss. Aufgrund des gewählten prädikativen Ansatzes zur losen Semantikdefinition schließt das nicht aus, das weitere, nicht im Diagramm erfasst Bestandteile des Systems existieren. Invarianten im Klassendiagramm, formuliert in einer eingebetteten Constraint-Sprache, können genutzt werden, um die Ausdrucksmächtigkeit zu verbessern und die Menge der erlaubten Objektstrukturen, die von einem Klassendiagramm impliziert wird, weiter einzuschränken.

Obwohl syntaktisch möglich, sollen Klassendiagramme in unserer Semantik nicht zur Verhaltensbeschreibung verwendet werden. Methodenrümpfe, die in einer Aktionssprache ausformuliert sind, werden als Abfrage ohne Seiteneffekte interpretiert. Dies stellt einen Variationspunkt dar und könnte in einer anderen Variante der Semantik anders gelöst werden.

Die MontiCore-Grammatik für Klassendiagramme ist in Definition C.21 in Abschnitt C.3 des Anhangs zu finden, die resultierende abstrakte Syntax in Definition C.22. Die Spracheinschränkungen in der Variante C.24 kodieren im Wesentlichen, welche Konstrukte semantisch nicht berücksichtigt werden und daher syntaktisch ausgeschlossen werden sollen. Zum einen werden Enumerationen nicht behandelt, da sie sich leicht in normale Klassen syntaktisch transformieren lassen. Zum anderen wird Generizität ausgeschlossen, da diese im Wesentlichen eine Rolle bei der (syntaktischen) Überprüfung der Typkorrektheit spielt und beispielsweise in Java zur Laufzeit weitgehend nicht vorhanden ist. Dennoch ließen sich generische Definition auch semantisch behandeln. Hierfür wäre jedoch eine umfangreichere Variante des Systemmodells, wie in [BCGR08] angedeutet, zu definieren. Die Deklaration von *Exceptions* (Ausnahmen) wird ebenfalls nicht berücksichtigt. Weiterhin wird überprüft, ob die verwendeten Typnamen den in Variante C.7 (RestrictTypes) geforderten Einschränkungen entsprechen.

Exemplarisch werden in Definition C.23 auch einige notwendige Kontextbedingungen angegeben, die die sinnvolle Verwendung einiger Konstrukte im Kontext beschreiben. So sind die Modifier *static* und *final* sowie ein Methodenrumpf für Interfaces nicht erlaubt. Kontextbedingungen zur Sicherung der referentiellen Integrität (hierunter fällt beispielsweise, dass linke und rechte Seite einer (binären) Assoziation ebenfalls im Diagramm definiert sind) werden im Allgemeinen nicht benötigt, da die semantische Abbildung selbst die Existenz der entsprechenden Elemente fordert. Sie könnten also auch in einem weiteren Diagramm oder gar nicht weiter spezifiziert werden. Für eingebettete Sprachteile (`InvariantExpression`, `Body` und `Value`) wird angenommen, dass eine Funktion zur Überprüfung der Wohlgeformtheit existiert. Veranschaulicht wird dies an der Überprüfung der Invarianten in der Definition C.23.

Eine optionale Kontextbedingung, die die Mehrfachvererbung von Klassen im Klassendiagramm syntaktisch ausschließt, ist in Variante C.25 kodiert. Die gesamte syntaktische Variabilität von Klassendiagrammen ist im Feature-Diagramm in Definition C.26 zusammengefasst. Sprachparameter für Invarianten können OCL-Invarianten sein, während Werte und Methodenrümpfe als Java-Ausdruck bzw. -BlockStatement konfiguriert werden müssen. Als Beispiel zur syntaktischen und semantischen Behandlung von Stereotypen ist `SingletonClass` definiert, wobei wir annehmen, dass diese Stereotypmenge nur den Stereotyp `<<singleton>>` für Klassen enthält. Die Verwendung dieses Stereotyps führt zu der Forderung, dass die semantische Abbildung die Variante `MapSingletonClass` verwenden muss.

Die semantische Abbildung wird sukzessive entlang der abstrakten Syntax aufgebaut. Definition 7.33 beinhaltet die semantische Abbildung `mCDDefinition` für ein Klassendiagramm insgesamt. `Completeness`-Informationen und der Diagrammname spielen semantisch keine Rolle. Falls Stereotypen definiert sind, werden diese ausgewertet. Ein Stereotyp ist eine Liste von Stereotyp-Namen (mit optionalen Werten), so dass auch mehrere Stereotypen gleichzeitig vergeben werden können. Die Auswertung selbst ist ein Variationspunkt, für den wir keine Varianten angeben und annehmen, dass das System `sm` als Kontext ausreicht. Sollte hier oder in einer der folgenden Definitionen ein unzureichender Kontext bei der praktischen Anwendung festgestellt werden, müssen die entsprechenden Signaturen erweitert werden. Das Klassendiagramm enthält eine *Liste* von Elementen und Invarianten. Aus jedem Element und jeder Invariante folgen Bedingungen für Systeme des Systemmodells. Die Funktion, die für eine Liste von Elementen einzig eine weitere Funktion zur Prüfung jedes Einzelelements aufruft (beispielsweise `mCDElements` enthält nur einen Aufruf für `mCDElement`) ist aus Übersichtsgründen hier und in folgenden semantischen Abbildungen ausgelassen. `mCDElements` und ähnliche Funktionen werden zur besseren Lesbarkeit der Semantik definiert, könnten aber immer durch eine Kombination von `map` (wie in funktionalen Sprachen) und Lambda-Abstraktion ersetzt werden, worunter die Lesbarkeit aber etwas leidet.

Definition SemAbb 7.33 (CD Semantik, Klassendiagramm)

mCDDefinition

Isabelle-Theorie

```
consts mCDDefinitionStereotype ::
  "Stereotype ⇒ SystemModel ⇒ bool"

fun mCDDefinition :: "CDDefinition ⇒ SystemModel ⇒ bool"
where
 "mCDDefinition
```

```
   (CDDefinition complOpt stereoOpt cdName invars cdElems) sm = (
      (if (stereoOpt = None) then True
       else (mCDDefinitionStereotype (the stereoOpt) sm)
      ) ∧
      mCDElements cdElems sm ∧
      mInvariants invars sm
  )"
```

Die Semantik einer Invariante (Definition 7.34), bei der es sich um eine Einbettung handelt, wird direkt an die Semantik der eingebetteten Sprache delegiert. Die Konfiguration ist wie beschrieben in einer Theorie mit dem Namen `ExternalCDSemantics` zu finden (vgl. Abb. 7.90).

Definition SemAbb 7.34 (CD Semantik, Invarianten)

mCDInvariant

Isabelle-Theorie

```
fun mInvariant :: "Invariant ⇒ SystemModel ⇒ bool"
where
 "mInvariant (Invariant ide invexpr) sm =
        ExternalCDSemantics.mInvariantExpression invexpr sm"
```

Elemente in einem Klassendiagramm können nach Definition 7.35 Klassen, Schnittstellen und Assoziationen sein. Enumerationen werden wie bereits erwähnt nicht berücksichtigt. Für jedes Element existiert eine entsprechende semantische Abbildung. Die semantische Abbildung von Assoziationen ist ein Variationspunkt, da auch die Definition von Assoziationen im Systemmodell ein Variationspunkt ist und diese Flexibilität zwar beibehalten werden soll, wir aber eine auf den Systemmodellvarianten basierende Variante angeben wollen.

Definition SemAbb 7.35 (CD Semantik, Klassendiagrammelemente)

mCDElement

Isabelle-Theorie

```
consts mCDAssociation ::
  "CDAssociation ⇒ SystemModel ⇒ bool"

fun mCDElement :: "CDElement ⇒ SystemModel ⇒ bool"
where
  "mCDElement (CDElementCDClass cl) sm = mCDClass cl sm"
| "mCDElement (CDElementCDInterface ci) sm = mCDInterface ci sm"
| "mCDElement (CDElementCDAssociation assoc) sm = mCDAssociation assoc sm"
```

Variante 7.36 definiert eine semantische Abbildung für binäre Assoziationen. Diese Variante verwendet keine Stereotypen, der Assoziationspfeil ist ohne Bedeutung und es wird nur der Assoziationstyp *composition* unterstützt. Gemäß der Funktion `vmCDAssociation` (ab Zeile 20) müssen im Systemmodell eine Assoziation und zwei Klassen existieren, die die linke und rechte Seite der Assoziation bilden (Zeile 27). Die Assoziation hat einen Assoziationsnamen gemäß der Funktion `assocName`, der entweder direkt angegeben ist oder mit Hilfe der Funktion `constName` konstruiert wird. Hierfür sind die Namen der beteiligten Klassen und ihre Rollen verfügbar. Die möglicherweise angegebenen Kardinalitäten an den Assoziationsenden werden durch die Funktion `binaryAssocMultis` berücksichtigt. Handelt es sich um eine Komposition, ist die Assoziation im Systemmodell ebenfalls eine Komposition. Ebenso verhält es sich, wenn die Assoziation abgeleitet oder qualifiziert ist. Die semantische Abbildung des relativ komplexen Konstrukts Assoziation ist deshalb vergleichsweise einfach, da die entsprechenden Varianten im Systemmodell B.23, B.26, B.27 und B.28 verwendet werden können. Die Abhängigkeit der Variante 7.36 von diesen Systemmodellvarianten ist im Feature-Diagramm in Definition 7.50 erfasst.

Variante SemAbb 7.36 (Assoziationen)

mCDAssociation

Isabelle-Theorie

```
theory MapAssociations
imports "$UMLP/def/mc/umlp/cd/semantics/CDSemantics-base"
begin

consts constrName ::
  "QualifiedName ⇒ LiteralsAS.Name option ⇒ QualifiedName
     ⇒ LiteralsAS.Name option ⇒ Name"

fun assocName ::
   "LiteralsAS.Name option ⇒ QualifiedName ⇒ LiteralsAS.Name option
               ⇒ QualifiedName ⇒ LiteralsAS.Name option ⇒ Name"
where
 "assocName assocNameOpt leftRef leftRoleOpt rightRef rightRoleOpt =
    (if (assocNameOpt = None)
     then (constrName leftRef leftRoleOpt rightRef rightRoleOpt)
     else (mName (the assocNameOpt)))"

fun vmCDAssociation :: "CDAssociation ⇒ SystemModel ⇒ bool"
where
 "vmCDAssociation
    (CDAssociation stereoOpt assocType derivedOpt assocNameOpt
                leftStereoOpt leftCardOpt
                leftReference leftQualifierOpt leftRoleOpt
                assocArrow
                rightRoleOpt rightQualifierOpt rightReference
                rightCardOpt rightStereoOpt)  sm =
   (∃ assoc ∈ UASSOC sm . ∃ C1 ∈ UCLASS sm . ∃ C2 ∈ UCLASS sm .
     assoc = Assoc (assocName assocNameOpt leftReference leftRoleOpt
                    rightReference rightRoleOpt) ∧
     C1 = Class (mQualifiedName leftReference) ∧
     C2 = Class (mQualifiedName rightReference) ∧
     classesOf sm assoc = [C1 , C2] ∧
     binaryAssocMultis sm assoc
                  (mCardinalityOpt leftCardOpt)
                  (mCardinalityOpt rightCardOpt) ∧
     (assocType = CDAssociationtypeCOMPOSITION
                  ⟶ compositeAssoc sm assoc) ∧
     (derivedOpt = Some CDAssociationDerivedDERIVED ⟶
              (∃ f . derivedAssoc sm assoc f)) ∧
     (leftQualifierOpt ≠ None
                  ⟶ leftQualifiedAssoc sm assoc
                      (mQualifier (the (leftQualifierOpt)) sm  )) ∧
     (rightQualifierOpt ≠ None
                  ⟶ rightQualifiedAssoc sm assoc
                      (mQualifier (the (rightQualifierOpt)) sm  )))"

defs mCDAssociation-def : "mCDAssociation == vmCDAssociation"

```

```
49  end
```

Die Definition 7.37 enthält die semantische Abbildung für Klassen des Klassendiagramms (ab Zeile 19) und ihre Variationspunkte. Sie besagt, dass es eine korrespondierende Klasse im System geben muss in deren Kontext die semantischen Abbildungen für Modifier, Superklassen, Methoden, Konstruktoren und Attribute erfüllt werden. `mImplInterface` fordert, dass es für ein Interface der betrachteten Klassen eine entsprechende Klasse im System gibt, die ein Interface und Superklasse ist. Über die Variante B.36 (TypeSafeOps) des Systemmodells ist gewährleistet, dass alle Operationen des Interfaces in der betrachteten Klasse existieren. Die Abhängigkeit zur Systemmodellvariante ist in Definition 7.50 erfasst. Für den Variationspunkt `mCDConstructor` wird in dieser Semantik keine Variante definiert.

Definition SemAbb 7.37 (CD Semantik, Klassen)

```
                                 mCDClass
                                                            Isabelle-Theorie
1   consts mSuperClass ::
2     "ReferenceType list ⇒ iCLASS ⇒ SystemModel ⇒ bool"
3
4   consts mCDClassStereotypes :: "Stereotype ⇒ iCLASS ⇒ SystemModel ⇒ bool"
5
6   consts mCDConstructor :: "CDConstructor ⇒ iCLASS ⇒ SystemModel ⇒ bool"
7
8   consts mCDClassModifier :: "Modifier ⇒ iCLASS ⇒ SystemModel ⇒ bool"
9
10  fun mImplInterface :: "ReferenceType ⇒ iCLASS ⇒ SystemModel ⇒ bool"
11  where
12   "mImplInterface tp C sm = (
13      ∃ I ∈ UCLASS sm .
14        I = toClass (mReferenceType tp) ∧
15        isInterface sm I ∧
16        sub sm C I
17   )"
18
19  fun mCDClass :: "CDClass ⇒ SystemModel ⇒ bool"
20  where
21   "mCDClass (CDClass complOpt modifier
22                       name tp
23                       supercls interfs
24                       meths constrs atts) sm = (
25      ∃ C ∈ UCLASS sm .
26        C = Class (mName name) ∧
27        mCDClassModifier modifier C sm ∧
28        mSuperClass supercls C sm  ∧
29        mImplInterfaces interfs C sm ∧
30        mCDMethods meths C  sm ∧
31        mCDConstructors constrs C sm ∧
32        mCDAttributes atts C  sm
33   )"
```

Für die semantische Abbildung von Superklassen einer Klasse geben wir zwei Varianten an. In der ersten Variante 7.38 ist die betrachtete Klasse jeweils Subklasse der Superklasse im System. Ist mehr als eine Superklasse angegeben, muss im Systemmodell also Mehrfachvererbung erlaubt sein (siehe Feature-Diagramm 7.50).

Variante SemAbb 7.38 (Superklassen, direkt)

mSuperClassDirect

Isabelle-Theorie

```
theory MapSuperclassesDirect
imports "$UMLP/def/mc/umlp/cd/semantics/CDSemantics-base"
begin

fun mSuperClassDirect :: "ReferenceType list ⇒ iCLASS ⇒ SystemModel ⇒ bool"
where
  "mSuperClassDirect [] c sm = True"
| "mSuperClassDirect (supc#xs) c sm = (
    sub sm c (toClass (mReferenceType supc)) ∧ mSuperClassDirect xs c sm
  )"

defs mSuperClass-def : "mSuperClass == mSuperClassDirect "

end
```

In der zweiten Variante 7.39 der Abbildung von Superklassen wird nur eine Superklasse auch direkt in Subklassenbeziehung gesetzt (Zeile 29). Weitere Superklassen werden über einen Delegationsmechanismus behandelt, der in der Funktion `delegate` skizziert wird. Hiernach muss es im System ein Interface geben, das beide Klassen implementieren (Zeile 9: hierzu in Subklassenrelation stehen, strikte einfache Vererbung ist in dieser Variante also auch ausgeschlossen, siehe Definition 7.50). Alle Operationen der Superklasse müssen im Interface und damit auch in der Subklasse vorhanden sein (Zeilen 10-11). Die Delegation selbst ist abstrakt spezifiziert: Sobald ein Methodenaufruf bei der Subklasse eintrifft, der eine Methode des Interface betrifft, erhält ein zweites Objekt der Superklasse im nächsten Schritt `s'` diesen Aufruf. Rückgaben werden über die Subklasse an den ursprünglichen Aufrufer weitergeleitet. Wie genau der Nachrichtenaustausch von Statten geht wird nicht spezifiziert und stellt damit eine Implementierungsfreiheit dar.

Variante SemAbb 7.39 (Superklassen, Delegation)

mSuperClassDelegate

Isabelle-Theorie

```
1   theory MapSuperclassesDelegate
2   imports "$UMLP/def/mc/umlp/cd/semantics/CDSemantics-base"
3   begin
4
5   fun delegate :: "iCLASS ⇒ iCLASS ⇒ SystemModel ⇒ bool"
6   where
7    "delegate subc supc sm = (
8      ∃ I ∈ UCLASS sm . isInterface sm I ∧
9       sub sm subc I ∧ sub sm supc I ∧
10      (∀ opn . classOf sm opn = supc ⟶ (
11        (∃ opn1 . classOf sm opn1 = I ∧ overrides sm opn opn1) ∧
12        (∀ s ∈ reachable sm . ∀ s' ∈ Delta (SYSSTS sm) s .
13          (∀ oid1 ∈ Class.oids sm subc .
14            (∀ oid2 ∈ Class.oids sm supc .
15                (∀ caller param th .
16                  hasMsg s oid1
17                   (caller, Operation.nameOf opn, param, oid1, th)  ⟶
18                  hasMsg s' oid2
19                   (oid1, Operation.nameOf opn, param, oid2, th) ∧
20                  hasMsg s oid1
```

```
                 (oid2, Operation.nameOf opn, param, oid1, th) ⟶
                hasMsg s' caller
                 (oid1, Operation.nameOf opn, param, caller, th)
              ))))))))"

fun mSuperClassDelegate :: "ReferenceType list ⇒ iCLASS ⇒ SystemModel ⇒ bool"
where
  "mSuperClassDelegate [] c sm = True"
| "mSuperClassDelegate [supc] c sm = sub sm c (toClass (mReferenceType supc))"
| "mSuperClassDelegate (supc#supc2#xs) c sm = (
      delegate c (toClass (mReferenceType supc)) sm ∧
      (mSuperClassDelegate (supc2#xs) c sm))"

defs mSuperClass-def : "mSuperClass == mSuperClassDelegate"

end
```

Die Semantik eines Modifier für eine Klasse wird in Variante 7.40 definiert. Modifier enthalten auch Stereotypen, die exemplarisch in Variante 7.41 behandelt werden. Sichtbarkeiten werden wie bereits erwähnt nicht in der semantischen Abbildung berücksichtigt. Finale Klassen besitzen laut `mClassFinal` keine Unterklassen (Zeile 4). Abstrakte Klassen können nicht instantiiert werden, für sie gibt es also keine Objektidentifikatoren (Zeile 8). Für andere Modifier ist für Klassen keine Bedeutung definiert.

Variante SemAbb 7.40 (Modifier für Klassen)

```
mCDClassModifier
                                                                Isabelle-Theorie
1   fun mClassFinal :: "iCLASS ⇒ SystemModel ⇒ bool"
2   where
3    "mClassFinal C sm =
4       (∀ C2 ∈ UCLASS sm . sub sm C C2 ⟶ C = C2)"
5
6   fun mClassAbstract :: "iCLASS ⇒ SystemModel ⇒ bool"
7   where
8    "mClassAbstract C sm = (Class.oids sm C = {})"
9
10  fun vmCDClassModifier :: "Modifier ⇒ iCLASS ⇒ SystemModel ⇒ bool"
11  where
12   "vmCDClassModifier (Modifier stereoOpt modiVals) C sm = (
13     (if (stereoOpt = None) then True
14      else (mCDClassStereotypes (the stereoOpt) C sm)) ∧
15     (getModifiers ModifierValFINAL modiVals ≠ [] ⟶ mClassFinal C sm) ∧
16     (getModifiers ModifierValABSTRACT modiVals ≠ [] ⟶ mClassAbstract C sm))"
17
18  defs mCDClassModifier-def : "mCDClassModifier == vmCDClassModifier"
```

Variante 7.41 dient schließlich dazu, die semantische Behandlung von Stereotypen zu verdeutlichen. Die Existenz des Stereotyps `<<singleton>>` impliziert (Zeile 14), dass zu jeder Zeit höchstens ein Objekt der Klasse existiert (vgl. Definition `isSingleton` in Definition B.9 auf Seite 250).

Variante SemAbb 7.41 (Stereotyp Singleton)

mSingleton

Isabelle-Theorie

```
fun has :: "StereoValue list ⇒ LiteralsAS.Name ⇒ bool"
where
 "has [] m = False"
|"has ((StereoValue n strOpt)#xs) m =
   (if (m = n) then True else (has xs m))"

fun stereoValues :: "Stereotype ⇒ StereoValue list"
where
 "stereoValues (Stereotype xs) = xs"

fun mSingleton :: "Stereotype ⇒ iCLASS ⇒ SystemModel ⇒ bool"
where
 "mSingleton stereo C sm = (
    (has (stereoValues stereo) ''singleton'') ⟶
       isSingleton sm C)"

defs mCDClassStereotypes-def : "mCDClassStereotypes == mSingleton"
```

Die semantische Abbildung für Interfaces (Definition 7.42) ähnelt der Abbildung für Klassen. Beide werden auf Klassen im Systemmodell abgebildet, da im Systemmodell keine explizite Struktur für Interfaces vorgesehen ist. Stattdessen erklärt das Prädikat `isInterface` (Definition B.32 auf Seite 262), dass es sich bei der Klasse um ein Interface handeln muss. Die Oberinterfaces werden mit Hilfe der in Definition 7.37 eingeführten Funktion `mImplInterfaces` behandelt. Methoden dürfen in Interfaces keine Rumpf haben. Dies wird syntaktisch durch eine entsprechende Überprüfung in Definition C.23 auf Seite 310 sichergestellt. Varianten für Modifier und darin enthaltene Stereotypen für Interfaces werden nicht definiert. Modifier zur Einschränkung von Sichtbarkeiten etc. dürfen bei Interfaces nicht verwendet werden. Interfaces müssen immer öffentlich sein und dürfen weder als statisch oder final deklariert werden.

Definition SemAbb 7.42 (CD Semantik, Interfaces)

mCDInterface

Isabelle-Theorie

```
consts mCDInterfaceStereotypes ::
  "Stereotype ⇒ iCLASS ⇒ SystemModel ⇒ bool"

consts mCDInterfaceModifier :: "Modifier ⇒ iCLASS ⇒ SystemModel ⇒ bool"

fun mCDInterface :: "CDInterface ⇒ SystemModel ⇒ bool"
where
 "mCDInterface (CDInterface complOpt modifier
                            name  None
                            superifs
                            meths atts) sm = (
   ∃ C ∈ UCLASS sm .
     C = Class (mName name) ∧
     isInterface sm C ∧
     mCDInterfaceModifier modifier C sm ∧
     mImplInterfaces superifs C sm ∧
     mCDMethods meths C sm ∧
     mCDAttributes atts C sm
  )"
```

Attribute (in Klassen oder Schnittstellen) im Klassendiagramm werden direkt auf Attribute von Klassen im Systemmodell umgesetzt, falls sie nicht statisch sind, siehe Definition 7.43 ab Zeile 17. Für statische Attribute ist ein Variationspunkt `mStaticAttribute` eingeführt, für den in Variante 7.45 eine Realisierung vorgeschlagen wird. Ansonsten müssen Name und Typ des Attributs mit einem Attribut der entsprechenden Klassen übereinstimmen. Ist ein initialer Wert angegeben, so muss in einem neu erzeugten Objekt der Klasse dieser Wert für das Attribut im Datenzustand abgelegt sein (Zeile 24). Stereotypen und Modifier werden in entsprechenden Varianten behandelt. Wir geben eine Variante für Modifier an, die finale und abgeleitete Attribute berücksichtigt. Attribute mit dem Modifier „nur-lesen“ werden wie finale Attribute behandelt.

Definition SemAbb 7.43 (CD Semantik, Attribute)

mCDAttribute

Isabelle-Theorie

```
1  consts mCDAttributeStereotypes ::
2     "Stereotype ⇒ iCLASS ⇒ iVAR ⇒ SystemModel ⇒ bool"
3
4  consts mCDAttributeModifier ::
5     "Modifier ⇒ iCLASS ⇒ iVAR ⇒ SystemModel ⇒ bool"
6
7  consts mStaticAttribute :: "CDAttribute ⇒ iCLASS ⇒ SystemModel ⇒ bool"
8
9  fun mCDAttribute :: "CDAttribute ⇒ iCLASS ⇒ SystemModel ⇒ bool"
10 where
11  "mCDAttribute
12    (CDAttribute (Modifier stero valList) type name valOpt) C sm  = (
13     (getModifiers ModifierValSTATIC valList ≠ [] ∧
14      mStaticAttribute
15       (CDAttribute (Modifier stero valList) type name valOpt) C sm)
16    ∨ (getModifiers ModifierValSTATIC valList = [] ∧
17      (∃ at ∈ attr sm C .
18        vname at = mName name ∧
19        vtype sm at = mType type ∧
20        mCDAttributeModifier (Modifier stero valList) C at sm ∧
21        (if (valOpt = None) then True else
22         (∀ t ∈ TRACES sm . ∀ s .
23           (∀ oid . VOid oid ∈ CAR sm (TO C) ∧ (∀ th ∈ UTHREAD sm . ∃ va .
24             created sm t s oid ⟶ the (val sm (dsOf s, oid, at))
25                               = mValue (the valOpt) s va oid th sm))))))
26   )"
```

Gemäß Funktion `mAttrFinal` aus Variante 7.44 zeichnet sich ein finales Attribut dadurch aus, dass der Wert des Attributs nach Erzeugung nicht mehr geändert wird, also in allen Folgezuständen gleich ist. Für abgeleitete Attribute gilt, dass der Wert des Attributs stets dem Wert einer hier nicht weiter definierten Funktion `f` folgt.

Variante SemAbb 7.44 (Modifier für Attribute)

mCDAttributeModifier

Isabelle-Theorie

```
fun mAttrFinal :: "iCLASS ⇒ iVAR ⇒ SystemModel ⇒ bool"
where
 "mAttrFinal C at sm =
  (∀ t ∈ TRACES sm . ∀ s . ∀ v ∈ (CAR sm (TO C)∩{VOid VNil}) .
    created sm t s (toOid v) ⟶
      (∀ s' . following sm t s s' ⟶
         the (val sm (dsOf s', (toOid v), at))
           = the (val sm (dsOf s, (toOid v), at))))"

fun mAttrDerived :: "iCLASS ⇒ iVAR ⇒ SystemModel ⇒ bool"
where
 "mAttrDerived C at sm =
  (∀ v ∈ (CAR sm (TO C)∩{VOid VNil}) .
    ( ∃ f . ∀ ds ∈ DataStore sm .
       (toOid v ∈ DataStore1.oids ds ⟶
         the (val sm (ds, (toOid v), at)) = f sm ds)))"

fun vmCDAttributeModifier :: "Modifier ⇒ iCLASS ⇒ iVAR ⇒ SystemModel ⇒ bool"
where
 "vmCDAttributeModifier (Modifier stereoOpt modiVals) C v sm = (
   (if (stereoOpt = None) then True
    else (mCDAttributeStereotypes (the stereoOpt) C v sm)) ∧
   (getModifiers ModifierValFINAL modiVals ≠ [] ⟶ mAttrFinal C v sm) ∧
   (getModifiers ModifierValDERIVED modiVals ≠ [] ⟶ mAttrDerived C v sm) ∧
   (getModifiers ModifierValREADONLY modiVals ≠ [] ⟶ mAttrFinal C  v sm))"

defs mCDAttributeModifier-def : "mCDAttributeModifier == vmCDAttributeModifier"
```

Die Behandlung statischer Attribute ist aus Übersichtlichkeitsgründen in eine eigene Variante 7.45 ausgelagert. Für die semantische Abbildung verwenden wir die Systemmodellvariante B.22 (StaticAttr) auf Seite 256. Das Attribut der Klasse ist also in der speziellen Klasse `StaticC` verfügbar und wird gegebenenfalls initialisiert.

Variante SemAbb 7.45 (Statische Attribute)

mStaticAttribute

Isabelle-Theorie

```
fun vmStaticAttribute ::
   "CDAttribute ⇒ iCLASS ⇒ SystemModel ⇒ bool"
where
 "vmStaticAttribute (CDAttribute modif type name valOpt) C sm = (
    (∃ at ∈ StaticAttr sm .
      vname at = mName name ∧
      vtype sm at = mType type ∧
      mCDAttributeModifier modif (StaticC sm) at sm ∧
      (if (valOpt = None) then True else
       (∀ t ∈ TRACES sm . ∀ s .
         (created sm t s (StaticOid sm) ⟶
           (∀ th ∈ UTHREAD sm . ∀ va .
            the (val sm (dsOf s, StaticOid sm, at))
                = mValue (the valOpt) s va (StaticOid sm) th sm)))))
 )"

defs mStaticAttribute-def : "mStaticAttribute == vmStaticAttribute"
```

Die Semantik für Methoden ist in Definition 7.46 angegeben. Es wird die Existenz einer passenden Operation im Kontext der übergebenen Klasse gefordert; Name, Rückgabetyp und Parametertypen müssen übereinstimmen (ab Zeile 24). Die Behandlung der Modifier ist wie bei Attributen ein Variationspunkt und eine Realisierung ist in Variante 7.47 enthalten. Statische Methoden werden gesondert in Variante 7.48 behandelt. Ab Zeile 30 ist kodiert, dass falls der Methodenrumpf existiert, es im Systemmodell eine passende Methode geben muss, die die Operation implementiert. Die eigentliche Methodenausführung kann mit Hilfe des Variationspunkts `mMethodExec` präzisiert werden. Stark vereinfacht interpretieren wir in Variante 7.49 alle Methoden als einfache Abfragen im Systemmodell. Diese rudimentäre Behandlung der Methodenausführung ist aber ausreichend, um die konkreten Verifikationsbeispiele im nächsten Kapitel durchzuführen.

Definition SemAbb 7.46 (CD Semantik, Methoden)

mCDAttribute

Isabelle-Theorie

```
1  fun mCDParameter :: "CDParameter ⇒ iTYPE"
2  where
3   "mCDParameter (CDParameter type name) = mType type"
4
5  consts mCDMethodStereotypes ::
6    "Stereotype ⇒ iCLASS ⇒ iOPN ⇒ SystemModel ⇒ bool"
7
8  consts mMethodExec :: "Body ⇒ Name ⇒ CDParameter list ⇒ SystemModel ⇒ bool"
9
10 consts mCDMethodModifier :: "Modifier ⇒ iCLASS ⇒ iOPN ⇒ SystemModel ⇒ bool"
11
12 consts mStaticMethod :: "CDMethod ⇒ iCLASS ⇒ SystemModel ⇒ bool"
13
14 fun mCDMethod :: "CDMethod ⇒ iCLASS ⇒ SystemModel ⇒ bool"
15 where
16  "mCDMethod (CDMethod (Modifier stereo valList) typParams
17                            retType name parList
18                            exceptionList bodyOpt) C sm  = (
19    (getModifiers ModifierValSTATIC valList ≠ [] ∧
20     mStaticMethod (CDMethod (Modifier stereo valList) typParams
21                            retType name parList
22                            exceptionList bodyOpt) C sm)
23   ∨  (getModifiers ModifierValSTATIC valList = [] ∧
24    (∃ opn ∈ UOPN sm .
25      classOf sm opn = C ∧
26      mCDMethodModifier (Modifier stereo valList) C opn sm  ∧
27      Operation.resType sm opn = mReturnType retType ∧
28      Operation.parTypes sm opn = mCDParameters parList ∧
29      Operation.nameOf opn = mName name ∧
30      (if bodyOpt ≠ None
31       then (
32         (∃ m ∈ UMETH sm .
33          the (impl sm opn) = m) ∧
34          mMethodExec (the bodyOpt) (mName name) parList sm)
35       else True)))
36  )"
```

Finale und abstrakte Methoden werden durch Variante 7.47 unterstützt. Gemäß Funktion `mMethodFinal` wird die Operation zu einer finalen Methode nicht mehr überschrieben und eine Systemmodellmethode für diese Operation existiert. Abstrakte Methoden sind dagegen nicht implementiert. Die sie definierende Klasse ist nicht instantiierbar. Konkrete Varianten für Stereotypen werden nicht betrachtet.

Variante SemAbb 7.47 (Modifier für Methoden)

mCDMethodModifier

Isabelle-Theorie

```
fun mMethodFinal :: "iCLASS ⇒ iOPN ⇒ SystemModel ⇒ bool"
where
 "mMethodFinal C opn sm = (
    impl sm opn ≠ None ∧
    (∀ op2 ∈ UOPN sm .
      overrides sm opn op2 ⟶ impl sm op2 = impl sm opn)
 )"

fun mMethodAbstract :: "iCLASS ⇒ iOPN ⇒ SystemModel ⇒ bool"
where
 "mMethodAbstract C opn sm = (
    Class.oids sm C = {} ∧
    impl sm opn = None
  )"

fun vmCDMethodModifier :: "Modifier ⇒ iCLASS ⇒ iOPN ⇒ SystemModel ⇒ bool"
where
 "vmCDMethodModifier (Modifier stereoOpt modiVals) C opn sm = (
    (if (stereoOpt = None) then True
     else (mCDMethodStereotypes (the stereoOpt) C opn sm)) ∧
    (getModifiers ModifierValFINAL modiVals ≠ []
        ⟶ mMethodFinal C opn sm) ∧
    (getModifiers ModifierValABSTRACT modiVals ≠ []
        ⟶ mMethodAbstract C opn sm)
  )"

defs mCDMethodModifier-def : "mCDMethodModifier == vmCDMethodModifier"
```

Statische Methoden werden ähnlich wie normale Methoden übersetzt. Sie gehören jedoch nach Systemmodellvariante B.37 zur Klasse `StaticC` und sind damit global verfügbar.

Variante SemAbb 7.48 (Statische Methoden)

mStaticMethod

Isabelle-Theorie

```
fun vmStaticMethod ::
  "CDMethod ⇒ iCLASS ⇒ SystemModel ⇒ bool"
where
 "vmStaticMethod (CDMethod modif typParams
                           retType name parList
                           exceptionList bodyOpt) C sm = (
   (∃ opn ∈ StaticOpn sm .
     mCDMethodModifier modif (StaticC sm) opn sm ∧
     Operation.resType sm opn = mReturnType retType ∧
     Operation.parTypes sm opn = mCDParameters parList ∧
     Operation.nameOf opn = mName name ∧
     (if bodyOpt ≠ None
      then (
        (∃ m ∈ UMETH sm .
         the (impl sm opn) = m) ∧
         mMethodExec (the bodyOpt) (mName name) parList sm)
      else True))
 )"
```

```

defs mStaticMethod-def : "mStaticMethod == vmStaticMethod"
```

Wie bereits erwähnt betrachten wir die Methodenausführung in Variante 7.49 vereinfacht als Abfrage über dem System. Dies funktioniert nur dann, wenn der Methodenrumpf nur aus einer Rückgabe besteht (siehe Funktion `getExpr`). Die Namen der Methodenparameter werden mit `toNameList` extrahiert. Unter dem Methodennamen wird in Funktion `vmMethodExec` eine Abfrage im Systemmodell abgelegt (wir setzen damit Variante B.47 von Seite 271 voraus). `buildQuery` erstellt diese Abfrage als Funktion, die für einen gegebenen Systemzustand, Objektidentifikator, Thread, eine Liste von Werten und eine Variablenbelegung, in der die Parameternamen an Werte gebunden werden, einen Wert zurück liefert. Dieser Wert entspricht der Auswertung des übergebenen Ausdrucks `expr`.

Variante SemAbb 7.49 (Vereinfachte Methodenausführung als Abfrage)

mMethodExec

Isabelle-Theorie

```
theory MapMethod
imports "$UMLP/def/mc/umlp/cd/semantics/CDSemantics-base"
begin

fun extendWithParams :: "iVarAssign ⇒ Name list ⇒ iVAL list ⇒ iVarAssign"
where
  "extendWithParams va [] [] = va"
| "extendWithParams va (x#xs) (y#ys) =
     (extendWithParams va xs ys)(x ↦ y)"

constdefs "buildQuery expr parList ==
   (λ s va oid th params sm .
    mExpression expr s (extendWithParams va parList params) oid th sm)"

fun getExpr :: "Body ⇒ Expression"
where
 "getExpr (BlockStatement
          [StatementReturnStatement (ReturnStatement (Some expr))]) = expr"

fun toNameList :: "CDParameter list ⇒ Base.Name list"
where
  "toNameList [] = []"
| "toNameList ((CDParameter type n)#xs) = (mName n)#(toNameList xs)"

fun vmMethodExec :: "Body ⇒ Name ⇒ CDParameter list ⇒ SystemModel ⇒ bool"
where
 "vmMethodExec body name parList sm =  (
     query sm name = buildQuery (getExpr body) (toNameList parList))"

defs mMethodExec-def : "mMethodExec == vmMethodExec"

end
```

In Definition 7.50 sehen wir die Variabilität der semantischen Abbildung von Klassendiagrammen als Feature-Diagramm. Die Varianten zur Behandlung der Modifier im Klassendiagramm sind in der Theorie `MapModifiers` zusammengefasst.

Definition Variabilität 7.50 (Semantik CD)

CDSemantics

Feature-Diagramm

```
package mc.umlp.cd.semantics;

featurediagram CDSemantics {

  CDSemantics = vMapSuperClasses? & vStereotypes? &
                vMapAssociations? & vMapMethod? &
                vMapConstructor? & vMapModifiers?;

  vStereotypes = MapSingletonClass;

  vMapSuperClasses = MapSuperclassesDirect ^ MapSuperclassesDelegate;

  vMapAssociations = MapAssociations;

  vMapMethod = MapMethod;

  vMapConstructor = MapConstructor;

  vMapModifiers = MapModifiers;

  constraints {
      CDSemantics -> SystemModel.vControl.TypeSafeOps;
      CDSemantics -> CDSyntax.CDConstr.RestrictCD;
      vMapSuperClasses.MapSuperclassesDirect ->
           !SystemModel.vObject.StrictSingleInheritance;
      vMapSuperClasses.MapSuperclassesDirect ->
           !SystemModel.vControl.ClassSingleInheritance;
      vMapSuperClasses.MapSuperclassesDelegate ->
           !SystemModel.vObject.StrictSingleInheritance;
      vMapAssociations.MapAssociations ->
          (SystemModel.vData.QualifiedAssociation &
           SystemModel.vData.CompositeAssociation &
           SystemModel.vData.DerivedAssociation );
      vMapModifiers.MapModifiers ->
            (SystemModel.vData.StaticAttr &
             SystemModel.vControl.StaticOpn);
   }
}
```

Die gesamte Variabilität der UML/P-Diagrammart Klassendiagramm ist in Definition 7.51 dargestellt.

Definition Variabilität 7.51 (CD)

CD

Feature-Diagramm

```
package mc.umlp.cd;

<<lang,cu>> featurediagram CD {

 CD = *mc.umlp.cd.syntax.CDSyntax &
      <<sm>> *mc.umlp.cd.semantics.CDSemantics &
      <<sd>> *systemmodel.SystemModel;
}
```

Die Semantik eines Klassendiagramms beschreibt eine Menge von Systemen, die die strukturellen Forderungen des Klassendiagramms erfüllen. Ein ähnlicher Ansatz wird beispielsweise in [Szl06] verfolgt, bei dem konzeptuelle Klassendiagramme ohne Attribute oder Methoden und einen eingeschränkte Menge von Beziehungen zwischen Klassen betrachtet werden. Ansonsten konzentrieren sich Arbeiten zur Semantik von Klassendiagrammen meist auf spezifische Aspekte wie Assoziationen (z.B. [DD06]) oder die Erfüllbarkeit von angegebenen Kardinalitäten (z.B. [MB07]). Eine solche Detailtiefe wie in den erwähnten Arbeiten kann unsere Arbeit nicht leisten. Die Semantik ist aber offen für Anpassungen und weitere Spezialisierung. Der Semantik einer Strukturbeschreibung wird generell weniger Bedeutung zugemessen als der Semantik von Verhaltensbeschreibungen. Zum Teil wird die abstrakte Syntax von Klassendiagrammen direkt als struktureller Kontext für andere Diagrammarten verwendet. In [KGKZ09] beispielsweise wird ein Klassendiagramm als Typgraph und als Bedingungen an Objektstrukturen interpretiert, die dazu dienen, zu prüfen, ob Graphtransformationen ausgeführt werden können. Wir definieren die Semantik von Klassendiagrammen vollständig unabhängig von anderen Diagrammarten.

7.5 Objektdiagramme

Die MontiCore-Grammatik für UML/P-Objektdiagramme ist in Definition C.27, die abstrakte Syntax in Definition C.28 in Anhang C.4 ab Seite 313 zu finden. Die syntaktische Variabilität in Definition C.29 spiegelt nur die Möglichkeiten der Spracheinbettung wider. Kontextbedingungen sowie Stereotypen werden nicht betrachtet.

UML/P-Objektdiagramme machen Aussagen über einen Systemzustand, genauer den Datenzustand eines Zustands. Die semantische Abbildung für ein Objektdiagramm in Definition 7.52 erfolgt daher im Kontext eines Systemzustands. Für diesen Zustand wird sukzessive geprüft, ob die spezifizierten Eigenschaften (welche Objekte sind existent, welche Attributbelegung ist angegeben, welche Links zwischen Objekten sind vorhanden) vom angegeben Zustand erfüllt werden. Die semantische Abbildung ist relativ unkompliziert. Da Objektdiagramme aber in einem gewissen Kontext verwendet werden müssen, ergeben sich aus ihrer methodischen Verwendung heraus verschiedene Möglichkeiten den Kontext zu belegen. Eine Invariante kann mit einem Objektdiagramm beschrieben werden, wenn der Kontext allquantifiziert wird, so dass das Objektdiagramm für jeden (erreichbaren) Zustand erfüllt werden muss. Ein Objektdiagramm kann aber auch eine Methodenvorbedingung oder -nachbedingung spezifizieren. Ein gültiger Systemzustand für die Vorbedingung und die Ausführung der Methode implizieren dann einen gültigen Systemzustand für die Nachbedingung. Eine enge Integration mit der OCL/P ist daher auch in [Rum04b] vorgesehen und beschrieben. Unsere Darstellung beschränkt sich aber auf die Interpretation eines Objektdiagramms bei gegebenem Systemkontext.

In Definition 7.52 sehen wir, dass neben dem Zustand eine Variablenbelegung `objmap` als weiterer Kontext aufgebaut wird. In dieser Variablenbelegung werden alle im Objektdiagramm verwendeten Objektnamen auf Objektidentifikatoren im System mit Hilfe der deklarierten Funktion `mObjName` abgebildet. Diese Abbildung ist ein Variationspunkt. Ein einfache Realisierung ist in Variante 7.53 definiert. Eine semantische Abbildung für konkrete Stereotypen wurde nicht definiert.

Definition SemAbb 7.52 (OD Semantik, Objektdiagramm)

mODDefinition

Isabelle-Theorie

```
consts mObjName :: "LiteralsAS.Name ⇒ iOID"

consts objNames2obj :: "ODElement list ⇒ iVarAssign ⇒ iVarAssign"

consts mODDefinitionStereotype :: "Stereotype ⇒ SystemModel ⇒ bool"

fun mODDefinition :: "ODDefinition ⇒ iSTATE ⇒ SystemModel ⇒ bool"
where
 "mODDefinition
    (ODDefinition completeness stereoOpt odname odelements invariants) s sm = (
     let objmap = objNames2obj odelements empty in
      mODElements odelements s objmap sm ∧
      mInvariants invariants s objmap sm ∧
     (if stereoOpt = None then True
      else mODDefinitionStereotype (the stereoOpt) sm))"
```

Die Funktion `vmObjName` in Variante 7.53 stellt eine sehr einfache Verbindung zwischen Objektnamen und -identifikatoren her, indem aus dem Objektnamen direkt ein Identifikator konstruiert wird. Die Variablenbelegung, die mit `vobjNames2obj` konstruiert werden kann, dient zum Auflösung von Querverweisen im Objektdiagramm (beispielsweise kann ein Attributwert auf den Namen eines anderen Objekts verweisen). Wir gehen vereinfachend von einer eindeutigen Zuordnung von Objektnamen zu realen Objekten im System aus. Eine Erweiterung dieser Definition ist jedoch leicht möglich.

Variante SemAbb 7.53 (Abbildung von Objektnamen auf Objekte)

MapObject

Isabelle-Theorie

```
theory MapObject
imports "$UMLP/def/mc/umlp/od/semantics/ODSemantics-base"
begin

fun vmObjName :: "LiteralsAS.Name ⇒ iOID"
where
 "vmObjName name = (Oid (mName name))"

defs mObjName-def : "mObjName == vmObjName"

fun vobjNames2obj :: "ODElement list ⇒ iVarAssign ⇒ iVarAssign"
where
  "vobjNames2obj [] vars = vars"
| "vobjNames2obj ((ODElementODLink l)#xs) vars = vobjNames2obj xs vars"
| "vobjNames2obj ((ODElementODObject (ODObject complOpt modifier
                    refOpt (Some name) atts))#xs) vars =
    (vobjNames2obj xs (vars(mName name ↦ VOid (mObjName name))))"

defs objNames2obj-def : "objNames2obj == vobjNames2obj"

end
```

Die semantische Abbildung für Invarianten wird wie üblich in ein separate Theorie ausgelagert und gemäß der eingebetteten Sprache konfiguriert. Die erforderliche Signatur ist aus Definition 7.54 erkennbar.

Definition SemAbb 7.54 (OD Semantik, Invarianten)

mODInvariant

Isabelle-Theorie

```
fun mInvariant ::
  "Invariant ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ bool"
where
 "mInvariant (Invariant ido invexpr) s vars sm =
    (ExternalODSemantics.mInvariantExpression invexpr s vars sm)"
```

Ein Element im Objektdiagramm ist entweder ein Objekt oder ein Link zwischen Objekten. Definition 7.55 delegiert daher an die entsprechenden Funktionen, wobei die Abbildung für Links ein Variationspunkt ist. Eine mögliche Instantiierung ist in Variante 7.56 angegeben.

Definition SemAbb 7.55 (OD Semantik, Objektdiagrammelemente)

mODElement

Isabelle-Theorie

```
consts mODLink ::
  "ODLink ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ bool"

fun mODElement :: "ODElement ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ bool"
where
  "mODElement (ODElementODObject x) s vars sm = mODObject x s vars sm"
  | "mODElement (ODElementODLink x) s vars sm = mODLink x s vars sm"
```

Gemäß Variante 7.56 muss es zu jedem Link im Objektdiagramm eine Assoziation im System geben und zusätzlich zwei Objektidentifikatoren (Zeile 24). Die Assoziation ist aus dem Namen oder den Rollen- und Objektnamen eindeutig bestimmt, hierfür soll `idAssoc` sorgen, wobei wir keine konkrete Variante der Funktion angeben. Für den angegebenen Zustand existiert ein Link zwischen diesen Objektreferenzen, da jedes Objekt jeweils das andere Objekt als Ziel enthält (Funktion `destinations` aus der Systemmodellvariante B.23). Ist der Link links oder rechts qualifiziert, dann wird mit der Funktion `mQualifierVal` der Wert des Qualifikators `v` bestimmt. Der qualifizierte Link `(oid1,v,oid2)` muss dann im aktuellen Zustand für die Assoziation im System vorhanden sein. Falls Stereotypen vorhanden sind, werden diese mit `mODLinkStereotypes` behandelt. Eine Realisierung dieses Variationspunkts für konkrete Stereotypen ist nicht angegeben.

Variante SemAbb 7.56 (Links im Objektdiagramm)

MapLink

Isabelle-Theorie

```
1  theory MapLink
2  imports "$UMLP/def/mc/umlp/od/semantics/ODSemantics-base"
3  begin
4
5  consts mODLinkStereotype :: "Stereotype ⇒ SystemModel ⇒ bool"
6
7  fun mQualifierVal ::
8   "Value ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ iVAL"
9  where
```

```
  "mQualifierVal v s vars oid th sm =
    ExternalODSemantics.mValue v s vars oid th sm"

consts idAssoc ::
  "iASSOC ⇒
   LiteralsAS.Name option ⇒ LiteralsAS.Name list ⇒ LiteralsAS.Name
   option ⇒ LiteralsAS.Name list ⇒ LiteralsAS.Name option
    ⇒ bool"

fun vmODLink :: "ODLink ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ bool"
where
"vmODLink (ODLink stereoOpt linkType derivedOpt nameOpt leftRefs
      leftQualifierOpt leftRoleOpt linkarrow rightRoleOpt rightQualifierOpt
      rightRefs) s vars sm = (
  ∃ assoc ∈ UASSOC sm . ∃ oid1 ∈ UOID sm . ∃ oid2 ∈ UOID sm .
  idAssoc assoc nameOpt leftRefs leftRoleOpt rightRefs rightRoleOpt ∧
  oid1 = mObjName (leftRefs!(length leftRefs+(-1))) ∧
  oid1 ∈ destinations sm (assoc, dsOf s, oid2) ∧
  oid2 = mObjName (rightRefs!(length rightRefs+(-1))) ∧
  oid2 ∈ destinations sm (assoc, dsOf s, oid1) ∧
  (leftQualifierOpt ≠ None
       ⟶ (∃ th .
         (oid1, mQualifierVal (the leftQualifierOpt) s vars oid2 th sm,oid2)
           ∈ qualifiedRelOf sm (assoc,dsOf s))) ∧
  (rightQualifierOpt ≠ None
       ⟶ (∃ th .
         (oid1,mQualifierVal (the rightQualifierOpt) s vars oid1 th sm,oid2)
           ∈ qualifiedRelOf sm (assoc,dsOf s)))
  ∧ (if stereoOpt = None then True else mODLinkStereotype (the stereoOpt) sm)
  )"

defs mODLink-def : "mODLink == vmODLink"

end
```

Objekte im Objektdiagramm entsprechen Objekten im System (Definition 7.57). Falls ein Name angegeben ist (Zeile 10), kann der Objektidentifikator mit Hilfe von `mObjName` identifiziert werden. Der Objektidentifikator muss im aktuellen Systemzustand vorhanden sein und, falls im Objektdiagramm angegeben, aus der Trägermenge der entsprechenden Klasse stammen. Die semantische Abbildung für Modifier ist ein Variationspunkt, es wird keine Realisierung als Variante definiert. Objekte können Attribute haben, die in Definition 7.58 im Kontext der Objektreferenz `oid` behandelt werden.

Definition SemAbb 7.57 (OD Semantik, Objekte)

mODObject

Isabelle-Theorie

```
1   consts mODObjectModifier ::
2     "Modifier ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ bool"
3
4   fun mODObject :: "ODObject ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ bool"
5   where
6     "mODObject (ODObject complOpt modifier
7                     refOpt nameOpt
8                    atts) s vars sm = (
9     ∃ oid ∈ UOID sm .(
10      (if nameOpt = None then True else oid = mObjName (the nameOpt)) ∧
11      oid ∈ oids s  ∧
```

```
     (if refOpt = None then True
      else VOid oid ∈ CAR sm (mReferenceType (the refOpt)) ∧
           toClass (mReferenceType (the refOpt)) ∈ UCLASS sm) ∧
     mODAttributes atts s vars oid sm ∧
     mODObjectModifier modifier s vars sm)
   )"
```

Die Angabe eines Attributs im Objektdiagramm bedeutet gemäß Definition 7.58, dass ein entsprechendes Attribut in der Klasse von `oid` vorhanden ist. Ist ein Typ im Objektdiagramm angegeben, entspricht der Typ des Attributs im System der semantischen Abbildung dieses Typs (Zeile 11). Falls ein Wert für das Attribut im Objektdiagramm definiert wurde, wird dieser Wert mit Hilfe der einbetteten Semantik für Werte übersetzt (Zeile 14). Der untersuchte Datenzustand muss den erhaltenen Wert gespeichert haben. Modifier für Attribute können als Variante der nicht weiter definierten Funktion `mODAttributeModifier` abgebildet werden.

Definition SemAbb 7.58 (OD Semantik, Attribute)

mODAttribute

Isabelle-Theorie

```
1   consts mODAttributeModifier ::
2     "Modifier ⇒ iSTATE ⇒ iVarAssign ⇒ SystemModel ⇒ bool"
3
4   fun mODAttribute ::
5     "ODAttribute ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ SystemModel ⇒ bool"
6   where
7     "mODAttribute (ODAttribute modif typeOpt name valueOpt) st vars oid sm = (
8       mODAttributeModifier modif st vars sm ∧
9       mName name ∈ attr sm (Class.classOf sm oid) ∧
10      (if typeOpt = None then True else
11          vtype sm (mName name) = mType (the typeOpt)) ∧
12      (if valueOpt = None then True else
13        (∃ th . the (val sm (dsOf st, oid, mName name)) =
14              ExternalODSemantics.mValue (the valueOpt) st vars oid th sm))
15     )"
```

Die eingeführten Varianten der semantischen Abbildung von Objektdiagrammen werden in Definition 7.59 zusammengetragen. Die gesamte Variabilität der Objektdiagramme ist in Definition 7.60 gezeigt.

Definition Variabilität 7.59 (Semantik OD)

ODSemantics

Feature-Diagramm

```
package mc.umlp.od.semantics;

featurediagram ODSemantics {

  ODSemantics = vMapObject? & vMapLink? & vStereotypes?;

  vMapObject = MapObject;

  vMapLink = MapLink;

  vStereotypes = MapStereotypes;

}
```

Definition Variabilität 7.60 (OD)

OD

Feature-Diagramm

```
package mc.umlp.od;

<<lang,cu>> featurediagram OD {

 OD    = *mc.umlp.od.syntax.ODSyntax &
          <<sm>> *mc.umlp.od.semantics.ODSemantics &
          <<sd>> *systemmodel.SystemModel;

}
```

Arbeiten zur Semantik von Objektdiagrammen sind insbesondere im Kontext von Graphtransformationsansätzen wie [KGKZ09] und [RK08] zu finden. Das liegt daran, dass Transformationsregeln auf (generischen) Instanzgraphen definiert werden, die weitgehend einem Objektdiagramm entsprechen. Die Konformität zu einem Typgraphen, der beispielsweise aus einem Klassendiagramm gewonnen werden kann, ist dann leicht prüfbar. Unsere Semantik hingegen definiert Bedingungen an die Struktur von Systemzuständen im Systemmodell.

7.6 Statecharts

UML/P-Statecharts dienen dazu, Objektverhalten zustandsbasiert zu beschreiben. Dabei können Statecharts eingesetzt werden, das Verhalten der Objekte einer Klasse als Ganzes oder einzelne Methoden einer Klassen zu beschreiben.

Die MontiCore-Grammatik für UML/P-Statecharts ist in Definition C.30 in Anhang C.5 ab Seite 317 angegeben. Die vollständige, resultierende abstrakte Syntax findet sich in Definition C.31. Für die semantische Abbildung gehen wir allerdings von vereinfachten Statecharts aus. Dazu nehmen wir an, dass Statecharts vor der semantischen Abbildung gemäß den in [CGR08a] beschriebenen Transformationsregeln vereinfacht wurden. Das Ergebnis der Transformation ist als Kontextbedingung in der Präsentationsvariante C.32 angegeben und kann wie folgt zusammengefasst werden:

- Die Hierarchie wurde entfernt. Zustände haben keine Subzustände, Transitionen oder interne Transitionen.
- Aktionen (bei Eintritt, Verlassen oder Verweilen) im Zustand wurden entfernt.
- Nur die Modifier `initial` zur Markierung von Startzuständen und `final` zur Markierung von Endzuständen werden betrachtet.
- Die in [Rum04b] beschriebenen Stereotypen wurden entfernt.

- Das Statechart wurde wie in [Rum04b] und [CGR08a] beschrieben vervollständigt, so dass für jede im Statechart beschriebene Eingabe und jeden Zustand immer mindestens eine Transition existiert, die schaltbereit ist (ggf. führt diese aber in einen expliziten Fehlerzustand).

Wir betrachten außerdem keine Methoden-Statecharts und beliebige Codeblöcke. Dies ist in der Kontextbedingung in Variante C.33 formuliert und stellt eine echte Spracheinschränkungen dar. Alle anderen aufgezählten Einschränkungen sind Abkürzungen und reduzieren die Ausdrucksstärke von Statecharts nicht. Basierend auf vereinfachten Statecharts wird die semantische Abbildung definiert. Zuvor werden in Definition 7.61 einige Hilfsfunktionen zur Selektion bestimmter Elemente der abstrakten Syntax eingeführt. Zum Beispiel liefert `preOf` die Vorbedingung einer Transition. Die Funktionsnamen sind selbsterklärend. Die Umsetzung ist unkompliziert und daher die genaue Definition ausgelassen.

Definition SemAbb 7.61 (SC Semantik, Hilfsfunktionen)

SCHelper

Isabelle-Theorie

```
fun preOf :: "SCTransition ⇒ SCAS.Invariant option"
fun postOf :: "SCTransition ⇒ SCAS.Invariant option"
fun srcOf :: "SCTransition ⇒ LiteralsAS.Name"
fun trgOf :: "SCTransition ⇒ LiteralsAS.Name"
fun stmtOf :: "SCTransition ⇒ Statements option"
fun exprOf :: "Invariant ⇒ InvariantExpression"
fun sName :: "SCState ⇒ LiteralsAS.Name"
fun isInitialMod :: "SCModifier ⇒ bool"
fun isFinalMod :: "SCModifier ⇒ bool"
fun methodName :: "SCTransition ⇒ Base.Name"
fun transitionsOf :: "SCElement list ⇒ SCTransition list"
fun initialOf :: "SCElement list ⇒ LiteralsAS.Name option"
fun statesOf :: "SCElement list ⇒ SCState list"
fun argListOf :: "SCTransition ⇒ Expression list"
```

Die Grundidee der semantischen Abbildung, die auch in [CGR08a] beschrieben ist, ist, dass ein Zustand im Statechart einem oder mehreren Zuständen im System entspricht. Die Zuordnung von Systemzuständen zu einem Statechart-Zustand ist ein semantischer Variationspunkt. Konkrete Realisierungen präzisieren die Funktion `proj` in Definition 7.62. Es ist keine Variante explizit angegeben. Eine typische Umsetzungen ist, dass der Statechart-Zustand zum Beispiel als aktueller Wert eines Klassenattributs kodiert ist. `proj x oid s sm` überprüft dann für ein bestimmtes Objekt, ob der Datenzustand des Objekts in `s` dem angegebenen Wert für `x` entspricht. Ebenso könnte das Entwurfsmuster *State* [GHJV95] verwendet werden. Hier ist eine Gruppe von Objekten relevant. `proj` hat dann zu prüfen, ob ein Link zu dem State-Objekt für `x` vorhanden ist.

Ein Zustandsübergang aus einem passenden Quellzustand findet statt, wenn die Transition schaltbereit ist, das heißt eine entsprechende Nachricht empfangen wurde und die Vorbedingung erfüllt ist. Beim Zustandsübergang können Aktionen ausgeführt werden. Anschließend muss die Nachbedingung erfüllt sein und der Zielzustand eingenommen worden sein.

Der erste Teil der semantischen Abbildung für Statecharts in Definition 7.62 führt zu einer Aussage über eine Statechart-Definition insgesamt. Es existiert eine Klasse `C`, für die das

Statechart die Verhaltensdefinition darstellt. Der optional angegebene Typ kann diese Klasse im System festlegen. Für alle Objekte der Klasse und für alle Systemabläufe müssen nun die Bedingungen für Statechart-Zustände, -Transitionen und -Invarianten erfüllt werden.

Objekte von Unterklassen von `C` können gemäß unserer Semantik ein vollständig anderes zustandsbasiertes Verhalten besitzen. Dieser Ansatz enthält die wenigsten Einschränkungen. Ein Entwickler kann dasselbe Statechart für Unterklassen definieren, hat aber auch beliebige Freiheit, Verhalten zu ändern.

Definition SemAbb 7.62 (SC Semantik, Statecharts)

mSCDefinition

Isabelle-Theorie

```
consts proj ::
  "LiteralsAS.Name ⇒ iOID ⇒ iSTATE ⇒ SystemModel ⇒ bool"

fun mSCDefinition :: "SCDefinition ⇒ SystemModel ⇒ bool"
where
 "mSCDefinition (SCDefinition comlOpt None name refTOpt
                                        None scelems invL) sm = (
    ∃ C ∈ UCLASS sm .
     (if refTOpt = None then True
      else TO C = mReferenceType (the refTOpt)) ∧
     (∀ oid ∈ Class.oids sm C .
        ∀ t ∈ TRACES sm .
         mSCStates (statesOf scelems) t oid sm  ∧
         mTrans (transitionsOf scelems) (the (initialOf scelems)) t oid sm   ∧
         mInvariants invL  t (map sName (statesOf scelems)) oid sm
     )
 )"
```

Statechart-Invarianten müssen gelten, wenn sich die Ausführung in einem Statechart-Zustand befindet. Dies wird in Definition 7.63 kodiert. Die semantische Abbildung für die Invarianten selbst ist über die Konfiguration der Spracheinbettung bestimmt.

Definition SemAbb 7.63 (SC Semantik, Statechart-Invarianten)

mSCInvariant

Isabelle-Theorie

```
fun mInvariant ::
  "InvariantExpression ⇒ LiteralsAS.Name list
                        ⇒ iSTATE list ⇒ iOID ⇒ SystemModel ⇒ bool"
where
 "mInvariant ivar stateNames t oid sm = (
   ∀ s ∈ set t .
    (∀ n ∈ set stateNames . proj n oid s sm ⟶
      ExternalSCSemantics.mInvariantExpression ivar s oid sm))"
```

Die semantische Abbildung für Statechart-Zustände in Definition 7.64 besagt, dass alle Systemzustände, die zu einem initialen Zustand gehören, auch initiale Systemzustände sein müssen (Zeile 6). Wird das Objekt `oid` entfernt und entspricht der letzte Zustand dem Statechart-Zustand, so muss dieser Zustand ein finaler Zustand sein. Zusätzlich muss in allen Systemzuständen, die dem Statechart-Zustand entsprechen, eine angegebene Invariante gelten.

Definition SemAbb 7.64 (SC Semantik, Zustände)

mSCState

Isabelle-Theorie

```
fun mSCState ::
  "SCState ⇒ iSTATE list ⇒ iOID ⇒ SystemModel ⇒ bool"
where
 "mSCState (SCState complOpt modifier n invOpt None None None [] [])
                              t oid sm  = (
    (isInitialMod modifier ⟶
       (∀ i . proj n oid (t!i) sm ⟶ (t!i) ∈ (Init (SYSSTS sm)))) ∧
    (∀ i . oid ∈ oids (t!i) ∧ oid ∉ oids (t!(i+1))
        ∧ proj n oid (t!i) sm ⟶  isFinalMod modifier) ∧
    (if invOpt = None then True else
          (∀ i . proj n oid (t!i) sm ⟶
              (ExternalSCSemantics.mInvariantExpression
                  (exprOf (the invOpt)) (t!i) oid sm))
    )
  )"
```

Um die semantische Abbildung von Statechart-Transitionen zu definieren, benötigen wir einige Voraussetzungen. In Definition 7.65 wird in `enabled` (Zeile 31) eine Liste mit schaltbereiten Statechart-Transitionen für den aktuellen Zustand `s` des Systems `sm` und das Objekt `oid` erstellt. Hierzu wird in `msgFor` (Zeile 18) geprüft, ob ein Aufruf im Nachrichtenpuffer von `oid` ist, der dem Namen der Transition entspricht. Aus der Parameterliste in der Nachricht und der erwarteten Parameter der Statechart-Transition wird eine Variablenbelegung konstruiert, mit der auch die Vorbedingung durch die Funktion `mPrePost` überprüft wird.

Definition SemAbb 7.65 (SC Semantik, Transitionen, Bereitschaft)

Enabled

Isabelle-Theorie

```
1  fun mPrePost ::
2   "SCAS.Invariant option ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD
3      ⇒ SystemModel ⇒ bool"
4  where
5    "mPrePost None s va oid th sm = True"
6  | "mPrePost (Some (Invariant idOpt inva)) s va oid th sm =
7         ExternalSCSemantics.mCondition inva s va oid th sm"
8
9  fun mExprList :: "Expressions list ⇒ Base.Name list"
10 where
11   "mExprList [] = []"
12 | "mExprList (x#xs) = (ExternalSCSemantics.mExpressions x)#(mExprList xs)"
13
14 fun mArguments :: "SCTransition ⇒ Base.Name list"
15 where
16   "mArguments t = mExprList (argListOf t)"
17
18 fun msgFor :: "SCTransition ⇒ iMessage buffer
19                ⇒ iOID ⇒ iSTATE ⇒ SystemModel ⇒
20                    (iOID × iTHREAD × iVarAssign) option"
21 where
22   "msgFor t  [] oid s sm = None"
23 | "msgFor t  ((oid1, n, valList , oid2, (Some th))#ms) oid s sm = (
24      let vars = (Variable.constrVarAssign valList (mArguments t)) in
25       if (oid = oid1 ∧ methodName t = n ∧
26           mPrePost (preOf t) s vars oid th sm)
```

```
       then (Some (oid2, th, vars))
       else msgFor t  ms oid s sm
     )"

fun enabled :: "SCTransition list ⇒ LiteralsAS.Name ⇒ iSTATE
             ⇒ iOID ⇒ SystemModel ⇒
                (SCTransition × (iOID × iTHREAD × iVarAssign)) list"
where
  "enabled [] cur s oid sm = []"
| "enabled (t#ts) cur s oid sm = (
   let x = msgFor t (the ((msOf s) oid)) oid s sm  in (
     if (cur = (srcOf t) ∧ x ≠ None)
     then (t,the x)#(enabled ts cur s oid sm)
     else (enabled ts cur s oid sm)))"

```

Der Effekt der Durchführung einer Transition schließt die Ausführung von Aktionen ein, mit der die Transition annotiert sein kann. `takeT` stellt in Definition 7.66 also fest, ob sich die Ausführung von Anweisungen im Zielzustand `s′` widerspiegelt für den Startzustand `s`, die Objektreferenz `oid`, den Thread `th` und die Variablenbelegung `vars`.

Definition SemAbb 7.66 (SC Semantik, Transitionen, Ausführung)

TakeT

Isabelle-Theorie

```
fun mSCStatements ::
  "Statements option ⇒ iSTATE ⇒ iSTATE ⇒ iVarAssign ⇒ iOID
                         ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
  "mSCStatements (Some stmts) s s′ va oid th sm =
       ExternalSCSemantics.mStatements stmts s s′ va oid  th sm"
| "mSCStatements None s s′ va oid th sm = True"

fun takeT :: "SCTransition ⇒ iSTATE ⇒ iSTATE ⇒ iVarAssign ⇒ iOID
                              ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
 "takeT t s s′ vars oid th sm =
   mSCStatements (stmtOf t) s s′ vars oid th sm"
```

Unsere Statechart-Semantik fordert, dass die Verarbeitung einer Transition abgeschlossen ist, bevor eine neue Transition verarbeitet werden kann. Dieses Verhalten wird auch als *Run-To-Completion* bezeichnet [OMG09b]. In Definition 7.67 wird dies folgendermaßen sicher gestellt. Solange die Ausführung der Transition noch nicht begonnen hat, werden die Zustände im Systemablauf übersprungen (Zeile 9). Sobald sie in einem Zustand beginnt, wird dieser Zustand (als dritter Parameter der Funktion) zwischengespeichert (Zeile 8). Ebenso werden bis zum Ende der Transitionsausführung alle Zustände ignoriert (Zeile 13). Der Rest des Systemablaufs und der gespeicherte Zustand mit dem Beginn der Ausführung werden zurückgeliefert (Zeile 12).

Definition SemAbb 7.67 (SC Semantik, Transitionen, Run-To-Completion)

RTC

Isabelle-Theorie

```
1  fun rtc :: "SCTransition  ⇒ iSTATE list ⇒
2    iSTATE option ⇒ iOID ⇒ iOID ⇒ iTHREAD  ⇒ SystemModel
```

```
    ⇒ (iSTATE option × iSTATE list)"
  where
    "rtc tra [] s oid oid' th sm = (s,[])"
  | "rtc tra (t#ts) None oid oid' th sm = (
      if (running (methodName tra) oid oid' th t)
      then rtc tra ts (Some t) oid oid' th sm
      else rtc tra ts None     oid oid' th sm)"
  | "rtc tra (t#ts) (Some s) oid oid' th sm = (
       if (notExecuting (methodName tra) t oid oid' th)
       then (Some s,(t#ts))
       else (rtc tra ts (Some s) oid oid' th sm))"
```

Die semantische Abbildung von Transitionen insgesamt wird in Definition 7.68 in der Funktion `mTrans` (ab Zeile 8) behandelt. Hierfür werden die Systemzustände in einem Ablauf untersucht. Bei einem nicht-leeren Systemablauf treten zwei Fälle auf. Im ersten Fall (Zeile 16) ist keine Transition im aktuellen Zustand schaltbereit. Dann wird der nächste Zustand im Systemablauf untersucht. Im zweiten Fall existiert eine schaltbereite Transition. Gibt es mehrere solcher Transitionen wird mit `chooseTrans` (Zeile 1) eine Transition ausgewählt.

Für die Durchführung einer Transition stellt `oid2` den Aufrufer der Methode dar, `th` ist der ausführende Thread. Die Variablenbelegung wurde in Funktion `enabled` erstellt und genutzt, um die Vorbedingung zu prüfen. `rtc` liefert den ersten Zustand `s'` der Transitionsausführung, im ersten Zustand des Rests `rs` ist die Ausführung abgeschlossen. Das bedeutet, dass die Ausführung der Aktionen an der Transition in `s'` beginnen und im ersten Zustand von `rs` enden (Aufruf `takeT`). Die Nachbedingung und das Erreichen des Zielzustands werden ebenso geprüft. Anschließend wird mit dem Rest des Systemablaufs rekursiv weiter verfahren (Zeile 23).

Definition SemAbb 7.68 (SC Semantik, Transitionen, gesamt)

mTrans

Isabelle-Theorie

```
1   fun chooseTrans ::
2       "(SCTransition × iOID × iTHREAD × iVarAssign) list ⇒
3           (SCTransition × iOID × iTHREAD × iVarAssign)"
4   where
5    "chooseTrans (x#xs) = x"
6
7   function
8    mTrans :: "SCTransition list ⇒ LiteralsAS.Name ⇒
9               iSTATE list ⇒ iOID ⇒ SystemModel ⇒ bool"
10  where
11    "mTrans transs current [] oid sm = True"
12  | "mTrans transs current (s#ts) oid sm = (
13      proj current oid s sm ∧
14      (let enabledTrans = enabled transs current s oid sm in (
15        if enabledTrans = []
16        then mTrans transs current ts oid sm
17        else
18          (let (tra,oid2,th,vars) = chooseTrans enabledTrans in (
19           (let (s',rs) = (rtc tra ts None oid oid2 th sm) in
20             (takeT tra (the s') (rs!0) vars oid th sm ∧
21              mPrePost (postOf tra) (rs!0) vars oid th sm ∧
22              proj (trgOf tra) oid (rs!0) sm ∧
23              mTrans transs (trgOf tra) (tl rs) oid sm)
24           ))
```

```
    ))))"
```

Semantische Variabilität wurde für UML/P-Statecharts nicht ausführlich definiert. Es wurden keine Varianten explizit beschrieben. Die Anwendung der semantischen Abbildung soll zudem nur für entsprechend vereinfachte Statecharts erfolgen (Definition 7.69), da sonst viele Konstrukte semantisch nicht berücksichtigt werden. Die gesamte Variabilität der Statecharts ist in Definition 7.70 angegeben. Gerade semantisch könnten einige Varianten definiert werden (alternative Definitionen für Schaltbereitschaft, Auswahl bei mehreren schaltbereiten Transitionen, etc.). Gute Übersichten über die Vielfalt der Bedeutung von Statecharts finden sich in [Bee94], [CD07] und [CD05]. Die in [TA06] ausführlich untersuchte semantische Variabilität von UML-StateMachines bezüglich der Schaltbereitschaft oder der Auswahl von Transitionen kann für zukünftige Erweiterungen unserer Statechart-Semantik als Vorlage dienen. Unsere Semantik bietet darüber hinaus die Möglichkeit, verschiedene Implementierungsvarianten von Statecharts durch unterschiedliche Ausprägungen der Funktion `proj` (Variationspunkt `ProjectStates`) zu untersuchen.

Definition Variabilität 7.69 (Semantik SC)

SCSemantics

Feature-Diagramm

```
package mc.umlp.sc.semantics;

featurediagram SCSemantics {

  SCSemantics = ProjectStates?;

  constraints {
    SCSemantics -> SCSyntax.SCConstr.NoCodeNoMethodSC &
                   SCPresentation.SCAbb.SimplifiedSC;
  }
}
```

Definition Variabilität 7.70 (SC)

SC

Feature-Diagramm

```
package mc.umlp.sc;

<<lang,cu>>featurediagram SC {

 SC    = *mc.umlp.sc.presentation.SCPresentation &
         *mc.umlp.sc.syntax.SCSyntax &
         <<sm>> *mc.umlp.sc.semantics.SCSemantics &
         <<sd>> *systemmodel.SystemModel;
}
```

7.7 Sequenzdiagramme

Während Statecharts das zustandsbasierte Verhalten von Objekten vollständig beschreiben, werden Sequenzdiagramme verwendet, um das Verhalten als exemplarische Interaktion zwischen

Objekten zu charakterisieren [BGH$^+$98]. Die MontiCore-Grammatik für die Sequenzdiagramme der UML/P ist in Definition C.36 und die entsprechende abstrakte Syntax in Definition C.37 in Anhang C.6 ab Seite 325 gelistet. Wie bei Statecharts betrachteten wir für die semantische Abbildung nur vereinfachte Sequenzdiagramme. Diese Vereinfachung können als syntaktische Transformation definiert werden und sind damit Abkürzungen. Ihr Ergebnis ist als Sammlung von Kontextbedingungen in Variante C.38 beschrieben. Die Vereinfachungen sind zusammengefasst:

- Objekterzeugung wird entfernt, da sie nach [Rum04b] bzw. [Rum96] durch eine vorgegebene Objektmenge simuliert und damit ignoriert werden kann.
- Aktivitätsbalken dienen dazu, Mehrdeutigkeiten von Aufrufen und Rückgaben bei verschachtelten Aufrufen aufzulösen [Rum04b]. Sie sind nicht notwendig, wenn allen Aufrufen auch eine Rückgabe zugeordnet ist. Daher werden Aktivitätsbalken und die damit zusammenhängenden syntaktischen Konstrukte hier entfernt.

Die Semantik ist unvollständig in dem Sinn, dass Modifier für Sequenzdiagramme nicht berücksichtigt werden. Bevor wir die semantische Abbildung für Sequenzdiagramme angeben, enthält Definition 7.71 eine Reihe von Hilfsfunktionen mit Hilfe derer einfacher auf bestimmte Komponenten des Sequenzdiagramms zugegriffen werden kann. Ihre exakte Definition ist unkompliziert und daher nicht angegeben. `callName`, `trg` und `src` liefern Methodennamen bzw. Quelle oder Ziel einer Interaktion. `sdInteractionNames` erzeugt eine Liste von Namen, die den Methodenaufrufen der Sequenzdiagramm-Interaktionen entsprechen, wobei jeder Name als Aufruf und als Rückgabe im System vorkommen kann, so dass die Liste jeden Namen doppelt enthält. `sdInteractions` liefert aus den Sequenzdiagrammelementen nur die Aufrufe, Returns und Bedingungen, die Funktion `sdObject` nur die Objekte. `hasValue` prüft, ob eine Liste von Stereotypwerten das angegebene Name-Wert-Paar enthält. Die drei `isMatch`-Funktionen überprüfen die zwei übergebene Stereotypen, ob sie ein Name-Wert-Paar, z.B. (`match="visible"`) enthalten.

Definition SemAbb 7.71 (SD Semantik, Hilfsfunktionen)

SDHelper

Isabelle-Theorie

```
fun callName :: "SDCallInteraction ⇒ LiteralsAS.Name"
fun trg :: "SDElement ⇒ LiteralsAS.Name"
fun src :: "SDElement ⇒ LiteralsAS.Name"
fun sdInteractionNames ::  "SDElement list
      ⇒ LiteralsAS.Name ⇒ LiteralsAS.Name stack ⇒ LiteralsAS.Name list"
fun sdInteractions :: "SDElement list ⇒ SDElement list"
fun sdObjects :: "SDElement list ⇒ SDObject list"
fun hasValue :: "LiteralsAS.Name ⇒ STRING ⇒ StereoValue list ⇒ bool"
fun isMatchComplete :: "Stereotype option ⇒ Stereotype option ⇒ bool"
fun isMatchVisible :: "Stereotype option ⇒ Stereotype option ⇒ bool"
fun isMatchInitial :: "Stereotype option ⇒ Stereotype option ⇒ bool"
```

Die semantische Abbildung für Sequenzdiagramme orientiert sich an der losen Semantik wie sie in [Rum04b, CG09] angegeben ist. Die Grundidee ist, dass Objekte im Sequenzdiagramm auf Objekte im System abgebildet werden. Hierzu dient in Definition 7.72 die Funktion `mObjName`.

Wir können in der Funktion `mSDDefinition` die semantische Abbildung für alle Sequenzdiagrammobjekte im Kontext nur eines Zustands bilden, da Objekterzeugung im Sequenzdiagramm keine Rolle spielt. Interaktionen im Sequenzdiagramm werden auf Interaktionen im System abgebildet. Dafür benötigen wir in der Funktion `mSDDefintion` einen Systemablauf `t` und prüfen dann mit Hilfe von `compatStrict`, ob dieser Systemablauf die Interaktionen des Sequenzdiagramms genau enthält. Die Kompatibilität wird für jedes Objekt `oid` einzeln geprüft. Der Systemablauf wird vorher bereinigt, so dass er nur noch bestimmte Interaktionen mit Beteiligung von `oid` enthält. Diese Bereinigung in der Funktion `filter` kann mit Hilfe von Stereotypen gesteuert werden. Es werden alle in [Rum04b] vorgeschlagenen Stereotypen berücksichtigt und im weiteren Verlauf dieses Abschnitts erklärt. Die Stereotypen können entweder global für das Sequenzdiagramm angegeben werden oder für jedes Objekt einzeln definiert werden. Daher müssen immer zwei Stereotypen ausgewertet werden. Die Funktion `objList` erstellt eine Liste von Objekten im System aus den Objekten im Sequenzdiagramm. `objStereoList` enthält Paare von Objektnamen und optionalen Stereotypen des Objekts. Alle Interaktionen werden im Kontext eines Thread `th` betrachtet, da im UML/P-Sequenzdiagramm typischerweise keine nebenläufigen Interaktionen stattfinden. Das Sequenzdiagramm muss für jeden Thread gelten, der die Interaktion im Sequenzdiagramm durchführt. Die semantische Abbildung ließe sich jedoch leicht, ähnlich wie in [DHC07] beschrieben, erweitern, indem für Interaktionen Stereotypen mit so genannten *thread tags* eingeführt werden, die festlegen, welcher Thread welche Interaktion im Sequenzdiagramm durchführt.

Definition SemAbb 7.72 (SD Semantik, Sequenzdiagramm)

mSDDefinition

Isabelle-Theorie

```
consts mObjName :: "LiteralsAS.Name ⇒ iOID"

consts filter :: "Stereotype option ⇒ Stereotype option ⇒ iSTATE list
    ⇒ iOID list ⇒ LiteralsAS.Name list ⇒ iOID ⇒ iTHREAD ⇒ iSTATE list"

fun objList :: "SDObject list ⇒ iOID list "
where
  "objList []  = []"
| "objList ((SDObject complOpt modifier
             name typeOpt)#xs)  =  (mObjName name#(objList xs))"


fun mSDDefinition :: "SDDefinition ⇒ iSTATE list ⇒ SystemModel ⇒ bool"
where
 "mSDDefinition (SDDefinition complOpt stereoOpt name ems invars) t sm =
   (mSDObjects (sdObjects ems) (t!0) sm ∧ t ∈ TRACES sm  ∧
   (∀ th ∈ UTHREAD sm.
     (∀ (obn,oidStereo) ∈ set (objStereoList (sdObjects ems)) .
       let t' =
         filter oidStereo stereoOpt t (objList (sdObjects ems))
           (sdInteractionNames (sdInteractions ems) obn []) (mObjName obn) th  in
       (if t' = [] then True else
        compatStrict (sdInteractions ems) t' (t'!0) [] (mObjName obn)  th  sm))
   )
 )"
```

Die Abbildung von prototypischen Objekten des Sequenzdiagramms auf Objekte im System ist ein Variationspunkt. Die einfachste Realisierung, eine eins zu eins Umsetzung, ist in Variante 7.73 angegeben. Wiederum nehmen wir vereinfachend eine eindeutige Zuordnung von Objekten im Sequenzdiagramm zu Objekten im System an. Eine Erweiterung dieser Definition ist aber leicht möglich.

Variante SemAbb 7.73 (Abbildung von Objektnamen auf Objekte)

mObjName

Isabelle-Theorie

```
theory MapObject
imports "$UMLP/def/mc/umlp/sd/semantics/SDSemantics-base"
begin

fun vmObjName :: "LiteralsAS.Name ⇒ iOID"
where
 "vmObjName n = Oid (mName n)"

defs mObjName-def : "mObjName == vmObjName"

end
```

Die folgenden Varianten beschreiben eine Realisierung der Funktion `filter`, die den Systemablauf entsprechend der in [Rum04b] angegebenen Stereotypen bereinigt. Berücksichtigt werden die folgenden Werte für den Stereotyp `match`:

complete Die Beobachtung für das Objekt ist vollständig, alle im System vorkommenden Interaktionen sind im Diagramm zu finden.

visible Die Beobachtung für das Objekt ist vollständig in Bezug auf andere im Diagramm angegebene Objekte. Interaktionen mit nicht im Diagramm modellierten Objekten dürfen existieren.

initial Andere Interaktionen zwischen im Diagramm angegebenen Objekten sind erlaubt. Das jeweils erste Auftreten einer Interaktion wird registriert. Nach „Abhaken" einer aufgetretenen Interaktion wird die nächste Interaktion des Diagramms erwartet.

free Beliebige Interaktionen zwischen Objekten sind erlaubt, so lange insgesamt die Interaktionen des Sequenzdiagramms beobachtet werden.

In Funktion `vfilter` in Variante 7.74 wird nur die entsprechende Filterfunktion für ein Objekt auf Basis der Stereotypen für das einzelne Objekt und für das gesamte Diagramm ausgewählt. Dabei wird der zweite Stereotyp für das gesamte Diagramm priorisiert.

Variante SemAbb 7.74 (Trace gemäß Stereotypen bereinigen)

Filter

Isabelle-Theorie

```
fun vfilter ::
  "Stereotype option ⇒ Stereotype option ⇒ iSTATE list
   ⇒ iOID list ⇒ LiteralsAS.Name list ⇒ iOID ⇒ iTHREAD ⇒ iSTATE list"
where
 "vfilter oidStereoOpt sdStereoOpt t oidl interNames oid th = (
    if isMatchComplete oidStereoOpt sdStereoOpt
    then filterComplete t oid th
    else (if isMatchVisible oidStereoOpt sdStereoOpt
          then filterVisible t oidl oid th
          else (if isMatchInitial oidStereoOpt sdStereoOpt
                then filterInitial t oidl interNames interNames oid th
                else filterFree t oidl interNames oid th)))"

defs filter-def : "filter == vfilter"
```

`filterComplete` in Variante 7.75 filtert den Trace für ein Objekt und einen Thread. Es werden alle Nachrichten beibehalten für die das Objekt entweder Sender oder Empfänger ist und der Thread in der Nachricht mit dem übergebenen Thread übereinstimmt. Wir erhalten also einen Systemablauf, mit allen Interaktionen, an denen das Objekt zusammen mit dem Thread beteiligt ist.

Variante SemAbb 7.75 (Trace gemäß Stereotyp Complete bereinigen)

FilterComplete

Isabelle-Theorie

```
1   fun filterCondComplete ::
2     "iSTATE ⇒ iSTATE ⇒ iOID ⇒ iTHREAD ⇒ bool"
3   where
4    "filterCondComplete s s’ oid th = (
5     (∃ m . (m ∈ set (the (msOf s oid)) ∧  m ∉ set (the (msOf s’ oid))
6          ∧ the (msgthread m) = th)) ∨
7     (∃ oid’ ∈ oids s . ∃ m .
8            (the (msgthread m) = th) ∧ m ∉ set (the (msOf s’ oid’)) ∧
9             m ∈ set (the (msOf s oid’)) ∧  Message.sender m = oid))"
10
11  fun filterComplete :: "iSTATE list ⇒ iOID ⇒ iTHREAD ⇒ iSTATE list"
12  where
13    "filterComplete (s#[]) oid th = (s#[])"
14  | "filterComplete (s#s’#xs) oid th = (
15      if   filterCondComplete s s’ oid th
16      then (s#filterComplete (s’#xs) oid th)
17      else filterComplete (s’#xs) oid th)"
```

Nach der Bedingung `filterVisibleCond` in Variante 7.76 erhalten wir nach Anwendung von `filterVisible` einen Systemablauf, der alle Interaktionen (Senden oder Empfangen einer Nachricht) des Objekts `oid` für den Thread `th` enthält, wobei der Interaktionspartner aus der Menge der im Sequenzdiagramm gegebenen Objekte `oidl` stammt (Zeile 7, vgl. hierzu auch Zeile 7 aus Variante 7.75, bei der alle Objekte im System berücksichtigt werden). Alle „nicht sichtbaren" Interaktionen mit anderen Objekten werden ignoriert.

Variante SemAbb 7.76 (Trace gemäß Stereotyp Visible bereinigen)

FilterVisible

Isabelle-Theorie

```
1   fun filterVisibleCond ::
2     "iSTATE ⇒ iSTATE ⇒ iOID list ⇒ iOID ⇒ iTHREAD ⇒ bool"
3   where
4    "filterVisibleCond s s’ oidl oid th = (
5      (∃ m . m ∈ set (the (msOf s oid)) ∧  m ∉ set (the (msOf s’ oid)) ∧
           (the (msgthread m) = th)) ∨
     (∃ oid' ∈ set oidl  . ∃ m . (the (msgthread m) = th) ∧
            m ∉ set (the (msOf s' oid')) ∧ m ∈ set (the (msOf s oid')) ∧
            Message.sender m = oid))"

fun filterVisible :: "iSTATE list ⇒ iOID list ⇒ iOID ⇒ iTHREAD ⇒ iSTATE list"
where
 "filterVisible (s#[]) oidl oid th = (s#[])"
|"filterVisible (s#s'#xs) oidl oid th =  (
    if filterVisibleCond s s' oidl oid th
    then (s#(filterVisible (s'#xs) oidl oid th))
    else filterVisible (s'#xs) oidl oid th
 )"
```

Die Variante 7.77, die ein initiales Matching der Interaktionen liefert, ist etwas umständlicher zu kodieren, da wir zusätzlich die Menge der Interaktionen des Objekts aus dem Sequenzdiagramm (als Listen von Namen `nl` bzw. `nl2`) mitführen müssen. Es wird geprüft, ob zwischen den aktuell betrachteten Zustände `s` und `s'` eine Interaktion stattfindet, die einer Interaktion des Sequenzdiagramms für das Objekt `oid` entspricht. Ist dies der Fall wird diese Interaktion „abgehakt“, das heißt aus der Liste der Interaktionen gelöscht (Zeile 18). Andernfalls (Zeile 19) wird rekursiv die nächste Interaktion aus `nl2` untersucht. Wird keine Interaktion gefunden, wird der betrachtete Zustand übersprungen (Zeile 20). Beim Bereinigen des Trace wird noch nicht die korrekte Reihenfolge der Nachrichten geprüft. Es werden nur Interaktionen aus dem Systemablauf gelöscht, die nicht im Sequenzdiagramm aufgeführt sind oder doppelt auftreten.

Variante SemAbb 7.77 (Trace gemäß Stereotyp Initial bereinigen)

FilterInitial

Isabelle-Theorie

```
1   constdefs filterInitialCond ::
2     "iSTATE ⇒ iSTATE ⇒ iOID list ⇒ LiteralsAS.Name ⇒ iOID ⇒ iTHREAD ⇒ bool"
3   filterInitialCond-def :
4     "filterInitialCond s s' oidl name oid th == (
5       (∃ m .
6         (m ∈ set (the (msOf s oid)) ∧  m ∉ set (the (msOf s' oid)) ∧
7          (the (msgthread m) = th))) ∨
8       (∃ oid' ∈ set oidl  . ∃ m . (the (msgthread m) = th) ∧
9         m ∉ set (the (msOf s' oid')) ∧ m ∈ set (the (msOf s oid')) ∧
10           Message.sender m = oid ∧ msgname m = mName name))"
11
12  fun filterInitial ::
13   "iSTATE list ⇒ iOID list ⇒ LiteralsAS.Name list ⇒ LiteralsAS.Name list
14       ⇒ iOID ⇒ iTHREAD ⇒ iSTATE list"
15  where
16  "filterInitial (s#s'#xs) oidl nl (n#nl2) oid th =  (
17      if filterInitialCond s s' oidl n oid th
18      then (s#(filterInitial (s'#xs) oidl (remove1 n nl) nl2) oid th)
19      else filterInitial (s#s'#xs) oidl  nl nl2 oid th)"
20  |"filterInitial (s#s'#xs) oidl nl [] oid th =
21            filterInitial (s#xs) oidl nl nl oid th"
22  |"filterInitial (s#xs) oidl [] [] oid th = []"
23  |"filterInitial (s#[]) oidl nl nl2 oid th = (s#[])"
```

Die letzte Variante 7.78, die den Standardfall darstellt, liefert einen beliebigen Systemablauf, der die Bedingung `matchFree` erfüllt. Sie sagt aus, dass für jede Interaktion im bereinigten

Trace eine Interaktion im Sequenzdiagramm vorhanden sein muss. Zusätzlich muss jeder Zustand in `t'` in der richtigen Reihenfolge auch in `t` auftauchen. Es ist offen gelassen, welche Interaktion gewählt wird, wenn mehrere Möglichkeiten zur Verfügung stehen.

Variante SemAbb 7.78 (Trace gemäß Stereotyp Free bereinigen)

FilterFree

Isabelle-Theorie

```
fun matchFree ::
  "iSTATE list ⇒ iSTATE list ⇒ iOID list ⇒ LiteralsAS.Name list
     ⇒ iOID ⇒ iTHREAD ⇒ bool"
where
 "matchFree t t' oidl nl oid th = (
   ∀ i . i < length t' ⟶
    (∃ m .
     (m ∈ set (the (msOf (t'!i) oid)) ∧
      m ∉ set (the (msOf (t'!(i+1)) oid)) ∨
     (∃ oid' ∈ set oidl . m ∉ set (the (msOf (t'!(i+1)) oid')) ∧
      m ∈ set (the (msOf (t'!i) oid')) ∧ Message.sender m = oid)
     ∧ msgname m = mName (nl!i)) ∧ (the (msgthread m) = th)) ∧
    (∃ j  . i ≤ j ∧  t'!i = t!j))"

constdefs filterFree ::
  "iSTATE list ⇒ iOID list ⇒ LiteralsAS.Name list ⇒
      iOID ⇒ iTHREAD ⇒ iSTATE list"
 filterFree-def: "filterFree t oidl nl oid th ==
    (if (∃ t' . matchFree t t' oidl nl oid th)
     then (SOME t' . matchFree t t' oidl nl oid th) else [])"
```

Nach der Bereinigung des Systemablaufs für ein Objekt `oid` und einen Thread `th` enthält dieser nur noch Interaktionen mit Beteiligung dieses Objekts und Threads. Je nachdem, welche Variante zur Bereinigung des Trace gewählt wurde, wurden weitere Zustände mit Interaktionen aus dem Trace entfernt. Je „strenger" das Matching definiert ist, desto mehr Nachrichten verbleiben im Trace, `complete` fordert zum Beispiel, dass alle Interaktionen des Objekts im Sequenzdiagramm erscheinen müssen. Im entgegengesetzten Fall können bei `free` viele Interaktionen des Objekts mit nicht dargestellten Objekten und wiederholte Interaktionen ignoriert werden. Es ist nun ausreichend, den Ablauf im Sequenzdiagramm direkt mit dem Ablauf im System zu vergleichen. Diese Funktion wird Definition 7.79 eingeführt.

Neben der Liste der Interaktionen, dem bereinigten Trace `ts` für das Objekt `oid` und den Thread `th` werden zwei weitere Argumente benötigt. Der zusätzliche Zustand `cnds` ist der Zustand, in dem Bedingungen des Sequenzdiagramms überprüft werden. Der Stack von Namen `st` speichert Namen von Methodenaufrufen, zu denen ein passender Return im Trace gefunden werden muss. `compatStrict` arbeitet rekursiv jeweils eine Sequenzdiagramm-Interaktion und einen Systemzustand ab. Eine Ausnahme bildet das Überprüfen von Bedingungen. Hier wird der Trace nicht verkürzt. Bedingungen werden in der Funktion `matchCond` geprüft, Aufrufe mit der Funktion `matchCall` und Rückgaben in der Funktion `matchRet`. Bei Methodenaufrufen wird der aktuelle Methodenname auf den Stapel gelegt, bei einem Return wieder entfernt. So können auch geschachtelte Methodenaufrufe korrekt behandelt werden. Sind Trace und Interaktionen abgearbeitet (Zeile 5), wurde die Kompatibilität erfolgreich festgestellt. Die Fälle, dass die Interaktion nur zum Teil bzw. mehrmals beobachtet wird, sind in `compatStrict` unterspezifiziert. Für eine wiederholte Prüfung der Interaktionen bei einem längeren Trace müssen die Interaktionen noch einmal als zusätzliches Argument zur Funktion hinzugefügt werden, damit wieder von vorn begonnen werden kann.

Definition SemAbb 7.79 (SD Semantik, Kompatibilität zum Trace)

CompatStrict

Isabelle-Theorie

```
fun compatStrict ::
  "SDElement list ⇒ iSTATE list ⇒ iSTATE ⇒ LiteralsAS.Name stack
                     ⇒ iOID ⇒  iTHREAD  ⇒ SystemModel ⇒ bool"
where
 "compatStrict [] [] cnds [] oid th sm = True"
|"compatStrict ((SDElementSDCondition x)#xs) ts cnds st oid th sm =
      (matchCond x cnds oid th sm ∧
       compatStrict xs ts cnds st oid th sm)"
|"compatStrict ((SDElementSDCallInteraction x)#xs) (t#ts) cnds st oid th sm =
   (matchCall x t oid th sm ∧
     compatStrict xs ts t (push (callName x) st) oid th sm)"
|"compatStrict ((SDElementSDReturnInteraction x)#xs) (t#ts) cnds st oid th sm =
   (let (st',m) = pop st in
    matchRet x t m oid th sm ∧
      compatStrict xs ts t st' oid th sm)"
```

In Definition 7.80 wird festgestellt, ob ein Methodenaufruf im Sequenzdiagramm im aktuellen Systemzustand zu finden ist. Falls es sich um einen Aufruf von dem oder an das betrachtete Objekt handelt, wird geprüft, ob eine passende Nachricht im Nachrichtenpuffer des Empfängers vorhanden ist. Die Liste der Nachrichtenparameter muss mit der Auswertung der Liste der Ausdrücke `exprl` übereinstimmen. Ausgewertet werden die Ausdrücke im Kontext des Aufrufers.

Definition SemAbb 7.80 (SD Semantik, Kompatibilität mit Aufruf)

MatchCall

Isabelle-Theorie

```
fun matchCall ::
 "SDCallInteraction ⇒ iSTATE ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
"matchCall (SDCallInteraction src1 None trg1 stereoOpt
            (SDCallMessageSDMethodCall
             (SDMethodCall n (Some (SDArguments exprl incom)))))
         s oid th sm =
 (if (mObjName src1 = oid ∨ mObjName trg1 = oid)
  then (mObjName trg1, mQualifiedName n,
          map (λ e . mExpressions e s (mObjName src1) th sm) exprl,
        mObjName src1, (Some th))
             ∈ set (the (msOf s (mObjName trg1)))
 else True)"
```

Analog werden Returns in Definition 7.81 behandelt. Falls das betrachtete Objekt Sender oder Empfänger des Returns ist, muss eine passende Nachricht beim Empfänger vorliegen.

Definition SemAbb 7.81 (SD Semantik, Kompatibilität mit Rückgabe)

MatchRet

Isabelle-Theorie

```
fun matchRet ::
 "SDReturnInteraction ⇒ iSTATE ⇒ LiteralsAS.Name ⇒
     iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
 "matchRet (SDReturnInteraction trg1 None src1 None
     (SDReturnMessageSDReturnStatement
       (SDReturnStatement exprOpt None)))
         s m oid th sm =
  (if (mObjName src1 = oid  ∨ mObjName trg1 = oid)
   then (mObjName trg1, mName m ,
      (if exprOpt = None then []
       else  [mExpressions (the exprOpt) s (mObjName src1) th sm]),
        oid, (Some th))
             ∈ set (the (msOf s (mObjName trg1)))
  else True)"
```

Für die Überprüfung, ob eine Bedingung im Sequenzdiagramm erfüllt wird, ist zunächst festzustellen, ob die Bedingung das betrachtete Objekt betrifft. Ist dies der Fall, wird die Bedingung ausgewertet und muss wahr ergeben. Die Auswertung von Bedingungen und Ausdrücken erfolgt in der eingebetteten Sprache.

Definition SemAbb 7.82 (SD Semantik, Kompatibilität mit Bedingung)

MatchCond

Isabelle-Theorie

```
fun matchCond ::
    "SDCondition ⇒ iSTATE ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
"matchCond (SDCondition xl (SDAS.Invariant n invar )) s oid th sm =
 (if (∃ x ∈ set xl . mObjName x = oid)
  then (ExternalSDSemantics.mInvariantExpression invar s oid th sm)
  else True)"
```

Die semantische Abbildung für Objekte im Sequenzdiagramm beschränkt sich darauf zu prüfen, ob das entsprechende Objekt im System existiert. Ist eine Klasse für das Objekt angegeben, muss das Objekt ein Objekt der Klasse sein und die Klasse selbst im Universum `UCLASS` existieren. Objekte von Unterklassen können wie in der Statechart-Semantik ein anderes Interaktionsverhalten aufweisen.

Definition SemAbb 7.83 (SD Semantik, Objekte)

mSDObject

Isabelle-Theorie

```
fun mSDObject :: "SDObject ⇒ iSTATE ⇒ SystemModel ⇒ bool"
where
 "mSDObject (SDObject complOpt modifier name typeOpt) s sm =
    (mObjName name ∈ oids s ∧
    (if typeOpt = None then True else
       mObjName name ∈
         Class.oids sm (toClass (mReferenceType (the typeOpt))) ∧
      (toClass (mReferenceType (the typeOpt))) ∈ UCLASS sm))"
```

Die Variabilität der semantischen Abbildung ist im Feature-Diagramm in Definition 7.84 dargestellt, während Definition 7.85 die gesamte Variabilität der UML/P-Sequenzdiagramme beinhaltet.

Definition Variabilität 7.84 (Semantik SD)

SDSemantics

Feature-Diagramm

```
package mc.umlp.sd.semantics;

featurediagram SDSemantics {

  SDSemantics = vMapObject? & vMatch?;

  vMapObject = MapObject?;

  vMatch = MatchSelect?;

  constraints {
    SDSemantics -> SDPresentation.SimplifiedSD;
    SDSemantics.vMatch.MatchSelect
       -> SDSyntax.vStereos.CompleteVisibleInitialFree;
  }
}
```

Definition Variabilität 7.85 (SD)

SD

Feature-Diagramm

```
package mc.umlp.sd;

<<lang,cu>>featurediagram SD {

 SD    = *mc.umlp.sd.presentation.SDPresentation &
         *mc.umlp.sd.syntax.SDSyntax &
         <<sm>> *mc.umlp.sd.semantics.SDSemantics &
         <<sd>> *systemmodel.SystemModel;

}
```

Eine Reihe von Arbeiten beschäftigt sich mit der Semantik von Sequenzdiagrammen bzw. UML-Interaktionen. Der größte Unterschied zu den meisten anderen Arbeiten wie zum Beispiel [HHRS05, HM08, CK04] ist, dass Sequenzdiagramme in unserer Semantik eine totale Ordnung der Interaktionen im System beschreiben, während sonst eine Halbordnungssemantik genutzt wird. In [Cen07] wird solch eine Semantik auch auf Systemmodellbasis definiert. Wir verwenden eine totale Ordnung, um einerseits den in [BGH$^+$98] beschriebenen exemplarischen Charakter von Sequenzdiagrammen zu unterstreichen. Andererseits sollen Sequenzdiagramme einfach verständlich und methodisch nicht zur Spezifikation von vollständigem Verhalten geeignet sein. Viele andere Ansätze versuchen aber genau diese Allgemeingültigkeit von Sequenzdiagrammen herzustellen und halten sich dabei an den UML-Standard, in dem mächtige Operatoren zur Kombination von Interaktionen definiert werden (parallele, alternative Kombination, negative Interaktionen etc.).

7.8 UML/P-Konfiguration

Im letzten Abschnitt dieses Kapitels zeigen wir, wie die UML/P-Diagrammarten und Sprachfragmente für eine konkrete Anwendung konfiguriert werden können. Dafür fassen wir zunächst die Variabilität aller Sprachen der UML/P im Feature-Diagramm in Definition 7.86 zusammen. Für jede Sprache kann eine Konfiguration angegeben werden, ist aber nicht zwingend erforderlich. Generell führt eine fehlende oder unvollständige Konfiguration dazu, dass Variationspunkte offen bleiben. Hierdurch erhöht sich die Unterspezifikation, die kein Problem in unserem Ansatz darstellt, eventuell sogar explizit gewünscht wird.

Definition Variabilität 7.86 (UML/P)

UMLP

Feature-Diagramm

```
package mc;

featurediagram UMLP {

 UMLP = *mc.literals.Literals |
        *mc.types.Types |
        *mc.umlp.common.Common |
        *mc.javap.JavaP |
        *mc.umlp.ocl.OCL |
        *mc.umlp.cd.CD |
        *mc.umlp.od.OD |
        *mc.umlp.sc.SC |
        *mc.umlp.sd.SD;
}
```

Die Konfiguration 7.87 zeigt eine Konfiguration von UML/P, bei der für jede Sprache auf eine entsprechende Subkonfiguration verwiesen wird. Wir stellen im Folgenden nur die Konfiguration der Klassendiagramme ausschnittsweise dar. Die Konfiguration der anderen Sprachen erfolgt analog. Mit Hilfe unseres Werkzeugs ThyConfig werden nach Überprüfung der Konformität zu den entsprechenden Feature-Diagrammen für jede Konfiguration die konfigurierte semantische Abbildung und die konfigurierten Kontextbedingungen generiert (vgl. Kapitel 6, Abschnitt 6.6). Da als semantische Domäne immer das Systemmodell verwendet wird, werden Konfigurationen des Systemmodells überlagert, anschließend die Konformität zum Feature-Diagramm des Systemmodells geprüft und dann das konfigurierte Systemmodell generiert. Alle Sprachen verwenden somit einheitlich das eine konfigurierte Systemmodell.

Konfiguration 7.87 (UML/P-Konfiguration)

UMLPKonf

Konfiguration

```
package conf.mc;

config UMLPConf for mc.UMLP {

 *conf.mc.literals.LiteralsConf,
 *conf.mc.types.TypesConf,
 *conf.mc.umlp.common.CommonConf,
 *conf.mc.javap.JavaPConf,
 *conf.mc.umlp.ocl.OCLConf,
  *conf.mc.umlp.cd.CDConf,
  *conf.mc.umlp.od.ODConf,
  *conf.mc.umlp.sc.SCConf,
  *conf.mc.umlp.sd.SDConf
}
```

Die Beispielkonfiguration 7.88 für Klassendiagramme zeigt, welche syntaktischen Varianten ausgewählt wurden. In den Zeilen 5-8 ist zu sehen, dass Invarianten als OCL-Invarianten spezifiziert werden, Werte und Methodenrümpfe als Java-Ausdrücke bzw. -BlockStatements. Zusätzlich fordert die Konfiguration die Verwendung der Stereotypmenge `SingletonClass`. Hierzu passend wird in Zeile 11 die semantische Abbildung `MapSingletonClass` für diese Stereotypenmenge (enthält nur den Stereotyp `singleton`) gewählt. Daneben werden weitere Varianten der semantischen Abbildung ausgewählt. Ebenfalls angegeben werden die Varianten des Systemmodells. In unserem Beispiel werden in nur einer Konfiguration sämtliche syntaktische und semantische Varianten ausgewählt. Es ist auch möglich, die Konfiguration weiter zu modularisieren und beispielsweise die Systemmodellkonfiguration auszulagern. Diese Konfiguration könnte dann von anderen Sprachen wiederverwendet werden. Mit diesem Mechanismus lassen sich auch leicht Teilkonfigurationen, z.B. als Standardkonfigurationen, erstellen, kombinieren und wiederverwenden.

Konfiguration 7.88 (Klassendiagramm-Konfiguration)

`CDKonf`

`Konfiguration`

```
1   package conf.mc.umlp.cd;
2
3   config CDConf for mc.umlp.cd.CD {
4
5     CDSyntax.LangParam.InvariantExpression.OCLInvariant,
6     CDSyntax.LangParam.Value.JavaExpression,
7     CDSyntax.LangParam.Body.JavaBlockStatement,
8     CDSyntax.Stereos.SingletonClass,
9     CDSyntax.CDConstr.RestrictCD,
10
11    CDSemantics.vStereotypes.MapSingletonClass,
12    CDSemantics.vMapSuperClasses.MapSuperclassesDirect,
13    CDSemantics.vMapMethod.MapMethod,
14    CDSemantics.vMapConstructor.MapConstructor,
15    CDSemantics.vMapModifiers.MapModifiers,
16    CDSemantics.vMapAssociations.MapAssociations,
17
18    SystemModel.vObject.AntiSymSub,
19    SystemModel.vControl.TypeSafeOps,
20    SystemModel.vData.QualifiedAssociation,
21    SystemModel.vData.CompositeAssociation,
22    SystemModel.vData.DerivedAssociation,
23    SystemModel.vData.BinaryAssociation,
24    SystemModel.vData.StaticAttr,
25    SystemModel.vControl.StaticOpn,
26    SystemModel.vState.Query
27  }
```

Nach erfolgreicher Überprüfung werden, wie bereits erwähnt, die konfigurierten semantischen Abbildungen, Kontextbedingungen und das konfigurierte Systemmodell automatisch erzeugt. Die Umsetzung der Konfiguration der Sprachparameter erfolgt jedoch manuell und ist in Abb. 7.89 gezeigt. Die Theorie `ExternalCDAS` wird von der generierten abstrakten Syntax `CDAS` (vgl. Definition C.22 auf Seite 309) verwendet. `ExternalCDAS` erfüllt die Konfiguration 7.88, Invarianten werden auf OCL-Invarianten abgebildet, für Methodenrümpfe und Werte werden Java-Anweisungen bzw. -Ausdrücke herangezogen.

`ExternalCDAS`

`Isabelle-Theorie`

```
theory ExternalCDAS
imports "$UMLP/gen/mc/javap/JavaPAS"
        "$UMLP/gen/mc/umlp/ocl/OCLAS"
begin

types InvariantExpression = OCLAS.OCLInvariant

types Body = JavaPAS.BlockStatement
types Value = JavaPAS.Expression

end
```

Abbildung 7.89: Konfiguration der Sprachparameter

Analog hierzu nutzt die semantische Abbildung für Klassendiagramme die in einer separaten Theorie definierte Abbildung für Invarianten, Methodenrümpfe und Werte. Diese Theorie ist in Abb. 7.90 dargestellt. Passend zur syntaktischen Konfiguration werden hier die semantischen Abbildungen für OCL-Invarianten, Java-Ausdrücke und -BlockStatements verwendet. Ebenso aufgebaut ist die nicht gezeigte Konfiguration der Kontextbedingungen für die Einbettung.

`ExternalCDSemantics`

`Isabelle-Theorie`

```
theory ExternalCDSemantics
imports "$UMLP/def/mc/umlp/cd/ExternalCDAS"
        "$UMLP/gen/mc/javap/semantics/JavaPSemantics"
        "$UMLP/gen/mc/umlp/ocl/semantics/OCLSemantics"
begin

fun mInvariantExpression :: "InvariantExpression ⇒ SystemModel ⇒ bool"
where
 "mInvariantExpression invar sm = mOCLInvariant invar sm"

fun mValue ::
 "Value ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ iVAL"
where
 "mValue m st vars oid th sm = mExpression m st vars oid th sm"

fun mBody :: "Body ⇒
 iSTATE ⇒ iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒ SystemModel ⇒ bool"
where
 "mBody b s s' vars oid th sm = mBlockStatement b s s' vars oid th sm"

end
```

Abbildung 7.90: Konfiguration der Semantik der Sprachparameter

Kapitel 8

Systemmodell-basierte Verifikation

Die vorgeschlagene Werkzeugunterstützung zur Definition einer Modellierungssprache im Allgemeinen und die kodierte UML/P-Semantik im Speziellen ermöglichen Verifikation in Isabelle. Die prinzipiellen Möglichkeiten sind in Abschnitt 8.1 beschrieben. In den dann folgenden Abschnitten werden konkrete Beispiele für die Verifikation in Isabelle auf Basis der UML/P-Semantik betrachtet. Die Beispiele sollen zeigen, dass mit dem vorgeschlagenen Vorgehen Verifikation auch praktisch möglich ist. Außerdem sind die Beispiele so gewählt, dass Teile aller definierten UML/P-Semantiken verwendet werden und so die Sinnhaftigkeit der semantischen Abbildung in Teilen plausibilisiert wird. Wir streben in den Beweisen auch keine hohe Automatisierung an, um die Beweisschritte im Detail nachvollziehen zu können. Dadurch können wir sicherstellen, dass das Ergebnis auch auf die antizipierte Art und Weise erreicht wurde und nicht etwa durch eine unabsichtliche Inkonsistenz in den Annahmen erzeugt wurde. Die abgeschlossenen Beweise können jederzeit automatisch überprüft werden, so dass nach Modifikationen an zugrunde liegenden Definitionen unerwünschte Nebenwirkungen wie mit einem Regressionstest aufgespürt werden können.

8.1 Verifikationsszenarien

Durch die tiefe Einbettung der abstrakten Syntax in Isabelle und durch die damit verbundene Formulierung der semantischen Abbildung direkt in Isabelle (vgl. Abschnitt 5.5.2) sind im Gegensatz zu einer flachen Einbettung (Eigenschaften des Modells werden direkt in Eigenschaften der Systeme übersetzt) neue Möglichkeiten für den Einsatz von Verifikationstechniken gegeben. Da Syntax und semantische Abbildung explizit in Isabelle kodiert sind, lassen sich auch Eigenschaften hierüber beweisen. Der gewählte prädikative Ansatz bei der Semantikdefinition und die gemeinsam genutzte semantische Domäne für alle Sprachen führen dazu, dass immer auch die integrierte Semantik einer Menge von Modellen gebildet und zur Verifikation eingesetzt werden kann. Welche prinzipiellen Möglichkeiten gegeben sind, wird in der folgenden Aufzählung aufgearbeitet. Die angegebenen Schemata können dazu dienen, den richtigen Ansatz für eigene Verifikationsaufgaben zu wählen.

- Basierend auf der Semantik eines konkreten Modells können Aussagen über spezifische Eigenschaften der durch das Modell beschriebenen Systeme verifiziert werden. Die Kodierung einer zu prüfenden Eigenschaft ϕ erfolgt direkt in Isabelle, also in der Form

```
  ∀ sm . sem m sm ⟶ φ sm
```

Methodisch gesehen kann ϕ als Anforderung verstanden werden, die ein Modell eines realen Systems erfüllen soll. Beispielsweise könnte eine Anforderung aus der Anwendungsdomäne gegeben sein, dass eine Fensterhebersteuerung einen Einklemmschutz aufweist. Welche Eigenschaft des Systems dazu nachgewiesen werden muss, hängt natürlich vom modellierten System ab. Die zu erfüllende Eigenschaft könnte sein, dass nach einem Signal „eingeklemmt“ nie das Signal „aufwärts“ weiter gesendet wird, sondern immer das Signal „abwärts“ folgt.

Eigenschaften aus einer Anwendungsdomäne lassen sich häufig in wiederkehrende Beweisverpflichtungen übersetzen:

- Variablen haben in einem Zustand immer einen bestimmten Wert
- ein Zustand ist (nicht) erreichbar
- bestimmte Nachrichtenfolgen sind ausgeschlossen oder kommen nach einem „Trigger“ immer vor

- Die Kodierung von Eigenschaften ϕ von Systemen kann aber auch in zusätzlichen Modellen erfolgen. Die zusätzlichen Modelle beschreiben gewünschte oder unerwünschte Strukturen, Verhalten oder Interaktionen. Zu zeigen ist dann, ob es Systeme gibt, die alle Eigenschaften erfüllen, bzw. umgekehrt, dass kein System alle Eigenschaften besitzt, der Schnitt aller Einzelsemantiken also leer ist.

  Typische Szenarien sind die (In)kompatibilität zum Beispiel von Klassendiagrammen und Objektdiagrammen, wobei ein Objektdiagramm hierbei eine (un)erwünschte Struktur eines Systemzustands beschreibt. Unter Verwendung der UML/P-Semantik aus dem vorherigen Kapitel kann beispielsweise die Gültigkeit des folgenden Lemmas untersucht werden:

  ```
  mCDDefinition cd1 sm ⟶
       (∀ s ∈ reachable sm . ¬ mODDefinition od1 sm s)
  ```

  Das bedeutet, dass alle Systeme, die die Klassendiagrammsemantik für das Klassendiagramm `cd1` erfüllen, keine erreichbaren Zustände besitzen, in denen das Objektdiagramm `od1` gilt. Die Statechart- und Sequenzdiagrammsemantik können ähnlich verwendet werden, um nachzuweisen, dass bestimmtes Systemverhalten (nie) eintritt.

- Zudem lassen sich allgemeingültige Systemeigenschaften ϕ unter einer semantischen Abbildung ableiten, die dann für alle Modellen Gültigkeit besitzen.

  ```
  ∀ sm . ∀ m . sem m sm ⟶ ϕ m sm
  ```

  Beispielsweise könnte für alle Klassendiagramme unter einer an Java angelehnten semantischen Abbildung gezeigt werden, dass alle Klassen die Oberklasse „Object“ besitzen.

- Durch die Kodierung der abstrakten Syntax lassen sich auch Aussagen über die Syntax der Sprache ableiten. Zum Beispiel können nicht automatisiert syntaktisch prüfbare Kontextbedingungen in Isabelle kodiert und nachgewiesen werden (vgl. Abschnitt 6.3 auf Seite 126). Hierzu ist für ein Modell `m` die Eigenschaft `wellformed m` zu prüfen.

- Wie bereits erwähnt lassen sich durch die tiefe Einbettung der Modellierungssprache auch Aussagen über die semantische Abbildung treffen. Wir können nachweisen, dass die semantische Abbildung konsistent ist, es also Systeme gibt, die die semantische Abbildung erfüllen:

  ```
  ∀ m . wellformed m ⟶ {sm | sm . sem m sm} ≠ {}
  ```

  Dabei ist auch ersichtlich, dass die semantische Abbildung dazu genutzt werden kann, Kontextbedingungen zu identifizieren, die dafür sorgen, dass allen wohlgeformten Modellen auch eine Semantik zugewiesen werden kann. Die Herausforderung ist, die Wohlgeformtheit `wellformed m` so zu definieren, dass die Menge der Systeme nie leer ist. Gerade im Zusammenhang mit der Variabilität einer Sprache können Inkonsistenzen entstehen. Es erscheint sinnvoll, zunächst solche Beweise für häufig verwendete Konfigurationen und Kombinationen von semantischen Abbildungen oder des Systemmodells durchzuführen.

- Das Systemmodell und seine Varianten können untersucht werden. Analog zur Konsistenz einer semantischen Abbildung ist eine Systemmodellkonfiguration konsistent, falls gilt

  ```
  {sm | sm . valid sm} ≠ {}
  ```

  Allgemeingültige Eigenschaften ϕ des Systemmodells können auf Basis des folgenden Schemas untersucht werden:

  ```
  ∀ sm . valid sm ⟶ φ sm
  ```

- Beziehungen zwischen konkreten Modellen lassen sich nachweisen. Ein Modell `m2` ist zum Beispiel eine Verfeinerung eines Modells `m1`, falls

  ```
  ∀ sm . sem m2 sm ⟶ sem m1 sm
  ```

  Es muss in diesem Fall also eine Teilmengenbeziehung zwischen den durch die Semantik implizierten Mengen von Systemen herrschen.

- Eine Verallgemeinerung des vorherigen Punktes ist, Modelloperatoren wie Verfeinerung, Komposition oder semantikerhaltende Transformation in ihrer Wirkung auf die abstrakte Syntax zu formalisieren und in Isabelle zu kodieren. Ob diese Operatoren den geforderten Eigenschaften genügen, kann dann mit Hilfe der Semantik nachgewiesen werden. Dies bedeutet dann eine Aussage über den Modelloperator und gilt für alle Modelle auf die der Operator anwendbar ist. Soll beispielsweise ein Transformationsregel `trafo` für Statecharts semantikerhaltend definiert sein, muss nachgewiesen werden, dass gilt:

  ```
  trafo :: SCDefinition ⇒ SCDefinition ⇒ bool
  ∀ sc1 sc2 . trafo sc1 sc2 ⟶
     { sm | sm . mSCDefinition sc1 sm } = { sm | sm . mSCDefinition sc2 sm}
  ```

- Unterschiedliche Varianten derselben Sprache sind im Allgemeinen nicht miteinander verträglich. Dies hat zur Folge, dass zum Beispiel Analyseergebnisse auf Basis der einen Konfiguration in der anderen Konfiguration nicht mehr gültig sind. In manchen Fällen können Ergebnisse aber erhalten bleiben, nämlich dann, wenn von einer „stärkeren“ zu einer „schwächeren“ Sprachvariante übergegangen wird.

Eine Konfiguration `K1` kann durch eine andere Konfiguration `K2` ersetzt werden, wenn für alle Modelle die resultierende Mengen der durch die semantischen Abbildungen (`semK1` und `semK2`) beschriebenen Systeme im Systemmodell in Teilmengenbeziehung stehen:

```
∀ m . { sm | sm . semK1 m sm } ⊇ { sm | sm . semK2 m sm}
```

Das bedeutet, dass alle Eigenschaften ϕ eines Modell `m`, die unter der Konfiguration `K1` gelten, auch unter der Konfiguration `K2` gelten und somit erhalten bleiben. Wir schlagen auf Grundlage dieser Überlegungen vor, Feature-Diagramme um eine weitere Beziehung anzureichern, die genau diesen Sachverhalt der *semantischen Verfeinerung* einer Sprachvariante dokumentiert (siehe Abschnitt 8.5).

8.2 Zirkuläre Vererbung in Klassendiagrammen

Als einfaches Beispiel für eine integrierte Semantik mehrerer Modelle betrachten wir die Semantik von zwei Klassendiagrammen. Die konkreten Modelle sind in Abb. 8.1 angegeben. Wir sehen, dass in jedem Modell eine Reihe von Klassen und ihre Subklassenbeziehung modelliert werden. Soll ein System beide Klassendiagramme gleichzeitig realisieren, ergibt sich eine zirkuläre Vererbungsbeziehung.

Die in Isabelle/HOL übersetzten Modelle sind aus Übersichtlichkeitsgründen nicht angegeben. In den folgenden Lemmas sind ihre Datentyp-Definitionen unter den Namen `ABC` und `CA` abgelegt.

ABC

Klassendiagramm

```
classdiagram ABC {

  class A;
  class B extends A;
  class C extends B;
}
```

CA

Klassendiagramm

```
classdiagram CA {

  class C;
  class A extends C;
}
```

Abbildung 8.1: Klassendiagramme ABC und CA

Voraussetzung für das Beispiel ist, dass in der verwendeten Systemmodellkonfiguration (z.B. Konfiguration 7.88, Seite 200) die Variante `AntiSymSub` ausgewählt ist, die die Antisymmetrie der Subklassenbeziehung fordert und damit Zyklen verbietet. In Lemma 8.2 zeigen wir zunächst, dass valide Systeme im Systemmodell in einer solchen Konfiguration tatsächlich keine zirkuläre Vererbung zulassen.

Lemma 8.2 (Keine zirkuläre Vererbung in validen Systemen)

SubNonCirc

Isabelle-Theorie

```
lemma SubNonCirc :
 "⟦valid sm; sub sm C2 C1; sub sm C1 C2⟧ ⟹ C1 = C2"
```

```
apply (unfold valid-def)
apply (unfold valid-AntiSymSub-def)
by auto
```

Das zweite Lemma 8.3 zeigt, dass in validen Systemen die Subklassenbeziehung transitiv ist. Diese Eigenschaft gilt variantenunabhängig für alle Systeme und wird aus der Theorie `Subclassing` gewonnen.

Lemma 8.3 (Transitive Vererbung in validen Systemen)

SubTrans

Isabelle-Theorie

```
lemma SubTrans :
  "⟦valid sm; sub sm C1 C2; sub sm C2 C3⟧ ⟹ sub sm C1 C3"
apply (unfold valid-def)
apply (unfold valid-base.simps)
apply (unfold valid-Subclassing.simps)
apply clarify
apply (unfold pSubTrans-def)
by best
```

Wir wollen nun zeigen, dass es aufgrund der zirkulären Vererbungsbeziehung keine Systeme gibt, die beide Klassendiagramme realisieren. Das bedeutet, die Menge der validen Systeme, die das erste Diagramm realisieren, geschnitten mit der Menge der validen Systeme, die das zweite Diagramm realisieren, ist leer. Die beiden Klassendiagramme sind also inkonsistent. Die Aussage und der Beweis dazu sind in Lemma 8.4 enthalten. Aus Übersichtlichkeitsgründen geben wir nur einen Teil des Beweises an. Die ausgelassenen Beweisschritte beschäftigen sich nur mit der Entfaltung der abstrakten Syntax, so dass die semantische Abbildung angewendet werden kann.

Lemma 8.4 (Keine validen Systeme für ABC und CA)

ABC-CA-circ

Isabelle-Theorie

```
lemma ABC-CA-circ :
 "{ sm | sm . valid sm ∧ mCDDefinition ABC sm} ∩
       { sm | sm . valid sm ∧ mCDDefinition CA sm} = {}"
apply auto
(* unfold definitions *)
apply (unfold mSuperClass-def)
apply simp
apply (unfold mNameList-def)
apply (frule SubTrans)
apply auto
apply (frule SubNonCirc)
apply auto
back
done
```

Der Beweis führt in Isabelle nach Anwendung von `auto` zu einem ersten Zwischenzustand:

```
mCDDefinition ABC sm ∧ valid sm ∧ mCDDefinition CA sm ⟶ False
```

Der Zwischenzustand zeigt an, dass wir eine widersprüchliche Aussage ableiten müssen, um das gewünschte Lemma zu zeigen. Nach Anwendung der Semantik für Superklassen erhalten wir unter anderem, dass Klasse `A` eine Subklasse von Klasse `C` ist. Mit Hilfe der Transitivität wird abgeleitet, dass auch `C` eine Subklasse von `A` ist. Durch die Antisymmetrie der Subklassenbeziehung entsteht schließlich:

```
valid sm ∧ sub sm (Class ''A'') (Class ''C'') ∧
sub sm (Class ''C'') (Class ''A'') ∧
  Class ''C'' = Class ''A'' ∧ ... ⟶ False
```

Dies führt zum gewünschten Widerspruch, da Klasse `C` nicht gleich Klasse `A` ist. Mit `back` im Beweis kann eine weitere Alternative ausprobiert werden, da bei der Umschreibung der Annahmen mit `auto` nicht die gewünschte Aussage erzielt wurde, weil es mehrere Möglichkeiten zur Anwendung der Transitivität gibt.

8.3 Verletzung einer Invariante im Klassendiagramm durch ein Objektdiagramm

Im zweiten Beispiel betrachten wir ein Klassendiagramm und ein Objektdiagramm. Das Klassendiagramm ist in Abb. 8.5 dargestellt und beschreibt die Struktur der Einträge eines vereinfachten Kalenders. Die Einträge (Klasse `Entry`, Zeile 12) haben einen Start und ein Ende. Diese bestehen gemäß der Klasse `DateTime` (Zeile 3) aus einem Datum und einer Zeit. Zudem ist eine Funktion `before` definiert, die prüft, ob das Datum zeitlich vor dem übergebenen Datum liegt. Neben den Klassen ist auch eine OCL-Invariante im Klassendiagramm definiert, die festlegt, dass für alle Einträge der Start vor dem Ende liegen muss.

Das Objektdiagramm in Abb. 8.6 beschreibt einen Zustand mit einem Kalendereintrag, dessen Startdatum `d1` zeitlich nach dem Enddatum `d2` liegt. Es ist zu erkennen, dass der Zustand die Invariante im Klassendiagramm verletzt. Dass dies auch mit Hilfe der UML/P-Semantiken nachweisbar ist, wird in den folgenden Lemmas gezeigt. Die Modellrepräsentation als Instanz der Isabelle-Datentypen ist wiederum nicht gezeigt. Die Klassendiagrammdefinition ist unter der Konstanten `Calendar` und die Objektdiagrammdefinition unter `Calendar1` abgelegt.

Zunächst definieren wir in Definition 8.7 einige Schreibabkürzungen. `startDate oid s` (Zeile 2) ist eine Abkürzung für die Abfrage des Startdatums aus dem Datenzustand für einen gegebenen Zustand `s` und ein Objekt `oid` im System. Analog werden die weiteren Abkürzungen `endDate` (Zeile 7), `startTime` (Zeile 12) und `endTime` (Zeile 17) aufgebaut.

Definition 8.7 (Definition einiger Abkürzungen)

Abkürzungen

Isabelle-Theorie

```
1  constdefs
2    startDate-def : "startDate oid s ==
3     toInt (the (getAttr (
4            the (dsOf s (toOid (
5             the (getAttr (the (dsOf s oid), ''start''))))),
6            ''date'')))"
7    endDate-def : "endDate oid s ==
8     toInt (the (getAttr (
```

Calendar

Klassendiagramm

```
classdiagram Calendar {

  class DateTime {
    int date;
    int time;
    boolean before(DateTime d) {
      return date < d.date
           || date == d.date && time < d.time;
    }
  }

  class Entry {
    DateTime start;
    DateTime end;
  }

  [
    context Entry e inv: e.start.before(e.end);
  ];

}
```

Abbildung 8.5: Klassendiagramm Calendar

Calendar1

Objektdiagramm

```
objectdiagram Calendar1 {

  e : Entry {
    start = d1;
    end   = d2;
  }

  d1 : DateTime {
    date = 3;
    time = 1;
  }

  d2 : DateTime {
    date = 2;
    time = 2;
  }

}
```

Abbildung 8.6: Objektdiagramm Calendar1

```
         the (dsOf s (toOid (
          the (getAttr (the (dsOf s oid), ''end''))))),
         ''date'')))"
  startTime-def : "startTime oid s ==
   toInt (the (getAttr (
         the (dsOf s (toOid (
          the (getAttr (the (dsOf s oid), ''start''))))),
         ''time''))))"
  endTime-def : "endTime oid s ==
   toInt (the (getAttr (
         the (dsOf s (toOid (
          the (getAttr (the (dsOf s oid), ''end''))))),
         ''time''))))"
```

Lemma 8.8 zeigt, dass wir aus der semantischen Abbildung für das Klassendiagramm ableiten können, dass für alle erreichbaren Zustände, die ein Objekt `Oid ''e''` der Klasse `Entry` enthalten, die Invariante gilt. Das heißt, das Startdatum liegt vor dem Enddatum und falls Start- und Enddatum übereinstimmen, muss die Startzeit vor der Endzeit liegen. Die Durchführung des Beweises beschränkt sich auf Entfaltung der Definitionen und kann mit `auto` abgeschlossen werden.

Lemma 8.8 (Invariante des Klassendiagramms gilt)

InvariantHolds

Isabelle-Theorie

```
lemma InvariantHolds: "⟦mCDDefinition Calendar sm⟧ ⟹
     ∀s∈reachable sm.
       Oid ''e'' ∈ oids s ∧
       VOid (Oid ''e'') ∈ CAR sm (TO (Class ''Entry'')) ⟶
         startDate (Oid ''e'') s  < endDate (Oid ''e'') s ∨
         startDate (Oid ''e'') s = endDate (Oid ''e'') s ∧
         startTime (Oid ''e'') s < endTime (Oid ''e'') s"
(* unfold definitions *)
by auto
```

In Lemma 8.9 zeigen wir, dass für einen erreichbaren Zustand `s`, der dem Objektdiagramm entspricht, tatsächlich die Werte für Start und Ende angenommen werden. Die triviale Beweisdurchführung ist wiederum nicht dargestellt.

Lemma 8.9 (Attributwerte des Objektdiagramms)

AttrValues

Isabelle-Theorie

```
lemma AttrValues :
   "⟦s ∈ reachable sm; mODDefinition Calendar1 s sm⟧ ⟹
     Oid ''e'' ∈ oids s ∧
     VOid (Oid ''e'') ∈ CAR sm (TO (Class ''Entry'')) ∧
     startDate (Oid ''e'') s = 3 ∧
     startTime (Oid ''e'') s = 1 ∧
     endDate (Oid ''e'') s =  2 ∧
     endTime (Oid ''e'') s  = 2 "
(* unfold definitions *)
by auto
```

Wir können schließlich unser gewünschtes Ergebnis in Lemma 8.10 formulieren und beweisen: Gilt die semantische Abbildung für das Klassendiagramm folgt hieraus, dass in keinem erreichbaren Zustand die Eigenschaften aus der semantischen Abbildung des Objektdiagramms erfüllt sind. Das Objektdiagramm verletzt immer die OCL-Invariante. Der Beweis verwendet die vorher gezeigten Lemmas und kann nach Elimination eines Allquantors (Zeile 8) mit `auto` abgeschlossen werden.

Lemma 8.10 (Invariante des Klassendiagramms im Objektdiagramm verletzt)

InvariantViolated

Isabelle-Theorie

```
1  lemma InvariantViolated :
2     "mCDDefinition Calendar sm ⟶
3          (∀ s ∈ reachable sm . ¬ mODDefinition Calendar1 s sm)"
4  apply auto
5  apply (frule InvariantHolds)
6  apply (frule AttrValues)
7  apply auto
8  apply (erule ballE)
9  by auto
```

Damit das Beispiel wie angegeben durchgeführt werden kann, müssen OCL- und Java-Ausdrücke ausgewertet werden. Die Spracheinbettung muss also entsprechend konfiguriert sein. Weiterhin benutzten wir die einfache Methodenausführung als Query (siehe hierzu Variante 7.49 der Klassendiagrammsemantik und die Systemmodellvariante 4.52). Weitere Varianten, wie zum Beispiel die einfache Umsetzung von Objektnamen auf Objekte (`d1` wird zum Objektidentifikator `Oid ''d1''`) werden verwendet. Die direkte Umsetzung von Namen (das heißt, eine Entfaltung der betroffenen Funktion) ist nicht unbedingt erforderlich, dient aber dazu, die Beweisschritte nachvollziehbarer zu machen. Das beschriebene Beispiel ist ein typisches Anwendungsszenario für Werkzeuge wie den Alloy Analyzer [ABGR10, ABGR07] oder das *UML-based Specification Environment* (USE) [GKH09, GBR05, GBR07], die es erlauben, Klassendiagramme mit OCL-Invarianten anhand von Objektdiagrammen oder anderweitig formulierten Fakten automatisch zu validieren. Ein weiteres Werkzeug für diese Zwecke ist die Maude-basierte Implementierung ITP/OCL [CE06]. Diese Werkzeuge haben den Vorteil, bestimmte OCL-Invarianten automatisch zu prüfen. Unser Ansatz hingegeben besitzt den Vorteil, nicht auf automatisch prüfbare Invarianten festgelegt zu sein, da ein allgemeiner Beweis interaktiv erfolgt.

8.4 Unerlaubte Interaktion im Sequenzdiagramm bezüglich eines Statechart

In diesem Beispiel untersuchen wir die Kombination eines Statechart mit einem Sequenzdiagramm. Das Statechart `Auth` in Abb. 8.11 stellt einen sehr einfachen Authentifizierungsmechanismus dar. Ausgehend vom Startzustand `NonAuth`, der für eine noch nicht erfolgte Authentifizierung steht, kann die Nachricht `auth` empfangen werden. Die Vorbedingung legt fest, ob als Folgezustand `Auth` eingenommen oder im Zustand `NonAuth` verblieben wird, je nachdem, ob der empfangene Wert `key` dem eigenen Schlüssel `ownkey` entspricht oder nicht. Der Aufrufer wird positiv oder negativ benachrichtigt.

Auth

Statechart

```
package mysys;

statechart Auth Auth  {

  initial state NonAuth;

  state Auth;

  NonAuth -> NonAuth : [key != ownkey] auth(key) / {return false;}

  NonAuth -> Auth : [key == ownkey] auth(key) / {return true;}

}
```

Abbildung 8.11: Statechart Auth

Auth1

Sequenzdiagramm

```
package mysys;

<<match="visible">> sequencediagram Auth1  {

  u:User;
  a:Auth;

  u -> a : auth(someKey);
  <a : [someKey != ownkey]>;
  u <- a : return true;

}
```

Abbildung 8.12: Sequenzdiagramm Auth1

Das Sequenzdiagramm `Auth1` in Abb. 8.12 stellt ein unerwünschtes Systemverhalten dar. Wird die Nachricht `auth` an ein Objekt der Klasse `Auth` geschickt und stimmen die Schlüssel nicht überein, wird laut Sequenzdiagramm trotzdem eine positive Antwort gesendet. Verhält sich ein Objekt der Klasse `Auth` im System gemäß der im Statechart angegebenen Spezifikation, darf diese Interaktion nicht möglich sein. Dies zu zeigen ist das Ziel dieses Abschnitts.

Zu diesem Zweck werden zunächst zwei Hilfslemmas benötigt. Das Lemma 8.13 zeigt, dass in einem validen System ein Objekt `Oid ''u''` einer Klasse `Class ''User''` auch im Universum `UOID` enthalten ist. Dies Eigenschaft ist aus der Theorie `Class` ableitbar.

Lemma 8.13 (Hilfslemma 1, UserOid)

UserOid

Isabelle-Theorie

```
lemma UserOid:
  "⟦valid sm; Class ''User'' ∈ UCLASS sm;
    Oid ''u'' ∈ Class.oids sm (Class ''User'')⟧ ⟹
    Oid ''u'' ∈ UOID sm"
apply (unfold valid-def)
apply (unfold valid-base.simps)
apply (unfold valid-Class-def)
apply (unfold pOids.simps)
apply clarify
by best
```

Das zweite Lemma beweist die Aussage, dass in einem validen System eine laufende Methode `''auth''` in einem Zustand `s` nicht gleichzeitig beendet ist.

Lemma 8.14 (Hilfslemma 2, RunNotFin)

RunNotFin

Isabelle-Theorie

```
lemma RunNotFin :
  "⟦valid sm; running ''auth'' (Oid ''a'') (Oid ''u'') th s⟧ ⟹
    ¬ notExecuting ''auth'' s (Oid ''a'') (Oid ''u'') th"
apply (unfold valid-def)
apply (unfold valid-base.simps)
apply (unfold valid-State.simps)
apply (unfold runningNotFinished.simps)
by best
```

Die Semantiken von Statecharts und Sequenzdiagrammen machen Aussagen über Systemabläufe. Dabei muss das spezifizierte Verhalten nicht unbedingt in einem Systemablauf vorkommen. Die semantische Abbildung für Statecharts prüft zum Beispiel, ob eine Nachricht, die einem Event im Statechart entspricht, in einem Zustand vorhanden ist. Ist dies nicht der Fall, wird der Zustand ignoriert, da er offensichtlich nicht Teil des Verhaltens für das Statechart ist. Wir müssen daher einige Bedingungen für einen Systemablauf definieren und damit das gewünschte Verhalten erzwingen, ohne es jedoch schon exakt festzulegen. Eine einfache Möglichkeit ist in Definition 8.15 angegeben. Die Definition formuliert die folgenden Eigenschaften für einen Systemablauf:

- Der Systemablauf besteht vereinfachend nur aus drei Zuständen. Diese Einschränkung macht den Beweis einfacher, ändert aber nichts an der Allgemeingültigkeit, da irrelevant Zwischenzustände durch die semantische Abbildung entfernt würden.
- Weiterhin gibt es einen Thread `th`, einen Parameterwert `vr` und einen Rückgabewert `ret`, so dass der Nachrichtenpuffer eines Objekts `Oid ''a''` im ersten Zustand `s1` den Aufruf der Methode `auth` enthält.
- Der Nachrichtenpuffer in Zustand `s2` des Objekts `Oid ''u''` enthält die Rückgabe der Methode `auth`.
- Im Zwischenzustand `se` werden keinen Nachrichten ausgetauscht, es erfolgt aber die Methodenausführung (`running`), die in Zustand `s2` wieder beendet ist (`notExecuting`).

Obwohl wir relativ genaue Angaben über die Eigenschaften des Trace gemacht haben, ist offen geblieben, welche Werte `vr` und `ret` annehmen können. Wir kommen auf diese Werte gleich noch einmal zurück.

Definition 8.15 (Eigenschaften eines Trace)

TraceProp

Isabelle-Theorie

```
constdefs traceProp-def :
  "traceProp sm trc ==
    ∃ s1 ∈ STATE sm . ∃ se ∈ STATE sm . ∃ s2 ∈ STATE sm .
      trc = [s1,se,s2] ∧ trc ∈ TRACES sm ∧
      (∃ th ∈ UTHREAD sm . ∃ ret ∈ UVAL sm . ∃ vr ∈ UVAL sm .
        the (msOf s1 (Oid ''a'')) =
         [(Oid ''a'', ''auth'', [vr], Oid ''u'', Some th)] ∧
        the (msOf s2 (Oid ''u'')) =
         [(Oid ''u'', ''auth'', [ret], Oid ''a'', Some th)] ∧
        the (msOf se (Oid ''a'')) = [] ∧
        the (msOf se (Oid ''u'')) = [] ∧
        running ''auth'' (Oid ''a'') (Oid ''u'') th se ∧
        notExecuting ''auth'' s2 (Oid ''a'') (Oid ''u'') th)"
```

Das Ziel dieses Abschnitts ist in Lemma 8.16 formuliert und bewiesen. Ist ein valides System und ein Trace des Systems mit den oben beschriebenen Eigenschaften gegeben und erfüllt das System die Statechart-Spezifikation, dann gilt das Sequenzdiagramm für den Trace nicht. Das heißt, die darin beschriebene Interaktion kann in einem gemäß des Statechart implementierten System nicht auftauchen.

Lemma 8.16 (Nachrichtenaustausch gemäß Sequenzdiagramm tritt nicht auf)

IllegalMsg

Isabelle-Theorie

```
lemma IllegalMsg :
  "valid sm ∧
   traceProp sm trc ∧
   mSCDefinition Auth sm
     ⟶ ¬ mSDDefinition Auth1 trc sm"
apply (unfold traceProp-def)
(* unfold definitions and instantiate quantifiers *)
by auto
```

Aus der Semantik des Statechart ergeben sich während der Beweisführung folgende Möglichkeiten für die Werte von `vr` und `ret` gemäß den Transitionen im Statechart:

```
vr = the (getAttr (the (dsOf s1 (Oid ''a'')), ''ownkey'')) ∧
VBool True = ret
∨
vr ≠ the (getAttr (the (dsOf s1 (Oid ''a'')), ''ownkey'')) ∧
VBool False = ret
```

Entweder entspricht `vr` dem Wert des Attributs `ownkey` des Objekts `a` und es wird `True` zurück geliefert oder `vr` ist ungleich dem Attributwert und die Rückgabe ist `False`. Aus dem Sequenzdiagramm ist jedoch lediglich die folgende Belegung erlaubt:

```
vr ≠ the (getAttr (the (dsOf s1 (Oid ''a'')), ''ownkey'')) ∧
ret = VBool True
```

Der erste Fall im Statechart scheidet für die Interaktion im Sequenzdiagramm aus und der zweite liefert einen Widerspruch, so dass Statechart und Sequenzdiagramm nicht vereinbar sind. Wir brauchen keine konkreten Werte für `vr` oder `ret` anzugeben, der Nachweis gilt daher für alle möglichen Werte, die die angegebenen Bedingungen erfüllen. Diese Allgemeingültigkeit ist bezeichnend für die Verwendung eine Theorembeweisers im Gegensatz, zum Beispiel zu einem model checker, wo meist konkrete Werte für Gegenbeispiele gefunden werden müssen. In diesem einfachen Fall ist natürlich durch Äquivalenzklassenbildung (gleich oder ungleich `ownkey`) das Finden von Werten einfach.

Im vorherigen Beispiel wurden zunächst die Semantik des Klassendiagramms und anschließend die Semantik des Objektdiagramms einzeln untersucht und die Ergebnisse später zusammengeführt. In diesem Beispiel können die eben diskutierten möglichen Werte nicht separat abgeleitet werden, da jeweils Informationen aus dem Statechart und dem Sequenzdiagramm gemeinsam verwendet werden müssen, der Beweis ist daher nicht modular aufgebaut.

Auch in diesem Beispiel wurden eine Reihe von Varianten verwendet. Insbesondere wurde bei der semantischen Abbildung des Sequenzdiagramms der Stereotyp `match="visible"` berücksichtigt. Des Weiteren wird wieder eine einfache eins zu eins Übersetzung von Objekten des Sequenzdiagramms in Objekte des Systems benutzt.

8.5 Vergleich von Sprachvarianten

Wir haben in dieser Arbeit das Rahmenwerk geschaffen, um Varianten einer Sprache zu definieren und zu konfigurieren. Zu Beginn des Kapitels haben wir bereits die Möglichkeit erörtert, sprachliche Varianten formal zu vergleichen und dabei ihre Beziehungen untereinander zu untersuchen. In diesem Abschnitt geben wir hierfür ein einfaches Beispiel auf Basis der Semantik einer Sprache an. Die Vielfalt der bisher definierten Varianten ist allerdings gering, so dass wir zunächst eine Systemmodellvariante `TypeSafeOpsStrict` in Variante 8.17 einführen. Die Variante wollen wir mit der Variante B.36 (`TypeSafeOps`) vergleichen. Informell ist aus den Definitionen ersichtlich, dass Variante 8.17 strikter ist und keine ko- bzw. kontravariante Anpassung der Operation erlaubt (Gleichheit statt Teilmengenbeziehung in Zeilen 9 und 10).

Variante SysMod 8.17 (Strikte, typsichere Vererbung von Operationen)

TypeSafeOpsStrict

Isabelle-Theorie

```
1  fun valid-TypeSafeOpsStrict :: "SystemModel ⇒ bool"
2  where
3   "valid-TypeSafeOpsStrict sm = (
4      ∀ op1 ∈ UOPN sm . ∀ C ∈ UCLASS sm .
5        (sub sm C (classOf sm op1)) ⟶
6          (∃ op2 ∈ UOPN sm .
7            classOf sm op2 = C ∧
8            nameOf op1 = nameOf op2 ∧
9            params sm op1 = params sm op2 ∧
10           CAR sm (resType sm op2) = CAR sm (resType sm op1)
11         )
12   )"
```

In Lemma 8.18 zeigen wir nun formal, dass Systeme, die die stärkere Systemmodellvariante `TypeSafeOpsStrict` verwenden, eine Teilmenge der Systeme bilden, die die schwächere Variante `TypeSafeOps` verwenden.

Lemma 8.18 (Implikation der Varianten)

TypeSafeImpl

Isabelle-Theorie

```
lemma TypeSafeOpsImplStrict :
    "{sm | sm . valid-TypeSafeOpsStrict sm}
             ⊆ {sm | sm . valid-TypeSafeOps sm}"
apply (unfold valid-TypeSafeOps.simps)
apply (unfold valid-TypeSafeOpsStrict.simps)
by best
```

Das bedeutet also, dass Eigenschaften, die für Systeme mit der schwächeren Variante gelten, automatisch auch für die stärkere Variante erfüllt sind. Es bietet sich an, diesen Zusammenhang auch im Feature-Diagramm zu dokumentieren. Abb. 8.19 zeigt einen Ausschnitt des Variationspunkts *vControl* des Systemmodells mit der neu definierten Variante und der zusätzlich dokumentierten *semantischen Verfeinerungsbeziehung* zwischen Varianten.

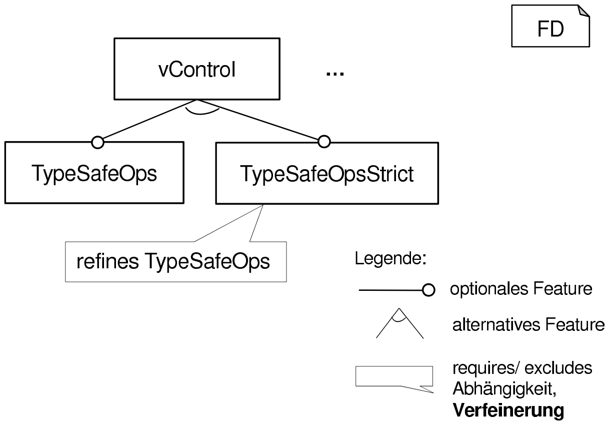


Abbildung 8.19: Variante TypeSafeOpsStrict verfeinert Variante TypeSafeOps

Solche Beziehungen können nicht nur für Systemmodellvarianten, sondern auch für Varianten semantischer Abbildungen nachgewiesen werden. Für weitergehende Untersuchungen ist es aber sinnvoll, zunächst weitere Infrastruktur (Hilfslemmas) für semantische Abbildungen aufzubauen, da ein ad hoc Beweis nur auf Basis der Definitionen schwierig durchzuführen ist.

Neben der Möglichkeit eine Konfiguration auf Konformität zu prüfen, erlaubt uns die neue Annotation nun, zwei Konfigurationen syntaktisch anhand des Feature-Diagramms auf semantische Verfeinerung hin zu untersuchen. Ein konkretes Anwendungsszenario ist die Werkzeugintegration und damit zum Beispiel die Frage, ob Analyseergebnisse, die mit dem ersten Werkzeug und einer bestimmten Sprachvariante erzeugt wurden, im zweiten Werkzeug zur Testfall- oder Codegenerierung mit einer anderen Sprachvariante erhalten bleiben. Voraussetzung hierfür ist, dass beide Werkzeuge die Sprachvarianten korrekt implementieren.

Teil IV

Epilog

Kapitel 9

Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse dieser Arbeit kurz zusammengefasst und anschließend ein Ausblick auf mögliche aufbauende Arbeiten gegeben.

Zusammenfassung der Ergebnisse

Ziel dieser Arbeit war die vollständige, formale Definition objektbasierter Modellierungssprachen, um die Effektivität einer modellbasierten Entwicklung zu verbessern, da sowohl die Kommunikation zwischen Entwicklern als auch die Austauschbarkeit von Modellen zwischen interoperablen Werkzeugen von einer präzisen Definition der Modellierungssprache profitieren kann.

Ein wichtiger und im Fokus dieser Arbeit stehender Bestandteil einer Sprachdefinition ist die formale Semantik der Sprache. Sie hilft, eine einheitliche Interpretation der Modelle einer Sprache herzustellen und somit Missverständnisse zu vermeiden. Der in dieser Arbeit verwendete Ansatz begegnet den in Kapitel 1 identifizierten Herausforderungen:

- Die Zugänglichkeit und Präzision der Semantik wird dadurch erreicht, dass das Systemmodell auf Basis einfacher mathematischer Grundlagen definiert wird. Mengen von Systemen des Systemmodells bilden die Semantik eines Modells und stellen so einen Bezug zu möglichen Implementierungen her.

- Der prädikative Ansatz bei der Formulierung der semantischen Abbildung unterstützt unvollständige Modelle. Das ist wichtig, da wir Modelle nicht ausschließlich als detaillierte Implementierungsbeschreibungen verstehen wollen, sondern sie auch zur abstrakten Spezifikation einsetzen möchten.

- Die Komposition von Semantiken von Modellen derselben oder unterschiedlicher Modellierungssprachen kann einfach durch Schnittmengenbildung erfolgen, da eine mengenwertige Abbildung und eine einzige semantische Domäne verwendet werden.

- Die Sprachdefinition ist flexibel durch Varianten anpassbar.

In dieser Arbeit wurde die Variabilität in Modellierungssprachen systematisch klassifiziert. Präsentationsvariabilität hilft Anwendern komfortabel mit der Sprache umzugehen, spielt aber für die Semantik der Sprache keine Rolle. Syntaktische Variabilität muss auch in der Semantik berücksichtigt werden und ist durch Varianten in Stereotypen, Sprachparametern und Spracheinschränkungen definiert. Stereotypen können genutzt werden, um semantische Variabilität syntaktisch zu kodieren, um sie damit einem Modellierer direkt zugänglich zu machen. Semantische

Variabilität ist unterteilt in Varianten der semantischen Domäne und Varianten der semantischen Abbildung. Erstere können anschaulich als Eigenschaften eines zu Grunde liegenden Laufzeitsystems oder einer Zielplattform verstanden werden. Letztere bilden Konfigurationsparameter bei der Übersetzung in diese Zielplattform. Zur Dokumentation der Varianten einer Sprache und der Abhängigkeiten zwischen Varianten wurden Feature-Diagramme vorgeschlagen. Eine Konfiguration der Sprache, die konform zu den entsprechenden Feature-Diagrammen ist, bezeichnet dann die Menge aller gültigen Definitionen.

Als gemeinsam genutzte semantische Domäne für objektbasierte Modellierungssprachen haben wir das Systemmodell definiert, das objektbasierte Systeme charakterisiert. Ein System im Systemmodell ist ein nicht-deterministisches Transitionssystem. Die Zustände beinhalten die Daten-, Kontroll- und Nachrichtenspeicher aller Objekte im System, die asynchron über Nachrichtenaustausch kommunizieren. Objekte gehören zu Klassen, die Attribute, Operationen und Methoden besitzen können. Nebenläufigkeit in Objekten und im System wird durch Threads modelliert.

Zur Unterstützung der Definition einer Modellierungssprache wurde eine Werkzeugkette vorgeschlagen. Wie in Abb. 9.1 dargestellt, wird die konkrete Syntax in MontiCore-Grammatiken definiert. Der Generator MontiCore-isabelle erstellt Isabelle/HOL-Datentypen, die die abstrakte Syntax in Isabelle explizit repräsentieren. Auf ihr können Kontextbedingungen und Varianten von Kontextbedingungen als separate Isabelle-Theorien definiert werden. Das Systemmodell und seine Varianten wurden in Isabelle kodiert. Basierend auf der tiefen Einbettung der Syntax und des kodierten Systemmodells kann auch die Definition der semantischen Abbildung und ihrer Varianten direkt in Isabelle erfolgen.

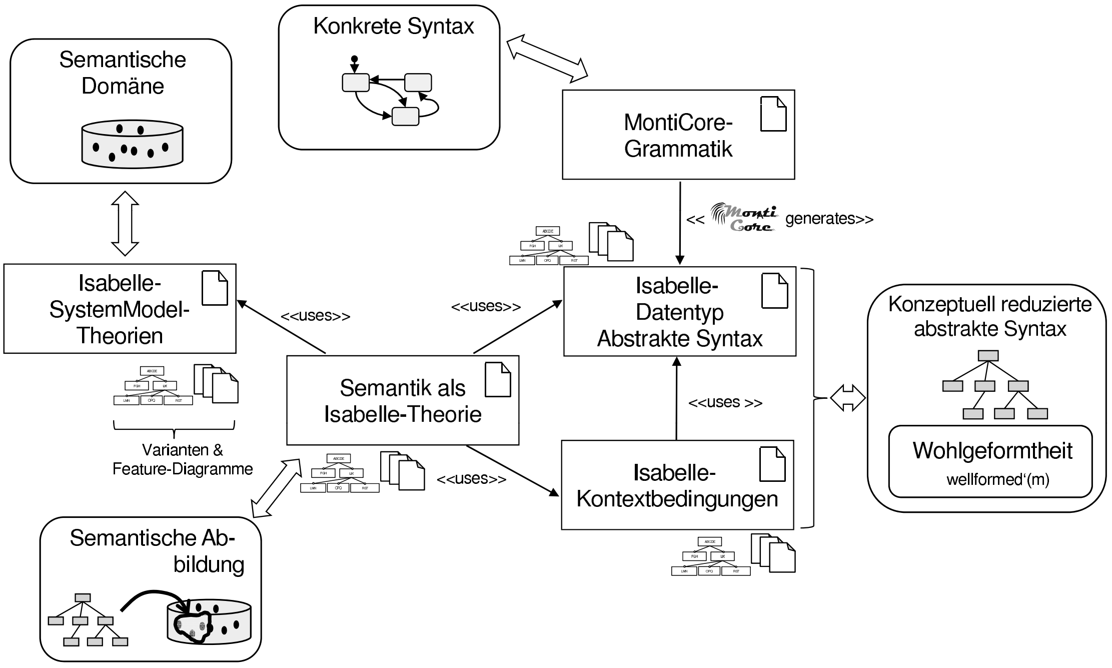

Konkrete Syntax
Semantische Domäne
MontiCore-Grammatik
<< MontiCore generates>>
Isabelle-SystemModel-Theorien
<<uses>>
<<uses>>
Isabelle-Datentyp Abstrakte Syntax
Konzeptuell reduzierte abstrakte Syntax
Semantik als Isabelle-Theorie
<<uses >>
Varianten & Feature-Diagramme
Wohlgeformtheit
wellformed'(m)
<<uses>>
Isabelle-Kontextbedingungen
Semantische Abbildung

Abbildung 9.1: Zuordnung der Sprachbestandteile zur Werkzeugunterstützung

Varianten werden in textuellen Feature-Diagrammen dokumentiert. Für ebenfalls textuell erstellte Konfigurationen einer Sprache ist eine automatische Prüfung der Konformität zu den entsprechenden Feature-Diagrammen mit Hilfe des Werkzeugs ThyConfig möglich. Gewählte Varianten, die als Isabelle-Theorien umgesetzt werden, werden dabei automatisch zu integrierten Theorien zusammengeführt, so dass die konfigurierte Sprache in Isabelle für die weitere Verwendung zur Verfügung steht. Konkrete Modelle lassen sich durch einen automatisch erzeugten Generator in Instanzen der abstrakten Syntax in Isabelle übersetzen.

Die Semantik von UML/P wurde auf Grundlage bereits existierender MontiCore-Grammatiken definiert. Dies diente gleichzeitig zur Demonstration der werkzeugunterstützten Vorgehensweise. Abb. 9.2 zeigt die formalisierten Diagrammarten. Dabei ist zu beachten, dass Java/P und OCL/P nur rudimentär behandelt wurden, so dass sie zur Einbettung in andere Diagrammarten und für die durchgeführten Verifikationsbeispiele ausreichend waren. Alle semantischen Abbildungen verwenden das Systemmodell als semantische Domäne und können daher wie beschrieben leicht integriert werden.

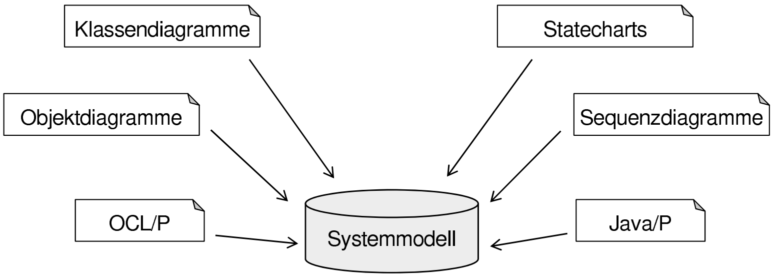


Abbildung 9.2: Formalisierte Diagrammarten dieser Arbeit

Weiterhin wurden Verifikationsszenarien auf Basis der vollständigen Definition der Modellierungssprache diskutiert und einige Beispiele hierfür angegeben. Verwendet wurden konkrete Modelle aller Diagrammarten, die automatisch in Isabelle übersetzt wurden. Eigenschaften der integrierten Semantik sowohl von Klassen- und Objektdiagrammen, als auch von Statecharts und Sequenzdiagrammen wurden untersucht. Zusätzlich wurde gezeigt, wie wir unseren Ansatz nutzen können, um Sprachvarianten semantisch zu vergleichen. Die Beispiele zeigen die praktische Durchführbarkeit der Verifikation in Isabelle und plausibilisieren gleichzeitig die entwickelte UML/P-Semantik.

Ausblick

Variationspunkte und Varianten der UML sind im OMG-Standard nicht systematisch erfasst. Für semantische Variationspunkte fehlt eine Trennung zwischen Varianten einer Abbildung und Varianten einer semantischen Domäne. Es bietet sich also an, unsere Methode der Dokumentation von Varianten in Feature-Diagrammen auf den OMG-Standard anzuwenden, um hierdurch einen besseren Überblick über Variationspunkte und folglich genauere Konformitätserklärungen beispielsweise durch Werkzeughersteller zu UML-Varianten zu ermöglichen. Interessant hierbei wäre auch die Betrachtung der UML-Profilbildung. Es könnte ein Mechanismus geschaffen werden, UML-Profile auf Basis von Konfigurationen der UML-Teilsprachen zu erstellen und zu verwalten. Da wir in dieser Arbeit von textuellen Sprachen ausgegangen sind, wäre es im Zusammenhang mit OMG-Standards auch sinnvoll, einen Formalismus bzw. Werkzeug der Metamodellierung wie MOF [OMG06a] oder EMF [BSM$^+$03] in der Werkzeugunterstützung zu berücksichtigen, also Metamodelle nach Isabelle zu übersetzen.

Zudem können weitere Modellierungssprachen wie zum Beispiel Aktivitätsdiagramme mit unserem Ansatz definiert werden. Hierfür gäbe es mehrere Möglichkeiten. Ist die Sprache verwandt zu einer bereits formalisierten UML/P-Diagrammart, ist eine syntaktische Transformation in diese Diagrammart ausreichend, um die Bedeutung zu klären. Handelt es sich um eine objektbasierte Sprache, kann eine Abbildung in das Systemmodell vorgenommen werden. Prinzipiell ist unser Ansatz jedoch nicht auf die Verwendung des Systemmodells als semantische Domäne festgelegt. Beliebige andere, in Isabelle kodierte semantische Domänen könnten gleichermaßen verwendet werden, wobei dann eine einfache Integration der Semantik mit anderen Sprachen nicht mehr möglich ist.

Für die UML/P-Semantiken kann eine höhere Detailtiefe für die einzelnen Diagrammarten erreicht werden. Insbesondere könnte die Variabilität jeder Diagrammart weiter untersucht und dabei zusätzliche Varianten definiert werden. Dann wäre es sinnvoll, den Vergleich von Varianten zum Beispiel bezüglich semantischer Verfeinerung genauer zu betrachten. Als Vorlage hierfür könnten Varianten in existierenden Werkzeugen oder wiederum der UML-Standard mit seinen zahlreichen Formalisierungsversuchen dienen. Darauf aufbauend könnte der in [CDGR07] implementierte Simulator für UML-Modelle auf Basis des Systemmodells, der UML/P-Semantik und deren Varianten konfigurabel erweitert werden.

Die Semantik der UML/P-Teilsprachen sind einzeln definiert. Die Integration wird über das Systemmodell erreicht. In welcher Art und Weise sich Modelle verschiedener Sprachen semantische beeinflussen, welche absichtlichen oder unabsichtlichen Interaktionen (in bestimmten Varianten) auftreten, wurde in dieser Arbeit nicht weiter untersucht. Es besteht aber die Möglichkeit, Wechselwirkungen zwischen Modellierungssprachen zu analysieren, indem speziell die sich semantisch überlappenden Konstrukte betrachtet werden. Die Erkenntnisse könnten auch zu neuen Inter-Kontextbedingungen oder methodischen Vorgaben wie Modellierungsrichtlinien führen.

Die praktische Durchführbarkeit der Verifikation wurde in dieser Arbeit an einfachen Beispielen gezeigt. Generell ist die Verwendung eines Theorembeweisers zur semi-automatischen Verifikation für realistische Anwendungen eine Herausforderung und zunächst nur für hochsicherheitskritische Systeme von praktischem Interesse. Trotzdem wäre es wünschenswert, eine höhere Automatisierung bei der Beweisdurchführung zu erreichen. Ein Schritt in diese Richtung wäre der Aufbau einer umfangreichen Sammlung von Lemmas für die UML/P-Semantiken. Können Standardaufgaben bei der Verifikation weitgehend automatisiert werden, ist sogar „Verifikation as-a-Service“ vorstellbar, also das Zurverfügungstellen eines Verifikationsdienstes. Basierend auf konkreten Modellen könnte exploriert werden, inwieweit sich hilfreiche Lemmas auch generieren lassen. Für weitergehende Aussagen über das Verhalten von Systemen fehlt die Verbindung zur Theorie der Ströme und stromverarbeitender Funktionen. Es ist wünschenswert, diese Lücke zu schließen. Eine weitere Herausforderung bei der Verifikation betrifft die Konsistenz der Definitionen. Für Standardkonfigurationen einer Sprache ist es sinnvoll, die Konsistenz explizit nachzuweisen.

Über die dargestellte Werkzeugunterstützung hinaus wäre es interessant, weitere Dienste zur kollaborativen Definition einer Modellierungssprache anzubieten. Genau wie die eigentlichen Entwicklungsartefakte eines Softwaresystems, könnten alle Artefakte der Sprachdefinition beispielsweise einer Versionskontrolle unterliegen. Issue-Tracking Systeme und Metriken wie Abdeckungskriterien könnten den Stand der Sprachentwicklung festhalten. Zusammen mit regelmäßigen, automatisiert erstellten Berichten könnte dies für eine hohe Qualität bei der Sprachdefinition sorgen.

Literaturverzeichnis

[ABGR07] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *Model Driven Engineering Languages and Systems (MoDELS) 2007 (Proceedings)*, volume 4735 of *LNCS*, pages 436–450. Springer, 2007.

[ABGR10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9(1):69–86, 2010.

[ADN+06] Joanne M. Atlee, Nancy A. Day, Jianwei Niu, Eunsuk Kang, Yun Lu, David Fung, and Leonard Wong. Metro: An Analysis Toolkit for Template Semantics. Technical Report CS-2006-34, David R. Cheriton School of Computer Science, University of Waterloo, 2006.

[All10] Alloy Website, `http://alloy.mit.edu/`, Feb. 2010.

[ALSU86] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ant10] Antlr Website, `http://www.antlr.org`, Feb. 2010.

[Asm10] AsmL Website, `http://www.research.microsoft.com/fse/asml`, Feb. 2010.

[BCD+89] Patrick Borras, Dominique Clément, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. Centaur: the system. *SIGPLAN Not.*, 24(2):14–24, 1989.

[BCD+06] Manfred Broy, Michelle L. Crane, Jürgen Dingel, Alan Hartman, Bernhard Rumpe, and Bran Selic. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Reports and Revised Selected Papers*, volume 4364 of *LNCS*, pages 318–323. Springer, 2006.

[BCGR08] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Modular Description of a Comprehensive Semantics Model for the UML (Version 2.0). Informatik-Bericht 2008-06, Technische Universität Braunschweig, 2008.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*. John Wiley & Sons, 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*. John Wiley & Sons, 2009.

[BCR04] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. On formalizing UML state machines using ASM. *Information & Software Technology*, 46(5):287–292, 2004.

[BCR06] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The Structural Data Model. Technical Report TUM-I0612, Institut für Informatik, Technische Universität München, 2006.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The Control Model. Technical Report TUM-I0710, Institut für Informatik, Technische Universität München, 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, Institut für Informatik, Technische Universität München, 2007.

[Bee94] Michael von der Beeck. A Comparison of Statecharts Variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (Proceedings)*, volume 863 of *LNCS*, pages 128–148. Springer, 1994.

[Bee02] Michael von der Beeck. A structured operational semantics for UML-statecharts. *Software and System Modeling*, 1(2):130–141, 2002.

[Ber07] Yves Bertot. Theorem proving support in programming language semantics. Technical Report 6242, INRIA Sophia Antipolis, 2007.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, O. Slotosch, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language Spectrum, An Informal Introduction (V 1.0), Part 1 & 2. Technical Report TUM-I9312, Technische Universität München, 1993.

[BGH+98] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and complete object interaction descriptions. *Comput. Stand. Interfaces*, 19(7):335–345, 1998.

[BHH$^+$97] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a Formalization of the Unified Modeling Language. In *European Conference on Object-Oriented Programming (ECOOP) 1997 (Proceedings)*, volume 1241 of *LNCS*, pages 344–366. Springer, 1997.

[BKRR09] Christian Berger, Holger Krahn, Holger Rendel, and Bernhard Rumpe. Featurebasierte Modellierung und Verarbeitung von Produktlinien am Beispiel eingebetteter Software. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme V*. Informatik-Bericht 2009-01, Technische Universität Braunschweig, 2009.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004.

[BS86] Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst.*, 8(4):547–576, 1986.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer, 2001.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[BSM$^+$03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[BW82] Manfred Broy and Martin Wirsing. Partial Abstract Types. *Acta Inf.*, 18(1):47–64, 1982.

[BW06] Achim D. Brucker and Burkhart Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[BW09] Achim D. Brucker and Burkhart Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4):255–284, 2009.

[CD05] Michelle L. Crane and Jürgen Dingel. On the semantics of UML state machines: Categorization and comparison. Technical Report 2005-501, School of Computing, Queen's University, 2005.

[CD07] Michelle L. Crane and Jürgen Dingel. UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and System Modeling*, 6(4):415–435, 2007.

[CD08a] Michelle L. Crane and Jürgen Dingel. Towards a Formal Account of a Foundational Subset for Executable UML Models. In *Model Driven Engineering Languages and Systems (MoDELS) 2008 (Proceedings)*, volume 5301 of *LNCS*, pages 675–689. Springer, 2008.

[CD08b] Michelle L. Crane and Jürgen Dingel. Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities. In *Centrer for Advanced Studies on Collaborative Research (CASCON) 2008 (Proceedings)*, pages 96–110. IBM, 2008.

[CDGR07] María Victoria Cengarle, Jürgen Dingel, Hans Grönniger, and Bernhard Rumpe. System-Model-Based Simulation of UML Models. In *Nordic Workshop on Model Driven Engineering (NW-MODE) 2007 (Proceedings)*, 2007.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CE06] Manuel Clavel and Marina Egea. ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams. In *Algebraic Methodology and Software Technology (AMAST) 2006 (Proceedings)*, volume 4019 of *LNCS*, pages 368–373. Springer, 2006.

[Cen07] María Victoria Cengarle. System model for UML – The interactions case. In *Methods for Modelling Software Systems (MMOSS) 2007 (Proceedings)*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[CG09] María Victoria Cengarle and Hans Grönniger. System Model Semantics of Interactions. Technical Report TUM-I0932, Institut für Informatik, Technische Universität München, 2009.

[CGR08a] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, Technische Universität Braunschweig, 2008.

[CGR08b] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Statecharts. Informatik-Bericht 2008-04, Technische Universität Braunschweig, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Model Driven Engineering Languages and Systems (MoDELS) 2009 (Proceedings)*, volume 5795 of *LNCS*, pages 670–684. Springer, 2009.

[CJ05] Franck Chauvel and Jean-Marc Jézéquel. Code Generation from UML Models with Semantic Variation Points. In *Model Driven Engineering Languages and Systems (MoDELS) 2005 (Proceedings)*, volume 3713 of *LNCS*, pages 54–68. Springer, 2005.

[CK04] María Victoria Cengarle and Alexander Knapp. UML 2.0 Interactions: Semantics and Refinement. In *Critical Systems Development with UML (CSDUML) 2004 (Proceedings)*. Technical Report TUM-I0415, Institut für Informatik, Technische Universität München, 2004.

[CKTW08] María Victoria Cengarle, Alexander Knapp, Andrzej Tarlecki, and Martin Wirsing. A Heterogeneous Approach to UML Semantics. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *LNCS*, pages 383–402. Springer, 2008.

[CKW$^+$99] Steve Cook, Anneke Kleppe, Jos Warmer, Richard Mitchell, Bernhard Rumpe, and Alan Cameron Wills. Defining UML Family Members Using Prefaces. In *Technology of Object-Oriented Languages and Systems (TOOLS Pacific) 1999 (Proceedings)*, pages 102–114. IEEE Computer Society, 1999.

[CMTG07a] Arnaud Cuccuru, Chokri Mraidha, François Terrier, and Sébastien Gérard. Enhancing UML Extensions with Operational Semantics. In *Model Driven Engineering Languages and Systems (MoDELS) 2007 (Proceedings)*, volume 4735 of *LNCS*, pages 271–285. Springer, 2007.

[CMTG07b] Arnaud Cuccuru, Chokri Mraidha, François Terrier, and Sébastien Gérard. Templatable Metamodels for Semantic Variation Points. In *Model Driven Architecture- Foundations and Applications (ECMDA-FA) 2007 (Proccedings)*, volume 4530 of *LNCS*, pages 68–82. Springer, 2007.

[Cra09] Michelle L. Crane. *Slicing UML's Three-layer Architecture: A Semantic Foundation for Behavioural Specification.* PhD thesis, Queen's University Kingston, Ontario, Canada, 2009.

[CSAJ05] Kai Chen, Janos Sztipanovits, Sherif Abdelwahed, and Ethan K. Jackson. Semantic Anchoring with Model Transformations. In *Model Driven Architecture - Foundations and Applications (ECMDA-FA) 2005 (Proceedings)*, volume 3748 of *LNCS*, pages 115–129. Springer, 2005.

[DD06] Zinovy Diskin and Jürgen Dingel. Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In *Model Driven Engineering Languages and Systems (MoDELS) 2006 (Proceedings)*, volume 4199 of *LNCS*, pages 230–244. Springer, 2006.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[DHC07] Haitao Dan, Robert M. Hierons, and Steve Counsell. A Thread-tag Based Semantics for Sequence Diagrams. In *Software Engineering and Formal Methods (SEFM) 2007 (Proceedings)*, pages 173–182. IEEE Computer Society, 2007.

[DHTT00] Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, and Michael Tyrsted. Supporting Several Levels of Restriction in the UML. In *The Unified Modeling Language, Advancing the Standard, Third International Conference (UML) 2000 (Proceedings)*, volume 1939 of *LNCS*, pages 396–409. Springer, 2000.

[Ecl10] Eclipse Website, `http://www.eclipse.org`, Feb. 2010.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In *The Unified Modeling Language, UML'98: Beyond the Notation, First International Workshop, Selected Papers*, volume 1618 of *LNCS*, pages 336–348. Springer, 1999.

[EJL$^+$03] J. Eker, J. W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[FM07] Christoph Ficek and Fabian May. Umsetzung der Java 5 Grammatik für MontiCore. Studienarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2007.

[Fow05] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, 2005. `http://martinfowler.com/articles/languageWorkbench.html`.

[FPR02] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[FRR09] Florian Fieber, Nikolaus Regnat, and Bernhard Rumpe. Assessing usability of model driven development in industrial projects. In *4th European Workshop "From code centric to model centric software engineering: Practices, Implications and ROI (C2M)" (Proceedings) 2009*, pages 1–9. University of Twente, Enschede: Centre for Telematics and Information Technology, 2009.

[FS06] Harald Fecher and Jens Schönborn. UML 2.0 State Machines: Complete Formal Semantics Via core state machine. In *Formal Methods: Applications and Technology (FMICS) 2006, Revised Selected Papers*, volume 4346 of *LNCS*, pages 244–260. Springer, 2006.

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *Formal Methods and Software Engineering, (ICFEM) 2005 (Proceedings)*, volume 3785 of *LNCS*, pages 52–65. Springer, 2005.

[GBR05] Martin Gogolla, Jörg Bohling, and Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Software and Systems Modeling*, 4(4):386–398, 2005.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKH09] Martin Gogolla, Mirco Kuhlmann, and Lars Hamann. Consistency, Independence and Consequences in UML and OCL Models. In *Tests and Proofs, Third International Conference (TAP) 2009 (Proceedings)*, volume 5668 of *LNCS*, pages 90–104. Springer, 2009.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Technical Report Informatik-Bericht 2006-04, Software Systems Engineering, Technische Universität, 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Text-based Modeling. In *Software Language Engineering (ateM) 2007 (Proceedings)*, 2007.

[GME10] GME Website, `http://www.isis.vanderbilt.edu/projects/gme`, Feb. 2010.

[GMF10] GMF Website, `http://www.eclipse.org/modeling/gmf`, Feb. 2010.

[GR03] Martin Große-Rhode. *Semantic Integration of Heterogeneous Software Specifications*. Monographs in Theoretical Computer Science. Springer, 2003.

[GR07] Borislav Gajanovic and Bernhard Rumpe. ALICE: An Advanced Logic for Interactive Component Engineering. In *4th International Verification Workshop in connection with CADE-21 (VERIFY) 2007 (Proceedings)*, volume 259 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[GR10] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Modeling, Development and Verification of Adaptive Systems, 16th Monterey Workshop 2010 (Proceedings)*. Microsoft Research, Redmond, 2010.

[GRe10] GReAT Website, `http://www.isis.vanderbilt.edu/tools/GReAT`, Feb. 2010.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-Based Definition of Modeling Language Semantics. In *Formal Techniques for Distributed Systems (FMOODS/FORTE) 2009 (Proceedings)*, volume 5522 of *LNCS*, pages 152–166. Springer, 2009.

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. *Specification and validation methods*, pages 9–36, 1995.

[HHRS05] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Stairs towards formal design with sequence diagrams. *Software and System Modeling*, 4(4):355–367, 2005.

[HKR$^+$07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Model Driven Architecture - Foundations and Applications, (ECMDA-FA) 2007 (Proceedings)*, number 4530 in LNCS, 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 Modelling. In *International Conference on Software Engineeering in Research and Practice (SERP), Vol. 1 (Proceedings) 2009*, pages 172–176. CSREA Press, 2009.

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[HTM10] HTML Spezifikation, `http://www.w3.org/html/wg/html5/`, Feb. 2010.

[Huß02] Heinrich Hußmann. Loose semantics for UML/OCL. In *Integrated Design and Process Technology (IDPT) 2002 (Proceedings)*, 2002.

[Int06] International Organization for Standardization. SQL database language, ISO/IEC 9075-14:2006, 2006.

[Isa10] Isabelle Website, `http://isabelle.in.tum.de`, Feb. 2010.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[Kah87] Gilles Kahn. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science (STACS) 1987*, volume 247 of *LNCS*, pages 22–39. Sringer, 1987.

[KGKZ09] Sabine Kuske, Martin Gogolla, Hans-Jörg Kreowski, and Paul Ziemann. Towards an integrated graph-based semantics for UML. *Software and System Modeling*, 8(3):403–422, 2009.

[KJ92] B. L. Kurtz and J. B. Johnston. Using the synthesizer-generator to teach principles of programming language semantics. In *Twenty-third SIGCSE technical symposium on Computer science education (Proceedings)*, pages 207–212. ACM, 1992.

[KKP$^+$09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *9th OOPSLA Workshop on Domain-Specific Modeling (DSM) 2009 (Proceedings)*, 2009.

[KM08] Pierre Kelsen and Qin Ma. A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In *Model Driven Engineering Languages and Systems (MoDELS) 2008 (Proceedings)*, volume 5301 of *LNCS*, pages 690–704. Springer, 2008.

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[Knu68] Donald F. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 12:127–145, 1968.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software Engineering*. Doktorarbeit, RWTH Aachen, 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model -. In *Formal Methods for Open Object-based Distributed Systems (FMOODS) 1996 (Proceedings)*, pages 323–338. ENST France Telecom, 1996.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *7th OOPSLA Workshop on Domain-Specific Modeling (Proceedings) 2007*, Technical Report TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Model Driven Engineering Languages and Systems (MoDELS) 2007 (Proceedings)*, volume 4735 of *LNCS*, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Objects, Components, Models and Patterns, TOOLS EUROPE 2008 (Proceedings)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer, 2008.

[KS97] Peter Klein and Andy Schürr. Constructing SDEs with the IPSEN meta environment. In *8th Conference on Software Engineering Environments (SEE) 1997 (Proceedings)*, pages 2–10. IEEE Computer Society, 1997.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[LMB$^+$01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The Generic Modeling Environment. In *International Workshop on Intelligent Signal Processing (WISP) 2001 (Proceedings)*. IEEE Computer Society, 2001.

[Loc08] Andreas Lochbihler. Type Safe Nondeterminism - A Formal Semantics of Java Threads. In *Foundations of Object-Oriented Languages (FOOL) 2008 (Proceedings)*. ACM, 2008.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[Mah06] Björn Mahler. Aspektgetriebene Modelltransformationen. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[Mat07] MathWorks Automotive Advisory Board (MAAB). Control Alogrithm Modeling Guidlines Using Matlab, Simulink, and Stateflow – Version 2.1, July 2007. `http://www.mathworks.com/automotive/standards/maab.html`.

[MB07] Azzam Maraee and Mira Balaban. Efficient Reasoning About Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. In *Model Driven Architecture - Foundations and Applications, (ECMDA-FA) 2007 (Proceedings)*, volume 4530 of *LNCS*, pages 17–31. Springer, 2007.

[MD08] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *Model Driven Architecture - Foundations and Applications, (ECMDA-FA) 2008, (Proceedings)*, volume 5095 of *LNCS*, pages 432–443. Springer, 2008.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[Mil93] Robin Milner. The Polyadic *pi*-Calculus: A Tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer, 1993.

[MIS10] MISRA C Website, `http://www.misra-c2.com/`, Feb. 2010.

[MOD06] MODELWARE. D5.3-1 Industrial ROI, Assessment, and Feedback - Master Document. Revision 2.2, 2006. `http://www.modelware-ist.org`.

[Mon10] MontiCore Website, `http://www.monticore.de`, Feb. 2010.

[Mos01] Peter D. Mosses. The Varieties of Programming Language Semantics. In *Perspectives of System Informatics (PSI) 2001 (Revised Papers)*, volume 2244 of *LNCS*, pages 165–190. Springer, 2001.

[NAD03] Jianwei Niu, Joanne M. Atlee, and Nancy A. Day. Template Semantics for Model-Based Notations. *IEEE Trans. Software Eng.*, 29(10):866–882, 2003.

[Nip98] Tobias Nipkow. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Nun09] Miguel P. Nunes. A Java Extension For Transformations. Masterarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2009.

[Ohe01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[O'K06] Greg O'Keefe. Improving the Definition of UML. In *Model Driven Engineering Languages and Systems (MoDELS) 2006 (Proceedings)*, volume 4199 of *LNCS*, pages 42–56. Springer, 2006.

[OMG06a] Object Management Group. Meta Object Facility Version 2.0 (2006-01-01), January 2006. `http://www.omg.org/spec/MOF/2.0`.

[OMG06b] Object Management Group. Object Constraint Language Version 2.0 (2006-05-01), May 2006. `http://www.omg.org/spec/OCL/2.0`.

[OMG08] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2008-04-03), April 2008. `http://www.omg.org/spec/QVT/1.0`.

[OMG09a] Object Management Group. Unified Modeling Language: Infrastructure Version 2.2 (2009-02-04), February 2009. `http://www.omg.org/spec/UML/2.2`.

[OMG09b] Object Management Group. Unified Modeling Language: Superstructure Version 2.2 (2009-02-02), February 2009. `http://www.omg.org/spec/UML/2.2`.

[Ope10] OpenArchitectureWare Website, Febuary 2010. `http://www.openarchitectureware.com/`.

[PADS08] Adam Prout, Joanne M. Atlee, Nancy A. Day, and Pourya Shaker. Semantically Configurable Code Generation. In *Model Driven Engineering Languages and Systems (MoDELS) 2008 (Proceedings)*, volume 5301 of *LNCS*, pages 705–720. Springer, 2008.

[PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[Pet62] C. A. Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *IFIP Congress*, pages 386–390, 1962.

[Pin07] Claas Pinkernell. Integration der Object Constraint Language in die UML/P. Studienarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2007.

[Plo81] Gordon D. Plotkin Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Rin08] Jan O. Ringert. Umsetzung und Anwendung des Systemmodells zur Semantikdefinition im Theorembeweiser Isabelle. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

[RK08] Arend Rensink and Anneke Kleppe. On a Graph-Based Semantics for UML Class and Object Diagrams. *Electronic Communications of the EASST*, 10, 2008.

[RMHV06] Damijan Rebernak, Marjan Mernik, Pedro Rangel Henriques, and Maria J. Varanda. AspectLISA: an aspect-oriented compiler construction system based on attribute grammars. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 164(2):37–53, Oct 2006.

[RT84] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *First ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (SDE 1) 1984 (Proceedings)*, pages 42–48. ACM, 1984.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, Technische Universität München, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML. A Vision or a Nightmare? In *Issues & Trends of Information Technology Management in Contemporary Associations*, pages 697 – 701. Idea Group Publishing, 2002.

[Rum04a] Bernhard Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer, 2004.

[Rum04b] Bernhard Rumpe. *Modellierung mit UML*. Springer, 2004.

[RW99] Gianna Reggio and Roel Wieringa. Thirty one problems in the semantics of UML 1.3 dynamics. In *OOPSLA 99: Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations 1999 (Proceedings)*, 1999.

[SASX05] Subash Shankar, Sinan Asa, Vladimir Sipos, and Xiaowei Xu. Reasoning about real-time statecharts in the presence of semantic variations. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2005 (Proceedings)*, pages 243–252. ACM, 2005.

[Sch91] Andy Schürr. PROGRES: A VHL-Language Based on Graph Grammars. In *4th International Workshop on Graph-Grammars and Their Application to Computer Science 1991 (Proceedings)*, volume 532 of *LNCS*, pages 641–659. Springer, 1991.

[Sch08] Martin Schindler. Semantikerhaltende Statechart-Transformationen mit MontiCore TF. Studienarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

[Sch09] Martin Schindler. UML/P Syntax with MontiCore. Interner Technischer Bericht, RWTH Aachen, 2009.

[SD02] Graeme Smith and John Derrick. Abstract Specification in Object-Z and CSP. In *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods (ICFEM) 2002 (Proceedings)*, volume 2495 of *LNCS*, pages 108–119. Springer, 2002.

[SDL$^+$08] Yu Sun, Zekai Demirezen, Tomaz Lukman, Marjan Mernik, and Jeff Gray. Model Transformations Require Formal Semantics. In *Domain-Specific Program Development (DSPD) 2008 (Proceedings)*, 2008.

[Sel04] Bran Selic. On the Semantic Foundations of Standard UML 2.0. In *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT) 2004 (Revised Lectures)*, volume 3185 of *LNCS*, pages 181–199. Springer, 2004.

[Sim10] Simulink Website `http://www.mathworks.com/products/simulink`, Feb. 2010.

[Smi00] Graeme Smith. *The Object-Z specification language*. Kluwer Academic Publishers, 2000.

[SS71] Dana Scott and Christopher Strachey. Toward A Mathematical Semantics for Computer Languages. In *Symposium on Computers and Automata (Proceedings)*, volume XXI, pages 19–46. Polytechnic Press, 1971.

[Szl06] Marcin Szlenk. Formal Semantics and Reasoning about UML Class Diagram. In *Dependability of Computer Systems (DepCoS-RELCOMEX) 2006 (Proceedings)*, pages 51–59. IEEE Computer Society, 2006.

[TA06] Ali Taleghani and Joanne M. Atlee. Semantic Variations Among UML State-Machines. In *Model Driven Engineering Languages and Systems (MoDELS) 2006 (Proceedings)*, volume 4199 of *LNCS*, pages 245–259. Springer, 2006.

[Tar72] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Ter95] Delphine Terrasse. Encoding Natural Semantics in Coq. In *Algebraic Methodology and Software Technology (AMAST) 1995 (Proceedings)*, volume 936 of *LNCS*, pages 230–244. Springer, 1995.

[The06] Jens Theeß. MDA Transformationstechniken. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999.

[Tor06] Christoph Torens. Konstruktion von Grammatiken aus exemplarischen Dokumenten. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[vdBvDH$^+$01] Mark van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Compiler Construction, 10th International Conference, (CC) 2001 (Proceedings)*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[Völ08] Markus Völter. A Family of Languages for Architecture Description. In *8th OOPSLA Workshop on Domain-Specific Modeling (DSM) 2008 (Proceedings)*, pages 86–93. University of Alabama at Birmingham, 2008.

[Wac07] Guido Wachsmuth. Modelling the Operational Semantics of Domain-Specific Modelling Languages. In *Generative and Transformational Techniques in Software Engineering II, International Summer School (GTTSE) 2007 (Revised Papers)*, volume 5235 of *LNCS*, pages 506–520. Springer, 2007.

[WGM08] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Softw., Pract. Exper.*, 38(10):1073–1103, 2008.

[Wil97] David S. Wile. Abstract syntax from concrete syntax. In *19th International Conference on Software Engineering (ICSE) 1997 (Proceedings)*, pages 472–480. ACM, 1997.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computer Science Series. MIT Press, 1993.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying Machine Code Safety: Shallow versus Deep Embedding. In *Theorem Proving in Higher Order Logics (TPHOLs) 2004 (Proceedings)*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.

[ZX04] Yingzhou Zhang and Baowen Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Not.*, 39(3):14–30, 2004.

Teil V

Anhänge

Anhang A

Mathematische Grundlagen

In diesem Kapitel fassen wir knapp die verwendeten grundlegenden, mathematischen Notationen zusammen.

A.1 Funktionen, Logik, Mengen

Definition A.1 (Funktionen)

Functions

$f : X \rightarrow Y, g : X \rightarrow Y \rightarrow Z$	totale Funktionen
$f : X \rightharpoonup Y$	partielle Funktionen
$f(a), g\ a\ b$	Funktionsanwendung (verschiedene Varianten)
$\mathrm{dom}(f) \subseteq X$	Definitionsbereich von f (interessant, wenn f partiell ist)
$[a = b]$	entspricht einer Funktion f mit $f(a) = b$
$[a = b, c = d, \dots]$	entspricht einer Funktion f mit $f(a) = b, f(c) = d$ usw.
$f \oplus g$	Vereinigung von Funktionen, g überschreibt f: $(f \oplus g)(a) = g(a)$, falls $a \in \mathrm{dom}(g)$, sonst $= f(a)$
$f\,\vert_S$	Einschränkung des Definitionsbereichs von f auf $\mathrm{dom}(f) \cap S$

Definition A.2 (Logik)

Logic

$P \wedge Q, P \vee Q, \neg P,$ $P \Rightarrow Q, P \Leftrightarrow Q$	logische Operatoren
$\mathrm{bool} = \{tt, ff\}$	Wahrheitswerte
$\exists x : P(x); \quad \exists^1, \exists^{\geq 1}, \exists^{\leq 1}$	Varianten des Existenzquantors
$\forall x : P(x)$	Allquantor
$P(a, *, *)$	wildcard $*$ ist Abkürzung für $\exists x, y : P(a, x, y)$

Definition A.3 (Mengen)

Sets

$\varnothing, \{1,2,3\}$	konkrete Mengen
$\wp(A), \wp_f(A)$	(endliche) Potenzmenge (Menge von Teilmengen von A)
$A \cup B, A \cap B, A \setminus B, \bigcup_{x \in \wp(X)}, \bigcap_{x \in \wp(X)}$	Operationen auf Mengen
$x \in A, B \subseteq A$	Enthaltensein und Teilmengenbeziehung
$\{x \mid P(x)\}, \{f(x) \mid P(x)\}$	Mengenkomprehension
$A_1 \times \ldots \times A_n$, $\pi_i(a_1, \ldots, a_i, \ldots, a_n) = a_i$	Kreuzprodukt und i-te Projektion
$\mathbb{N}, \mathbb{Z}$	natürliche und ganze Zahlen

A.2 Listen, Puffer, Stacks

Definition A.4 enthält eine Reihe von gemeinsamen Funktionen für Sammlungen von Elementen (*Collections*). $\mathbb{C}(X)$ bezeichnet eine Sammlung über einer Menge X. Solch eine Collection kann beispielsweise eine Liste, ein Puffer oder ein Stack sein.

Neben den allgemeinen Funktionen für eine Collection gibt es auch spezifische Funktionen. Beispielsweise benötigen wir die Funktionen *pop* und *push* für einen Stack, um das korrekte Verhalten als Stapelspeicher zu spezifizieren. Wir gehen auf solche Funktionen nicht gesondert ein. Verwendet werden folgende Arten von Collections:

- $\mathrm{List}(S), S^*$ als Menge der Listen über der Menge S
- $\mathrm{Stack}(S)$ als Menge der Stacks über S und
- $\mathrm{Buffer}(S)$ als Menge der Puffer über S.

Definition A.4 (Sammlungen von Elementen)

Collections

$. \in . : X \times \mathbb{C}(X) \to \mathbb{B}$	Enthaltensein
$\#. : \mathbb{C}(X) \to \mathbb{N} \cup \{\infty\}$	Länge
$set : \mathbb{C}(X) \to \wp(X)$	als Menge
$s[i]$	für den Zugriff aus das i-te Elemente von s
$c \mid_S$	Einschränkung von c auf die in der Menge S enthaltenen Elemente

Notation:
Konkrete Listen sind beispielsweise die leere Liste $[]$ oder eine Liste mit drei Elementen $[1,2,3]$.

A.3 Zustandsmaschinen

Definition A.5 (Zustandsmaschinen STS)

STS

$\mathrm{STS}(S,I,O) = \{S,I,O,\delta,s0) \mid$
$\quad \delta \in S \times I^* \rightarrow \wp(S \times O^*) \wedge$
$\quad s0 \subseteq S \wedge s0 \neq \varnothing \wedge$
$\quad \forall s \in S, i \in I^* :$
$\qquad \delta(s,i) \neq \varnothing \wedge$
$\qquad \forall (t,o) \in \delta(s,i), i' \in I^*, (t',o') \in \delta(s,i') : o' = o\}$

$\delta : s \xrightarrow{i/o} t$ ist Abkürzung für $(t,o) \in \delta(s,i)$

Zustandsmaschinen (State Transition Systems), die *input enabled* sind. Das bedeutet, dass für jeden Zustand und jede Eingabe eine Transition des Automaten definiert ist. Hinzukommt die Bedingung, dass die Ausgabe o in jeden Schritt nur vom Zustand s abhängt und nicht von der Eingabe. Effektiv betrachten wir Taktautomaten im Sinne von [Rum96].

Definition A.6 (Komposition von Zustandsmaschinen)

KompSTS

use STS

$\otimes : \mathrm{STS}(S_1,I_1,O_1) \times \mathrm{STS}(S_2,I_2,O_2) \rightarrow \mathrm{STS}(S,I,O)$

$\forall A_1 = (S_1,I_1,O_1,\delta_1,s0_1) \in \mathrm{STS}(S_1,I_1,O_1), A_2 = (S_2,I_2,O_2,\delta_2,s0_2) \in \mathrm{STS}(S_2,I_2,O_2) :$
$I = (I_1 \cup I_2) \backslash (O_1 \cup O_2) \wedge$
$O = (O_1 \cup O_2) \backslash (I_1 \cup I_2) \wedge$
$S = S_1 \times S_2 \wedge$
$s0 = s0_1 \times s0_2 \wedge$
$O_1 \cap O_2 = \varnothing \Rightarrow$
$A_1 \otimes A_2 = (S,I,O,\delta,s0) \in \mathrm{STS}(S,I,O) \wedge$
$\forall x \in I^*, (s_1,s_2) \in S :$
$\quad \delta((s_1,s_2),x) = \{((t_1,t_2), y\,|_O) \mid \quad y \in (I \cup O_1 \cup O_2)^* \wedge y\,|_I = x \wedge$
$\qquad (t_1, y\,|_{O_1}) \in \delta_1(s_1, y\,|_{I_1}) \wedge$
$\qquad (t_2, y\,|_{O_2}) \in \delta_2(s_2, y\,|_{I_2})\}$

Komposition von Automaten. Die Bedingung aus Definition A.5, dass die Ausgabe nur vom Zustand und nicht von der Eingabe abhängt, stellt sicher, dass es bei der Komposition nicht zu Kausalitätsproblemen kommt (die Ausgabe des einen Automaten schon von der Ausgabe des anderen Automaten im selben Ausführungsschritt bestimmt wird).

Anhang B

Isabelle-Theorien des Systemmodells

Dieser Anhang enthält alle in Isabelle kodierten Theorien des Systemmodells. Am Ende jedes Abschnitts sind auch die Varianten der jeweiligen Theorie angegeben.

B.1 Base

Definition SysMod B.1 (Base)

Base

Isabelle-Theorie

```
theory Base
imports Main
begin

typedecl SystemModel

types Name = "char list"

types iVAR = Name

datatype iCLASS = Class Name

datatype iTYPE =  TBool | TInt | TVoid
               | TO iCLASS
               | TI iCLASS
               | TChar | TString
               | TExt Name "iTYPE list"

datatype iOID = Oid Name | VNil

datatype iVAL =  VBool bool | VInt int | VVoid
              | VOid iOID   | VI iINSTANCE
              | VChar char | VString "char list"
              | VExt Name "iVAL list"
and
  iINSTANCE = I "iOID × (iVAR ⇀ iVAL)"

types iVarAssign = "iVAR ⇀ iVAL"

fun iOf :: "iINSTANCE ⇒ (iOID × iVarAssign)"
where
 "iOf (I m) = m"

fun toOid :: "iVAL ⇒ iOID"
where
 "toOid (VOid oid) = oid"

```

```
fun toInt :: "iVAL ⇒ int"
where
 "toInt (VInt i) = i"

fun toBool :: "iVAL ⇒ bool"
where
 "toBool (VBool x) = x"

fun toClass :: "iTYPE ⇒ iCLASS"
where
 "toClass (TO t) = t"

types ’a stack = "’a list"

fun topF :: "’a stack ⇒ ’a"
where
 "topF (x#xs) = x"

fun push :: "’a ⇒ ’a stack ⇒ ’a stack"
where
 "push a st = (a#st)"

fun pop :: "’a stack ⇒ (’a stack * ’a)"
where
 "pop (x#xs) = (xs,x)"

types ’a buffer = "’a list"

datatype iASSOC = Assoc Name

types iDataStore = "iOID ⇀ iINSTANCE"

datatype iTHREAD = Thread Name

datatype iMETH = Meth Name

typedecl iPC

types iFRAME = "iOID × Name × iVarAssign × iVarAssign × iPC × iOID"

datatype iOPN = Op Name

types iControlStore = "iOID ⇀ iTHREAD ⇀ iFRAME stack"

types iMessage = "iOID × Name × iVAL list × iOID × iTHREAD option"

types iMessageStore = "iOID ⇀ iMessage buffer"

types iSTATE = "iDataStore × iControlStore × iMessageStore"

types iOSTATE =
  "iINSTANCE × (iTHREAD ⇀ iFRAME stack) × iMessage buffer"

end
```

B.2 Type

Definition SysMod B.2 (Type)

Type

Isabelle-Theorie

```
theory Type
imports Type1 BoolIntVoid Variable
begin
end
```

Definition SysMod B.3 (Type1)

Type1

Isabelle-Theorie

```
theory Type1
imports Base
begin

consts
 UTYPE :: "SystemModel ⇒ iTYPE set"
 UVAL  :: "SystemModel ⇒ iVAL set"
 CAR :: "SystemModel ⇒ (iTYPE ⇒ iVAL set)"

fun pCARNotEmpty :: "SystemModel ⇒ bool"
where
 "pCARNotEmpty sm = (∀ u ∈ UTYPE sm . CAR sm u ≠ {})"

fun valid-Type1 :: "SystemModel ⇒ bool"
where
 "valid-Type1 sm = pCARNotEmpty sm"

end
```

Definition SysMod B.4 (BoolIntVoid)

BoolIntVoid

Isabelle-Theorie

```
theory BoolIntVoid
imports Type1
begin

fun pTBoolIntExist :: "SystemModel ⇒ bool"
where
 "pTBoolIntExist sm = (TBool ∈ UTYPE sm ∧ TInt ∈ UTYPE sm)"

fun pTVoidExist :: "SystemModel ⇒ bool"
where
 "pTVoidExist sm = (TVoid ∈ UTYPE sm)"

fun pVBoolIntExist :: "SystemModel ⇒ bool"
where
 "pVBoolIntExist sm = ({VBool True, VBool False} ⊆  UVAL sm ∧
                       {VInt n | n . True} ⊆ UVAL sm)"

fun pVVoidExist :: "SystemModel ⇒ bool"
where
 "pVVoidExist sm = (VVoid ∈ UVAL sm)"

```

```
fun pCARBoolInt :: "SystemModel ⇒ bool"
where
 "pCARBoolInt sm = (CAR sm TBool = {VBool True, VBool False} ∧
                    CAR sm TInt  = {VInt n | n . True})"

fun pCARVoid :: "SystemModel ⇒ bool"
where
 "pCARVoid sm = (CAR sm TVoid = {VVoid})"

fun valid-BoolIntVoid :: "SystemModel ⇒ bool"
where
 "valid-BoolIntVoid sm =
   (pTBoolIntExist sm ∧ pVBoolIntExist sm ∧ pCARBoolInt sm ∧
    pTVoidExist sm ∧ pVVoidExist sm ∧ pCARVoid sm)"

end
```

Definition SysMod B.5 (Variable)

Variable

Isabelle-Theorie

```
theory Variable
imports Type1
begin

consts
 UVAR :: "SystemModel ⇒ iVAR set"
 vtype :: "SystemModel ⇒ iVAR ⇒ iTYPE"

fun vname :: "iVAR ⇒ Name"
where
 "vname n = n"

fun pUVAR :: "SystemModel ⇒ bool"
where
 "pUVAR sm = (∀ var ∈ UVAR sm . vtype sm var ∈ UTYPE sm)"

fun pUVARUnique :: "SystemModel ⇒ bool"
where
 "pUVARUnique sm =
     (∀ var1 ∈ UVAR sm .
       ∀ var2 ∈ UVAR sm .
         vname var1 = vname var2 ⟶ var1 = var2)"

fun VarAssign :: "SystemModel ⇒ iVarAssign set"
where
 "VarAssign sm = {vassign | vassign .
      dom vassign ⊆ UVAR sm ∧
      (∀ var ∈ dom vassign .
           the (vassign(var)) ∈ CAR sm (vtype sm var)) }"

fun asParameters :: "iVarAssign ⇒ Name list ⇒ iVAL list"
where
  "asParameters argassign [] = []"
| "asParameters argassign (n#xs) =
     (the(argassign(n)))#(asParameters argassign xs)"

fun constrVarAssign :: "iVAL list ⇒ Name list ⇒ iVarAssign"
where
  "constrVarAssign [] [] = empty"
| "constrVarAssign (v#vs) (n#ns) =
          (constrVarAssign vs ns)(n↦v)"

fun valid-Variable :: "SystemModel ⇒ bool"
where
 "valid-Variable sm =
    (pUVAR sm ∧ pUVARUnique sm)"

end
```

Variante SysMod B.6 (TypeOf)

TypeOf

Isabelle-Theorie

```
theory TypeOf
imports "$SYSTEMMODEL/Type1"
begin

consts
 typeOf :: "SystemModel ⇒ (iVAL ⇀ iTYPE)"

fun valid-TypeOf :: "SystemModel ⇒ bool"
where
 "valid-TypeOf sm =
    (∀ v ∈ UVAL sm .
      v ∈ dom (typeOf sm) ⟶ v ∈ CAR sm (the (typeOf sm v)))"

end
```

Variante SysMod B.7 (CharAndString)

CharAndString

Isabelle-Theorie

```
theory CharAndString
imports "$SYSTEMMODEL/Type1"
begin

fun pTCharStringExist :: "SystemModel ⇒ bool"
where
 "pTCharStringExist sm =
    (TChar ∈ UTYPE sm ∧ TString ∈ UTYPE sm)"

fun pVCharStringExist :: "SystemModel ⇒ bool"
where
 "pVCharStringExist sm =
    ( {VString s | s . True} ⊆ UVAL sm ∧
      {VChar c | c . True}   ⊆ UVAL sm )"

fun pCARCharString :: "SystemModel ⇒ bool"
where
 "pCARCharString sm = (
    CAR sm TChar = {VChar c | c . True} ∧
    CAR sm TString = {VString s | s . True})"

fun valid-CharAndString :: "SystemModel ⇒ bool"
where
 "valid-CharAndString sm = (
    pTCharStringExist sm ∧
    pVCharStringExist sm ∧
    pCARCharString sm      )"
```

```

end
```

B.3 Object

Definition SysMod B.8 (Object)

Object

Isabelle-Theorie

```
theory Object
imports Class Attribute Nil Subclassing
begin
end
```

Definition SysMod B.9 (Class)

Class

Isabelle-Theorie

```
theory Class
imports Type
begin

consts
 UCLASS :: "SystemModel ⇒ iCLASS set"
 UOID :: "SystemModel ⇒ iOID set"
 attr :: "SystemModel ⇒ iCLASS ⇒ iVAR set"
 oids :: "SystemModel ⇒ iCLASS ⇒ iOID set"
 classOf :: "SystemModel ⇒ iOID ⇒ iCLASS"

fun objects :: "SystemModel ⇒ iOID ⇒ iINSTANCE set"
where
 "objects sm oid = { I (oid, r) | r .
       dom r = attr sm (classOf sm oid) ∧
       r ∈ VarAssign sm ∧ oid ∈ UOID sm}"

fun objectsOfClass :: "SystemModel ⇒ iCLASS ⇒ iINSTANCE set"
where
 "objectsOfClass sm C = { inst | inst .
        ∃ oid ∈ oids sm C . inst ∈ objects sm oid ∧ C ∈ UCLASS sm}"

fun INSTANCE :: "SystemModel ⇒ iINSTANCE set"
where
 "INSTANCE sm = { inst | inst .
           ∃ C ∈ UCLASS sm . inst ∈ objectsOfClass sm C }"

fun pClassOf :: "SystemModel ⇒ bool"
where
 "pClassOf sm =
   (∀ C ∈ UCLASS sm . (∀ oid ∈ oids sm C . classOf sm oid = C))"

fun pOids :: "SystemModel ⇒ bool"
where
 "pOids sm =
    (( ∀ C ∈ UCLASS sm . (∀ oid ∈ oids sm C . oid ∈ UOID sm)) ∧
     (∀ oid ∈ UOID sm . classOf sm oid ∈ UCLASS sm))"

fun isSingleton :: "SystemModel ⇒ iCLASS ⇒ bool"
```

```
where
 "isSingleton sm C = (card (oids sm C) = 1)"

constdefs "valid-Class sm == pOids sm ∧ pClassOf sm"

end
```

Definition SysMod B.10 (Attribute)

Attribute

Isabelle-Theorie

```
theory Attribute
imports Class Type
begin

fun this :: "iINSTANCE ⇒ iOID"
where
 "this (I (oid, r)) = oid"

fun getAttr :: "iINSTANCE × iVAR ⇀ iVAL"
where
 "getAttr (I(oid, r),var) = r var"

fun attrOid :: "SystemModel ⇒ iOID ⇒ iVAR set"
where
 "attrOid sm oid = attr sm (classOf sm oid)"

end
```

Definition SysMod B.11 (Nil)

Nil

Isabelle-Theorie

```
theory Nil
imports Class Type Attribute
begin

fun pNil :: "SystemModel ⇒ bool"
where
 "pNil sm = (VNil ∈ UOID sm)"

defs uoid-def :
 "UOID sm == { oid | oid .
                oid = VNil ∨
                (∃ C ∈ UCLASS sm . oid ∈ oids sm C) }"

fun pNilOids :: "SystemModel ⇒ bool"
where
 "pNilOids sm = (∀ C ∈ UCLASS sm . VNil ∉ oids sm C)"

fun pThisNotNil :: "SystemModel ⇒ bool"
where
 "pThisNotNil sm = (∀ inst ∈ INSTANCE sm . this inst ≠ VNil)"

fun valid-Nil :: "SystemModel ⇒ bool"
where
 "valid-Nil sm = (pNil sm ∧ pNilOids sm ∧ pThisNotNil sm)"

```

```
end
```

Definition SysMod B.12 (Subclassing)

Subclassing

Isabelle-Theorie

```
theory Subclassing
imports Class Nil Type
begin

consts
 sub :: "SystemModel ⇒ iCLASS ⇒ iCLASS ⇒ bool"

fun pClassIsType :: "SystemModel ⇒ bool"
where
 "pClassIsType sm = (
    ∀ C ∈ UCLASS sm . TO C ∈ UTYPE sm  (* TO entspricht .& *)
  )"

fun pOIDIsValue :: "SystemModel ⇒ bool"
where
 "pOIDIsValue sm = (∀ oid ∈ UOID sm . VOid oid ∈ UVAL sm)"

fun psubRefl :: "SystemModel ⇒ bool"
where
 "psubRefl sm = (∀ C ∈ UCLASS sm . sub sm C C)"

constdefs pSubTrans :: "SystemModel ⇒ bool"
"pSubTrans sm ==
  (∀ C1 C2 C3 . sub sm C1 C2 ∧ sub sm C2 C3 ⟶ sub sm C1 C3)"

fun psubCAR :: "SystemModel ⇒ bool"
where
 "psubCAR sm = ( ∀ C ∈ UCLASS sm . CAR sm (TO C) =
                  { VOid oid | oid .
                     oid = VNil ∨
                     (∃ C1 ∈ UCLASS sm .
                         sub sm C1 C ∧  oid ∈ oids sm C1) })"

fun valid-Subclassing :: "SystemModel ⇒ bool"
where
 "valid-Subclassing sm =
    (pClassIsType sm ∧ pOIDIsValue sm ∧
     psubRefl sm ∧ pSubTrans sm ∧ psubCAR sm)"

end
```

Variante SysMod B.13 (LiskovPrinciple)

LiskovPrinciple

Isabelle-Theorie

```
theory LiskovPrinciple
imports "$SYSTEMMODEL/Subclassing"
        "$SYSTEMMODEL/Type"
begin

fun valid-LiskovPrinciple :: "SystemModel ⇒ bool"
where
 "valid-LiskovPrinciple sm = (
    (∀ C1 ∈ UCLASS sm .
      ∀ C2 ∈ UCLASS sm .
        (sub sm C1 C2 ⟶ attr sm C2 ⊆ attr sm C1)))"

end
```

Variante SysMod B.14 (AntiSymSub)

AntiSymSub

Isabelle-Theorie

```
theory AntiSymSub
imports "$SYSTEMMODEL/Subclassing"
        "$SYSTEMMODEL/Type"
begin

constdefs valid-AntiSymSub :: "SystemModel ⇒ bool"
 "valid-AntiSymSub sm ==
    (∀ C1 C2 . sub sm C2 C1 ∧  sub sm C1 C2  ⟶ C1 = C2)"

end
```

Variante SysMod B.15 (ValueObjects)

ValueObjects

Isabelle-Theorie

```
theory ValueObjects
imports "$SYSTEMMODEL/Subclassing"
        "$SYSTEMMODEL/Type"
begin

fun pClassIsType2 :: "SystemModel ⇒ bool"
where
 "pClassIsType2 sm = (
    ∀ C ∈ UCLASS sm . TI C ∈ UTYPE sm  (* TI entspricht .* *)
  )"

fun pInstIsValue :: "SystemModel ⇒ bool"
where
 "pInstIsValue sm = (∀ inst ∈ INSTANCE sm . VI inst ∈ UVAL sm)"

fun pCARValueObj :: "SystemModel ⇒ bool"
where
 "pCARValueObj sm =
       ( ∀ C ∈ UCLASS sm . CAR sm (TI C) =
              {VI inst | inst . inst ∈ objectsOfClass sm C})"

fun valid-ValueObjects :: "SystemModel ⇒ bool"
where
 "valid-ValueObjects sm =
    (pClassIsType2 sm ∧ pInstIsValue sm ∧ pCARValueObj sm)"

end
```

Variante SysMod B.16 (StrictSingleInheritance)

StrictSingleInheritance

Isabelle-Theorie

```
theory StrictSingleInheritance
imports "$SYSTEMMODEL/Subclassing"
        "$SYSTEMMODEL/Type"
begin

fun valid-StrictSingleInheritance :: "SystemModel ⇒ bool"
where
 "valid-StrictSingleInheritance sm = (∀ C1 C2 C3.
           sub sm C1 C2 ∧ sub sm C1 C3 ⟶ sub sm C2 C3 ∨ sub sm C3 C2)"
end
```

B.4 Data

Definition SysMod B.17 (Data)

Data

Isabelle-Theorie

```
theory Data
imports DataStore1 DataStore Association
begin
end
```

Definition SysMod B.18 (DataStore1)

DataStore1

Isabelle-Theorie

```
theory DataStore1
imports "$SYSTEMMODEL/Object"
begin

fun DataStore :: "SystemModel ⇒ iDataStore set"
where
 "DataStore sm = { ds | ds.
     dom ds ⊆ UOID sm ∧
     (∀ oid ∈ dom ds .
        ds oid ≠ None ⟶ this (the (ds oid)) = oid ∧
                       (the (ds oid)) ∈ objects sm oid)
  }"

fun oids :: "iDataStore ⇒ iOID set"
where
 "oids ds = dom ds"

end
```

Definition SysMod B.19 (DataStore)

DataStore

Isabelle-Theorie

```
theory DataStore
imports Object DataStore1
begin
```

```

fun val :: "SystemModel ⇒ iDataStore × iOID × iVAR ⇀ iVAL"
where
 "val sm (ds, oid, var) = (getAttr ( (the (ds oid)), var))"

fun addobj :: "SystemModel ⇒ iDataStore × iINSTANCE ⇒ iDataStore"
where
 "addobj sm (ds, inst) = (ds( (this inst) ↦ inst))"

fun setval :: "SystemModel ⇒
          iDataStore × iOID × iVAR × iVAL ⇀ iDataStore"
where
 "setval sm (ds, oid, var, v) =
       (Some (ds(oid ↦
               I (oid,
                  (snd (iOf (the (ds oid))))(var ↦ v))))
       )"

end
```

Definition SysMod B.20 (Association)

Association

Isabelle-Theorie

```
theory Association
imports DataStore
begin

consts
 UASSOC :: "SystemModel ⇒ iASSOC set"
 classesOf :: "SystemModel ⇒ iASSOC ⇒ iCLASS list"
 extraVals :: "SystemModel ⇒ iASSOC ⇒ iVAL set"
 relOf :: "SystemModel ⇒ iASSOC × iDataStore ⇒ (iOID list × iVAL) set"

fun pAssoc :: "SystemModel ⇒ bool"
where
 "pAssoc sm = (
    ∀ assoc ∈ UASSOC sm .
      (∀ C ∈ set (classesOf sm assoc) . C ∈ UCLASS sm) ∧
      extraVals sm assoc ⊆ UVAL sm
  )"

fun pAssocRelOf :: "SystemModel ⇒ bool"
where
 "pAssocRelOf sm = (
   ∀ R ∈ UASSOC sm . ∀ ds ∈ DataStore sm .
      (∀ link ∈ (relOf sm (R,ds)) .
        length (fst link) = length (classesOf sm R) ∧
        (∀ n . VOid (nth (fst link) n) ∈
                    CAR sm (TO (nth (classesOf sm R) n))) ∧
        snd link ∈ extraVals sm R
      )
  )"

fun valid-Association :: "SystemModel ⇒ bool"
where
 "valid-Association sm =
    (pAssoc sm ∧ pAssocRelOf sm)"

```

```
end
```

Variante SysMod B.21 (FiniteObjects)

FiniteObjects

Isabelle-Theorie

```
theory FiniteObjects
imports "$SYSTEMMODEL/DataStore1"
begin

fun valid-FiniteObjects :: "SystemModel ⇒ bool"
where
 "valid-FiniteObjects sm =
   (∀ ds ∈ DataStore sm . finite(oids(ds)))"

end
```

Variante SysMod B.22 (StaticAttr)

StaticAttr

Isabelle-Theorie

```
theory StaticAttr
imports "$SYSTEMMODEL/Data"
begin

consts
 StaticC :: "SystemModel ⇒ iCLASS"
 StaticOid :: "SystemModel ⇒ iOID"
 StaticAttr :: "SystemModel ⇒ iVAR set"

fun valid-StaticAttr :: "SystemModel ⇒ bool"
where
 "valid-StaticAttr sm = (
     StaticC sm ∈ UCLASS sm ∧
     StaticOid sm ∈ UOID sm ∧
     StaticAttr sm ⊆ UVAR sm ∧
     Class.oids sm (StaticC sm) = {StaticOid sm} ∧
     (∀ ds ∈ DataStore sm . StaticOid sm ∈ DataStore1.oids ds) ∧
     Class.attr sm (StaticC sm) = StaticAttr sm)"

end
```

Variante SysMod B.23 (BinaryAssociation)

BinaryAssociation

Isabelle-Theorie

```
theory BinaryAssociation
imports "$SYSTEMMODEL/Association"
begin

consts
 binaryAssoc :: "SystemModel ⇒ iASSOC ⇒ bool"
 binaryRelOf ::
       "SystemModel ⇒ iASSOC × iDataStore ⇒ (iOID × iOID) set"
 binaryAssocMultis ::
       "SystemModel ⇒ iASSOC ⇒ nat set ⇒ nat set ⇒ bool"

```

```
fun pBinaryAssoc :: "SystemModel ⇒ bool"
where
 "pBinaryAssoc sm = (
    ∀ R ∈ UASSOC sm .  ∀ ds ∈ DataStore sm .
     (binaryAssoc sm R  ⟶
       length (classesOf sm R) = 2 ∧
       card (extraVals sm R) = 1 ∧
       binaryRelOf sm (R,ds) = { (oid1,oid2) | oid1 oid2 . ∃ v .
                                 ([oid1,oid2],v) ∈ relOf sm (R,ds) }
     )
 )"

fun sources :: "SystemModel ⇒ iASSOC × iDataStore × iOID ⇒ iOID set"
where
 "sources sm (R,ds, oid2) =
       { oid1 | oid1 . (oid1, oid2) ∈ binaryRelOf sm (R, ds) }"

fun destinations ::
  "SystemModel ⇒ iASSOC × iDataStore × iOID ⇒ iOID set"
where
 "destinations sm (R, ds, oid1) =
       { oid2 | oid2 . (oid1, oid2) ∈ binaryRelOf sm (R, ds) }"

fun pBinaryAssocMultis :: "SystemModel ⇒ bool"
where
 "pBinaryAssocMultis sm = (
    ∀ R ∈ UASSOC sm . ∀ N1 N2 .
      (binaryAssocMultis sm R N1 N2 ⟶
        binaryAssoc sm R ∧
        (let C1 = (classesOf sm R)!0 in
         let C2 = (classesOf sm R)!1 in
         (∀ v1 ∈ CAR sm (TO C1) . ∀ v2 ∈ CAR sm (TO C2) . ∀ ds .
         card (destinations sm (R, ds, toOid v1)) ∈ N1 ∧
         card (sources sm (R, ds, toOid v2)) ∈ N2))))"

fun valid-BinaryAssociation :: "SystemModel ⇒ bool"
where
 "valid-BinaryAssociation sm =
    (pBinaryAssoc sm ∧ pBinaryAssocMultis sm)"

end
```

Variante SysMod B.24 (AttributeAssociation)

AttributeAssociation

Isabelle-Theorie

```
theory AttributeAssociation
imports "$SYSTEMMODEL/Association"
        BinaryAssociation
        "$SYSTEMMODEL/vObject/LiskovPrinciple"
begin

consts
 attributeAssoc :: "SystemModel ⇒ iASSOC ⇒ iVAR ⇒ bool"

fun valid-AttributeAssociation :: "SystemModel ⇒ bool"
where
 "valid-AttributeAssociation sm = (
    ∀ R ∈ UASSOC sm . ∀ at ∈ UVAR sm . ∀ ds ∈ DataStore sm .
      (attributeAssoc sm R  at ⟶
          binaryAssoc sm R  ∧
      (let C1 = (classesOf sm R)!0 in
        let C2 = (classesOf sm R)!1 in
          at ∈ attr sm C1 ∧ vtype sm at = TO C2 ∧
          binaryRelOf sm (R,ds) =
            { (o1, toOid (the (val sm (ds, o1, at)))) | o1 .
               VOid o1 ∈ CAR sm (TO C1) ∧ o1 ∈  oids ds
            }
      )
      )
  )"

end
```

Variante SysMod B.25 (MtoMAssociation)

MtoMAssociation

Isabelle-Theorie

```
theory MtoMAssociation
imports "$SYSTEMMODEL/Association"
        BinaryAssociation
begin

consts
 mtomAssoc :: "SystemModel ⇒ iASSOC ⇒ (iCLASS × iVAR × iVAR) ⇒ bool"

fun pmtomAssoc :: "SystemModel ⇒ bool"
where
 "pmtomAssoc sm = (
   ∀ R ∈ UASSOC sm .
     ∀ M ∈ UCLASS sm . ∀ at1 ∈ UVAR sm . ∀ at2 ∈ UVAR sm .
      ∀ ds ∈ DataStore sm .
       (mtomAssoc sm R  (M, at1, at2) ⟶
        (let C1 = (classesOf sm R)!0 in
         let C2 = (classesOf sm R)!1 in
          binaryAssoc sm R ∧
          at1 ∈ attr sm M ∧ at2 ∈ attr sm M ∧
          vtype sm at1 = TO C1 ∧ vtype sm at2 = TO C2 ∧
           binaryRelOf sm (R,ds) =
             { (toOid (the (val sm (ds, m, at1))),
               toOid (the (val sm (ds, m, at2)))) | m .
                VOid m ∈ CAR sm (TO M) ∧ m ∈  oids ds
            }
       ))
 )"

end
```

Variante SysMod B.26 (CompositeAssociation)

CompositeAssociation

Isabelle-Theorie

```
theory CompositeAssociation
imports "$SYSTEMMODEL/Association"
        BinaryAssociation
begin

consts compositeAssoc :: "SystemModel ⇒ iASSOC ⇒ bool"

fun valid-CompositeAssociation :: "SystemModel ⇒ bool"
```

```
where
 "valid-CompositeAssociation sm  = (
   ∀ R ∈ UASSOC sm .
    (compositeAssoc sm R ⟶
      (∀ ds1 ds2 . ∀ oid1 oid2 .
       (oid1,oid2) ∈ binaryRelOf sm (R, ds1) ∧ oid1 ∉ oids ds2 ⟶
         oid2 ∉ oids ds2)
    )
 )"

end
```

Variante SysMod B.27 (QualifiedAssociation)

QualifiedAssociation

Isabelle-Theorie

```
theory QualifiedAssociation
imports "$SYSTEMMODEL/Association"
        BinaryAssociation
begin

consts
 qualifiedRelOf ::
   "SystemModel ⇒ iASSOC × iDataStore ⇒ (iOID × iVAL × iOID) set"
 leftQualifiedAssoc ::
    "SystemModel ⇒ iASSOC ⇒ (iVAL set) ⇒ bool"
 rightQualifiedAssoc ::
    "SystemModel ⇒ iASSOC ⇒ (iVAL set) ⇒ bool"

fun valid-QualifiedAssociation :: "SystemModel ⇒ bool"
where
 "valid-QualifiedAssociation sm  = (
  ∀ R ∈ UASSOC sm . ∀ values .
   binaryAssoc sm R ∧
  (let C1 = (classesOf sm R)!0 in
   let C2 = (classesOf sm R)!1 in
    (leftQualifiedAssoc sm R values ⟶
     (∀ ds .
        ∀ oid1 . VOid oid1 ∈ CAR sm (TO C2) ∧
        (∀ oid2 . VOid oid2 ∈ CAR sm (TO C1) ∧
         (oid1, oid2) ∈ binaryRelOf sm (R,ds) ⟶
           (∃ v ∈ values .
           (oid1, v, oid2) ∈ qualifiedRelOf sm (R,ds))

        ))) ∧
    (rightQualifiedAssoc sm R values ⟶
    (∀ ds .
      ∀ oid1 .
       VOid oid1 ∈ CAR sm (TO C1) ∧
       (∀ oid2 . VOid oid2 ∈ CAR sm (TO C2)  ∧
           (oid1, oid2) ∈ binaryRelOf sm (R,ds) ⟶
         (∃ v ∈ values .
           (oid1, v, oid2) ∈ qualifiedRelOf sm (R,ds))
        ))))
  )"

end
```

Variante SysMod B.28 (DerivedAssociation)

DerivedAssociation

Isabelle-Theorie

```
theory DerivedAssociation
imports "$SYSTEMMODEL/Data"
        "$SYSTEMMODEL/vData/BinaryAssociation"
begin

consts
 derivedAssoc :: "SystemModel ⇒ iASSOC ⇒
         (iASSOC × iDataStore ⇒ (iOID list × iVAL) set) ⇒ bool"

fun valid-DerivedAssociation :: "SystemModel ⇒ bool"
where
 "valid-DerivedAssociation sm = (
    ∀ R ∈ UASSOC sm . ∀ f .
     ∀ ds ∈ DataStore sm .
     ((derivedAssoc sm R f) ⟶
       relOf sm (R,ds) = f (R,ds)
     )
 )"

end
```

B.5 Control

Definition SysMod B.29 (Control)

Control

Isabelle-Theorie

```
theory Control
imports Operation Method1 Method StackFrame
        Thread ControlStore
begin
end
```

Definition SysMod B.30 (Operation)

Operation

Isabelle-Theorie

```
theory Operation
imports Object
begin

consts
 UOPN :: "SystemModel ⇒ iOPN set"
 classOf :: "SystemModel ⇒ iOPN ⇒ iCLASS"
 parTypes :: "SystemModel ⇒ iOPN ⇒ iTYPE list"
 resType :: "SystemModel ⇒ iOPN ⇒ iTYPE"

fun params :: "SystemModel ⇒ iOPN ⇒ (iVAL list) set"
where
 "params sm opn = { pars | pars .
      ∀ (i::nat) < (length (parTypes sm opn))  .
         pars!i ∈ (CAR sm ((parTypes sm opn)!i))
     }"

```

```
fun nameOf :: "iOPN ⇒ Name"
where
 "nameOf (Op n) = n"

fun pClassOf :: "SystemModel ⇒ bool"
where
 "pClassOf sm = (∀ opn ∈ UOPN sm . (classOf sm opn) ∈ UCLASS sm)"

fun pParTypes :: "SystemModel ⇒ bool"
where
 "pParTypes sm = (∀ opn ∈ UOPN sm . (set (parTypes sm opn)) ⊆ UTYPE sm)"

fun pResType :: "SystemModel ⇒ bool"
where
 "pResType sm = (∀ opn ∈ UOPN sm . (resType sm opn) ∈ UTYPE sm)"

fun valid-Operation :: "SystemModel ⇒ bool"
where
 "valid-Operation sm = (pResType sm ∧ pParTypes sm ∧ pClassOf sm)"

end
```

Definition SysMod B.31 (Method1)

Method1

Isabelle-Theorie

```
theory Method1
imports Operation
begin

consts
 UMETH :: "SystemModel ⇒ iMETH set"
 UPC :: "SystemModel ⇒ iPC set"
 definedIn :: "SystemModel ⇒ iMETH ⇒ iCLASS"
 parNames :: "SystemModel ⇒ iMETH ⇒ iVAR list"
 localNames :: "SystemModel ⇒ iMETH ⇒ iVAR list"
 resType :: "SystemModel ⇒ iMETH ⇒ iTYPE"
 pcOf :: "SystemModel ⇒ iMETH ⇒ iPC set"

fun nameOf :: "iMETH ⇒ Name"
where
 "nameOf (Meth n) = n"

fun parOf :: "SystemModel ⇒ iMETH ⇒ iVarAssign set"
where
 "parOf sm m = { vassign | vassign .
       vassign ∈ VarAssign sm ∧
       dom vassign = set (parNames sm m) }"

fun localsOf :: "SystemModel ⇒ iMETH ⇒ iVarAssign set"
where
 "localsOf sm m = { vassign | vassign .
       vassign ∈ VarAssign sm ∧
       dom vassign = set (localNames sm m) }"

fun pLocalVarNamesDisjoint :: "SystemModel ⇒ bool"
where
 "pLocalVarNamesDisjoint sm =
    (∀ m ∈ UMETH sm .
      set (parNames sm m) ∩ set (localNames sm m) = {})"

```

```
fun pAttrVarNamesDisjoint :: "SystemModel ⇒ bool"
where
 "pAttrVarNamesDisjoint sm =
    (∀ m ∈ UMETH sm .
      set (parNames sm m) ∩ attr sm (definedIn sm m) = {})"

fun pLocalAttrDisjoint :: "SystemModel ⇒ bool"
where
 "pLocalAttrDisjoint sm =
    (∀ m ∈ UMETH sm .
      set (localNames sm m) ∩ attr sm (definedIn sm m) = {})"

fun pdefinedIn :: "SystemModel ⇒ bool"
where
 "pdefinedIn sm = (
    ∀ m ∈ UMETH sm . definedIn sm m ∈ UCLASS sm
 )"

fun pparNames :: "SystemModel ⇒ bool"
where
 "pparNames sm = (
    ∀ m ∈ UMETH sm . set (parNames sm m) ⊆ UVAR sm
 )"

fun plocalNames :: "SystemModel ⇒ bool"
where
 "plocalNames sm = (
   ∀ m ∈ UMETH sm . set (localNames sm m) ⊆ UVAR sm
 )"

fun presType :: "SystemModel ⇒ bool"
where
 "presType sm = (
   ∀ m ∈ UMETH sm . resType sm m ∈  UTYPE sm
 )"

fun ppcOf :: "SystemModel ⇒ bool"
where
 "ppcOf sm = (
    ∀ m ∈ UMETH sm . pcOf sm m ⊆ UPC sm
 )"

fun valid-Method1 :: "SystemModel ⇒ bool"
where
 "valid-Method1 sm =
   (pLocalVarNamesDisjoint sm ∧ pAttrVarNamesDisjoint sm ∧
    pLocalAttrDisjoint sm ∧  pdefinedIn  sm ∧
    pparNames sm ∧ plocalNames sm ∧ presType sm ∧ ppcOf sm)"

end
```

Definition SysMod B.32 (Method)

Method

Isabelle-Theorie

```
theory Method
imports Method1
begin

consts
 impl :: "SystemModel ⇒ (iOPN ⇀ iMETH)"

fun conv :: "iVAR list ⇒ iVarAssign set ⇒ iVAL list set"
where
 "conv names vassigns = { l | l.
       ∃ vassign ∈ vassigns .
        ∀ i < length names .
         l!i = the (vassign (names!i))
   }"

fun pImpl :: "SystemModel ⇒ bool"
where
 "pImpl sm = (
    ∀ opn ∈ UOPN sm .
      ( ∀ m ∈ UMETH sm . impl sm opn = (Some m) ⟶
          Method1.nameOf m = Operation.nameOf opn ∧
          (sub sm (classOf sm opn) (definedIn sm m)) ∧
          (CAR sm (Method1.resType sm m) ⊆
                  CAR sm (Operation.resType sm opn)) ∧
          params sm opn ⊆ (conv (parNames sm m) (parOf sm m))
      )
 )"

fun pImpl2 :: "SystemModel ⇒ bool"
where
 "pImpl2 sm = (
    ∀ opn ∈ UOPN sm .
     (∃ C ∈ UCLASS sm . classOf sm opn = C ∧ oids sm C ≠ {})
       ⟶ opn ∈ dom (impl sm)
 )"

fun phImpl :: "SystemModel ⇒ bool"
where
 "phImpl sm = (
   ∀ opn ∈ UOPN sm .
    impl sm opn ≠ None ⟶ the (impl sm opn) ∈ UMETH sm
 )"

fun isInterface :: "SystemModel ⇒ iCLASS ⇒ bool"
where
 "isInterface sm C = (
    oids sm C = {} ∧
    (∀ opn ∈ UOPN sm . classOf sm opn = C ⟶  impl sm opn = None)

 )"

fun valid-Method :: "SystemModel ⇒ bool"
where
 "valid-Method sm =
   (phImpl sm ∧ pImpl sm ∧ pImpl2 sm)"

end
```

Definition SysMod B.33 (StackFrame)

StackFrame

Isabelle-Theorie

```
theory StackFrame
imports Method
begin

fun framesOf :: "SystemModel ⇒ iMETH ⇒ iFRAME set"
where
 "framesOf sm m = {
    (callee, name, pars, locals, pc, caller) |
     callee name pars locals pc caller .
      ∃ opn ∈ UOPN sm .
        m = the (impl sm opn) ∧
        name = nameOf m ∧
        callee ∈ oids sm (classOf sm opn) ∧
        pc ∈ pcOf sm m ∧
        pars ∈ parOf sm m ∧
        locals ∈ localsOf sm m ∧
        caller ∈ UOID sm
 }"

fun callee :: "iFRAME ⇒ iOID"
where
 "callee (c1, n, p, l, pc, c2) = c1"

fun caller :: "iFRAME ⇒ iOID"
where
 "caller (c1, n, p, l, pc, c2) = c2"

fun methodNameOf :: "iFRAME ⇒ Name"
where
 "methodNameOf (c1, n, p, l, pc, c2) = n"

fun localVarAssign :: "iFRAME ⇒ iVarAssign"
where
 "localVarAssign (-, -, localVars, -, -, -) = localVars"

end
```

Definition SysMod B.34 (Thread)

Thread

Isabelle-Theorie

```
theory Thread
imports StackFrame
begin

consts
 UTHREAD :: "SystemModel ⇒ iTHREAD set"

end
```

Definition SysMod B.35 (ControlStore)

ControlStore

Isabelle-Theorie

```
theory ControlStore
imports Thread
begin

fun pControlStore :: "SystemModel ⇒ iControlStore ⇒ bool"
where
 "pControlStore sm ctl = (
    ∀ oid ∈ dom ctl .
      oid ∈ UOID sm ∧
      (∀ th ∈ dom (the (ctl oid)) .
        th ∈ UTHREAD sm ∧
        (∀ i < length (the ((the (ctl oid)) th)) .
          callee ((the (the (ctl oid) th))!i) = oid ∧
          (∀ oid2 ∈ UOID sm .
            caller ((the (the (ctl oid) th))!i) = oid2 ⟶
            (∃ j . callee ((the (the (ctl oid2) th))!j) = oid)
          )
        )
      )
    )"

fun ControlStore :: "SystemModel ⇒ iControlStore set"
where
 "ControlStore sm = { ctl | ctl .
             pControlStore sm ctl }"

end
```

Variante SysMod B.36 (TypeSafeOps)

TypeSafeOps

Isabelle-Theorie

```
theory TypeSafeOps
imports "$SYSTEMMODEL/Operation"
begin

fun valid-TypeSafeOps :: "SystemModel ⇒ bool"
where
 "valid-TypeSafeOps sm = (
    ∀ op1 ∈ UOPN sm . ∀ C ∈ UCLASS sm .
       (sub sm C (classOf sm op1)) ⟶
         (∃ op2 ∈ UOPN sm .
           classOf sm op2 = C ∧
           nameOf op1 = nameOf op2 ∧
           params sm op1 ⊆ params sm op2 ∧
           CAR sm (resType sm op2) ⊆ CAR sm (resType sm op1)
         )
  )"

fun overrides :: "SystemModel ⇒ iOPN ⇒ iOPN ⇒ bool"
where
 "overrides sm op1 op2 = (
     sub sm (classOf sm op1) (classOf sm op2) ∧
     nameOf op1 = nameOf op2 ∧
     params sm op1 ⊆ params sm op2 ∧
     CAR sm (resType sm op2) ⊆ CAR sm (resType sm op1) )"

```

```
end
```

Variante SysMod B.37 (StaticOpn)

StaticOpn

Isabelle-Theorie

```
theory StaticOpn
imports "$SYSTEMMODEL/Method"
        "$SYSTEMMODEL/vData/StaticAttr"
begin

consts
 StaticOpn :: "SystemModel ⇒ iOPN set"

fun valid-StaticOpn :: "SystemModel ⇒ bool"
where
 "valid-StaticOpn sm = (
     (∀ opn ∈ (StaticOpn sm) . classOf sm opn = (StaticC sm)) ∧
     (∀ opn ∈ (StaticOpn sm) . impl sm opn ≠ None ∧
         definedIn sm (the (impl sm opn)) = (StaticC sm))
 )"

end
```

Variante SysMod B.38 (SingleThread)

SingleThread

Isabelle-Theorie

```
theory SingleThread
imports "$SYSTEMMODEL/Thread"
begin

fun valid-SingleThread :: "SystemModel => bool"
where
 "valid-SingleThread sm = (card (UTHREAD sm) = 1)"

end
```

Variante SysMod B.39 (ActiveObjects)

ActiveObjects

Isabelle-Theorie

```
theory ActiveObjects
imports "$SYSTEMMODEL/ControlStore"
begin

fun valid-ActiveObjects :: "SystemModel ⇒ bool"
where
 "valid-ActiveObjects sm = (
    ∃ f .
    ∀ oid1 ∈ UOID sm . ∀ oid2 ∈ UOID sm .
     oid1 ≠ oid2 ⟶ f oid1 ≠ f oid2 ∧
     (∀ ctl ∈ ControlStore sm . ∀ n .
       let frame = (the ((the (ctl oid1)) (f oid1)))!n in
        caller frame = oid1 ∧ callee frame = oid1)
 )"
```

```

end
```

Variante SysMod B.40 (ClassSingleInheritance)

ClassSingleInheritance

Isabelle-Theorie

```
theory ClassSingleInheritance
imports "$SYSTEMMODEL/Method"
begin

fun valid-ClassSingleInheritance :: "SystemModel ⇒ bool"
where
 "valid-ClassSingleInheritance sm = (∀ C1 C2 C3.
   sub sm C1 C2 ∧ sub sm C1 C3 ⟶
    (sub sm C2 C3 ∨ sub sm C3 C2 ∨ isInterface sm C2 ∨ isInterface sm C3)
 )"

end
```

B.6 Messages

Definition SysMod B.41 (Messages)

Messages

Isabelle-Theorie

```
theory Messages
imports Message MethodCall MethodReturn Signal
begin
end
```

Definition SysMod B.42 (Message)

Message

Isabelle-Theorie

```
theory Message
imports Object Thread
begin

fun sender :: "iMessage ⇒ iOID"
where
 "sender (recv, n, values, sndr, th) = sndr"

fun receiver :: "iMessage ⇒ iOID"
where
 "receiver (recv, n, values, sndr, th) = recv"

fun msgparam :: "iMessage ⇒ iVAL list"
where
 "msgparam (recv, n, values, sndr, th) = values"

fun msgname :: "iMessage ⇒ Name"
where
  "msgname (recv, n, values, sndr, th) = n"

fun msgthread :: "iMessage ⇀ iTHREAD"
```

```
where
 "msgthread (recv, ne, values, sndr, th) = th"

fun msgIn :: "SystemModel ⇒ iOID ⇒ iMessage set"
where
 "msgIn sm oid = { msg | msg .
    oid ∈ UOID sm ∧
    receiver msg = oid
 }"

fun msgOut :: "SystemModel ⇒ iOID ⇒ iMessage set"
where
 "msgOut sm oid = { msg | msg .
    oid ∈ UOID sm ∧
    sender msg = oid
 }"

fun MessageStore :: "SystemModel ⇒ iMessageStore set"
where
 "MessageStore sm = { msgst | msgst .
   ∀ oid ∈ dom msgst .
     (set (the (msgst(oid)))) ⊆ msgIn sm oid
 }"

fun messages :: "SystemModel ⇒ iOID ⇒ iMessage set"
where
 "messages sm oid = (msgIn sm oid ∪ msgOut sm oid)"

fun MESSAGE :: "SystemModel ⇒ iMessage set"
where
 "MESSAGE sm = { m | m . ∃ oid ∈ UOID sm . m ∈ messages sm oid }"

fun hasMessageWithName :: "iMessage list ⇒ Name ⇒ bool"
where
  "hasMessageWithName [] n = False"
| "hasMessageWithName (x#xs) n = ((msgname x = n) ∨
                          (hasMessageWithName xs n))"

end
```

Definition SysMod B.43 (MethodCall)

MethodCall

Isabelle-Theorie

```
theory MethodCall
imports Message Thread
begin

fun callsOf :: "SystemModel ⇒ iOID ⇒ iOPN ⇒ iOID ⇒ iTHREAD
                 ⇒ iMessage set"
where
 "callsOf sm recv opn sndr th =
 { (recv, name, values, sndr, (Some th)) |  name values .
    (recv, name, values, sndr, (Some th)) ∈ MESSAGE sm ∧
    name = Operation.nameOf opn ∧
    recv ∈ oids sm (classOf sm opn) ∧
    values ∈ params sm opn
 }"

fun callsOfO :: "SystemModel ⇒ iOID ⇒ iMessage set"
where
```

```
 "callsOfO sm oid = { msg | msg .
    msg ∈ MESSAGE sm ∧
    (∃ s ∈ UOID sm . ∃ opn ∈ UOPN sm . ∃ th ∈ UTHREAD sm .
     msg ∈ callsOf sm oid opn s th)
 }"

end
```

Definition SysMod B.44 (MethodReturn)

MethodReturn

Isabelle-Theorie

```
theory MethodReturn
imports Message Thread
begin

fun returnsOf :: "SystemModel ⇒ iOID ⇒ iOPN ⇒ iOID ⇒ iTHREAD
                    ⇒ iMessage set"
where
 "returnsOf sm recv opn sndr th =
  { (recv, name, values, sndr, (Some th)) |  name values .
    (recv, name, values, sndr, (Some th)) ∈ MESSAGE sm ∧
    name = Operation.nameOf opn ∧
    length values = 1 ∧
    (values!0) ∈ CAR sm (Operation.resType sm opn) ∧
    sndr ∈ oids sm (classOf sm opn)
  }"

fun returnsOfO :: "SystemModel ⇒ iOID ⇒ iMessage set"
where
 "returnsOfO sm oid = { msg | msg .
     msg ∈ MESSAGE sm ∧
     (∃ r ∈ UOID sm . ∃ opn ∈ UOPN sm . ∃ th ∈ UTHREAD sm .
      msg ∈ returnsOf sm r opn oid th)
  }"

end
```

Definition SysMod B.45 (Signal)

Signal

Isabelle-Theorie

```
theory Signal
imports Message MethodCall MethodReturn
begin

consts
 USIGNAL :: "SystemModel ⇒ iMessage set"

fun valid-Signal :: "SystemModel ⇒ bool"
where
 "valid-Signal sm = (
   USIGNAL sm ⊆ MESSAGE sm ∧
   (∀ signal ∈ USIGNAL sm .
    ∀ oid ∈ UOID sm .
     signal ∉ callsOfO sm oid ∧
     signal ∉ returnsOfO sm oid)
 )"
```

```

end
```

B.7 State

Definition SysMod B.46 (State)

State

Isabelle-Theorie

```
theory State
imports Data Control Messages
begin

constdefs STATE :: "SystemModel ⇒ iSTATE set"
 STATE-def :
 "STATE sm == { (ds,cs,ms) | ds cs ms .
      ds ∈ DataStore sm ∧
      cs ∈ ControlStore sm ∧
      ms ∈ MessageStore sm ∧
      dom ds = dom cs ∧
      dom cs = dom ms
 }"

fun oids :: "iSTATE ⇒ iOID set"
where
 "oids (ds,cs,ms) = dom ds"

fun OSTATE :: "SystemModel ⇒ iOSTATE set"
where
 "OSTATE sm = { (a,b,c) | a b c .
    ∃ oid ∈ UOID sm .
      (∃ (ds,cs,ms) ∈ STATE sm .
        (a = the (ds oid) ∧ b = the (cs oid) ∧ c = the (ms oid))
      )
 }"

fun state :: "iSTATE ⇒ iOID ⇀ iOSTATE"
where
 "state (ds,cs,ms) oid =
       (if oid ∈ oids (ds,cs,ms)
        then Some (the (ds oid), the (cs oid), the (ms oid))
        else None)"

fun statesO :: "SystemModel ⇒ iOID ⇒ iOSTATE set"
where
 "statesO sm oid = { ost | ost .
    ∃ st ∈ STATE sm .
       oid ∈ oids st ∧ ost = the (state st oid)
 }"

fun dsOf :: "iSTATE ⇒ iDataStore"
where
 "dsOf (ds,cs,ms) = ds"

fun csOf :: "iSTATE ⇒ iControlStore"
where
 "csOf (ds,cs,ms) = cs"

```

```
fun msOf :: "iSTATE ⇒ iMessageStore"
where
 "msOf (ds,cs,ms) = ms"

fun hasMsg :: "iSTATE ⇒ iOID ⇒ iMessage ⇒ bool"
where
 "hasMsg s oid msg = (msg ∈ set (the ((msOf s) oid)))"

consts running :: "Name ⇒ iOID ⇒ iOID ⇒ iTHREAD ⇒ iSTATE ⇒ bool"

defs running-def : "running mname oid oid′ th s ==
   (methodNameOf (topF (the (the (csOf s oid) th))) = mname ∧
        caller (topF((the ( the ((csOf s) oid) th)))) = oid′)"

consts notExecuting :: "Name ⇒ iSTATE ⇒ iOID ⇒ iOID ⇒ iTHREAD ⇒ bool"

defs notExecuting-def : "notExecuting  mname s oid oid′ th ==
     ∀ fr ∈ set (the (the (csOf s oid) th)) .
      (¬ methodNameOf fr = mname ∧
        caller fr = oid′)"

fun runningNotFinished  :: "SystemModel ⇒ bool"
where
 "runningNotFinished sm =
   (∀ mname s oid oid′ th .
     running mname oid oid′ th s ⟶
          ¬ notExecuting  mname s oid oid′ th)"

fun valid-State :: "SystemModel ⇒ bool"
where
 "valid-State sm = runningNotFinished sm"

end
```

Variante SysMod B.47 (Query)

Query

Isabelle-Theorie

```
theory Query
imports "$SYSTEMMODEL/State"
begin

consts query ::
 "SystemModel ⇒ Name ⇒
   (iSTATE ⇒ iVarAssign ⇒ iOID ⇒ iTHREAD ⇒
               iVAL list ⇒ SystemModel ⇒ iVAL)"

fun valid-Query :: "SystemModel => bool"
where
 "valid-Query sm = True"

end
```

B.8 SMSTS

Definition SysMod B.48 (STS)

STS

Isabelle-Theorie

```
theory STS
imports Main
begin

types (’S, ’I ,’O) iSTS = "’S set × ’I set × ’O set ×
         (’S × ’I list ⇒ (’S × ’O list) set) × ’S set"

fun STS :: "(’S set × ’I set × ’O set) ⇒ (’S, ’I, ’O) iSTS set"
where
 "STS (S,I,OU) = { (S,I,OU, d, s0) | d s0 .
      s0 ⊆ S ∧ s0 ≠ {} ∧
      (∀ s ∈ S . ∀ i . ∃ res .
       res = d (s,i) ∧
       res ≠ {} ∧
       (∀ (ns, ol) ∈ res . ns ∈ S ∧ set ol ⊆ OU)
      )
 }"

fun deltaOf :: "(’S, ’I ,’O) iSTS ⇒  (’S × ’I list ⇒ (’S × ’O list) set)"
where
 "deltaOf (S,I,OU, d, s0) = d"

end
```

Definition SysMod B.49 (SMSTS)

SMSTS

Isabelle-Theorie

```
theory SMSTS
imports OSTS KompO sysSTS Reachable
begin
end
```

Definition SysMod B.50 (OSTS)

OSTS

Isabelle-Theorie

```
theory OSTS
imports State STS
begin

typedecl IN

consts
 osts :: "SystemModel ⇒ iOID ⇒ (iOSTATE,IN,iMessage)iSTS"
 toIn :: "iMessage ⇒ IN"

fun valid-OSTS :: "SystemModel ⇒ bool"
where
 "valid-OSTS sm = (
   ∀ oid ∈ UOID sm .
    (oid ≠ VNil ⟶
      (∃ i . ∀ m ∈ msgIn sm oid . toIn m ∈ i ∧
       osts sm oid ∈ STS(statesO sm oid,i,msgOut sm oid)))
  )"

end
```

Definition SysMod B.51 (KompO)

KompO

Isabelle-Theorie

```
theory KompO
imports OSTS
begin
 (* Die Komposition von Zustandsmaschienen
    wird zukuenftig aus einer Isabelle Kodierung
    von FOCUS uebernommen. *)
end
```

Definition SysMod B.52 (SYSSTS)

SYSSTS

Isabelle-Theorie

```
theory sysSTS
imports OSTS
begin

consts
 sts :: "SystemModel ⇒ iOID set ⇒ (iSTATE, IN, iMessage) iSTS"

fun OSTS :: "SystemModel ⇒ (iSTATE, IN, iMessage) iSTS"
where
 "OSTS sm = (sts sm (UOID sm))"

fun pOSTS :: "SystemModel ⇒ bool"
where
 "pOSTS sm =
     (∃ i . ∀ m ∈ MESSAGE sm  . toIn m ∉ i ∧
      OSTS sm ∈ STS(STATE sm,i, {}))"

consts SYSSTS ::
  "SystemModel ⇒ (iSTATE set × (iSTATE ⇒ iSTATE set) × iSTATE set)"

fun allStates ::
  "(iSTATE set × (iSTATE ⇒ iSTATE set) × iSTATE set) ⇒ iSTATE set"
where
 "allStates (aS, Delta, Init) = aS"

fun Delta ::
  "(iSTATE set × (iSTATE ⇒ iSTATE set) × iSTATE set) ⇒ (iSTATE ⇒ iSTATE set)"
where
 "Delta (aS, x, Init) = x"

fun Init :: "(iSTATE set × (iSTATE ⇒ iSTATE set) × iSTATE set) ⇒ iSTATE set"
where
 "Init (aS, x, i) = i"

fun pSYSSTS :: "SystemModel ⇒ bool"
where
 "pSYSSTS sm =
    (allStates (SYSSTS sm) = STATE sm ∧
     Init (SYSSTS sm) ⊆ STATE sm)"


fun valid-SysSTS :: "SystemModel ⇒ bool"
where
 "valid-SysSTS sm = (pOSTS sm ∧ pSYSSTS sm)"

end
```

Definition SysMod B.53 (Reachable)

Reachable

Isabelle-Theorie

```
theory Reachable
imports sysSTS
begin

inductive-set
    reachable :: "SystemModel ⇒ iSTATE set"
    for sm :: "SystemModel"
where
  init: "⟦s ∈ Init (SYSSTS sm)⟧  ⟹  s ∈ (reachable sm)"
| step: "⟦x ∈ reachable sm; s ∈ (Delta (SYSSTS sm)) x⟧
                                ⟹  s ∈ (reachable sm)"

fun trace :: "SystemModel ⇒ iSTATE list ⇒ bool"
where
 "trace sm run = (run!0 ∈ reachable sm ∧
      (∀ i . run!(i+1) ∈ Delta (SYSSTS sm) (run!i)))"

constdefs TRACES :: "SystemModel ⇒ (iSTATE list) set"
 traces-def : "TRACES sm == { run | run . trace sm run }"

fun following :: "SystemModel ⇒ iSTATE list ⇒ iSTATE ⇒ iSTATE ⇒ bool"
where
 "following sm t s s' = (
    (trace sm t ∧
       (∃ i j . (i < j ∧ t!i = s ∧ t!j = s')))
 )"

fun created :: "SystemModel ⇒ iSTATE list ⇒ iSTATE ⇒ iOID ⇒ bool"
where
 "created sm t s oid =
   (trace sm t ∧
    (∃ i . t!i = s ∧ oid ∈ oids s ∧ oid ∉ oids (t!(-1+i))))"

end
```

Variante SysMod B.54 (Events)

Events

Isabelle-Theorie

```
theory Events
imports "$SYSTEMMODEL/Reachable"
begin

typedecl iEVENT

consts UEVENT :: "SystemModel ⇒ iEVENT set"
```

```
       ReceivedEvent :: "iOID ⇒ iMessage ⇒ iEVENT"
       SentEvent :: "iOID ⇒ iMessage ⇒ iEVENT"
       esOf :: "SystemModel ⇒ iSTATE ⇒ (iOID ⇀ iEVENT set)"

fun msgReceivedNow ::
  "SystemModel ⇒ iOID ⇒ iSTATE ⇒ iMessage ⇒ iSTATE list ⇒ bool"
where
 "msgReceivedNow sm oid s m t = (
    ∃ i . s = t!i ∧ hasMsg s oid m ∧ ¬hasMsg (t!(i+1)) oid m)"

fun msgSentNow ::
  "SystemModel ⇒ iOID ⇒ iSTATE ⇒ iMessage ⇒ iSTATE list ⇒ bool"
where
 "msgSentNow sm oid s m t = (
   ∃ oid' . ∃ i . s = t!i ∧
    sender m = oid ∧ hasMsg s oid' m ∧ ¬hasMsg (t!(i+1)) oid' m
 )"

fun valid-Events :: "SystemModel ⇒ bool"
where
 "valid-Events sm = (
    ∀ t ∈ TRACES sm . ∀ s m oid .
     msgReceivedNow sm oid s m t
         ⟶ ReceivedEvent oid m ∈ (the (esOf sm s oid)) ∧
     msgSentNow sm oid s m t
         ⟶ SentEvent oid m ∈ (the (esOf sm s oid))
  )"

end
```

B.9 SystemModel-base

Definition SysMod B.55 (SystemModel-base)

SystemModel-base

Isabelle-Theorie

```
theory SystemModel-base
imports "$SYSTEMMODEL/SMSTS"
begin

fun valid-base :: "SystemModel ⇒ bool"
where
 "valid-base sm = (
   valid-Type1 sm ∧
   valid-BoolIntVoid sm ∧
   valid-Variable sm ∧
   valid-Class sm ∧
   valid-Nil sm ∧
   valid-Subclassing sm ∧
   valid-Association sm ∧
   valid-Operation sm ∧
   valid-Method1 sm ∧
   valid-Method sm ∧
   valid-State sm ∧
   valid-OSTS sm ∧
   valid-SysSTS sm
  )"

```

```
end
```

B.10 Variabilität des Systemmodells

Definition Variabilität B.56 (Variabilität des Systemmodells)

SystemModel

Feature-Diagramm

```
package systemmodel;

featurediagram SystemModel {

  SystemModel = vType? & vObject? & vData? &
                vControl? & vState? & vSTS?;

  vType = TypeOf? & CharAndString?;

  vObject= LiskovPrinciple? & AntiSymSub? & ValueObjects? &
           StrictSingleInheritance?;

  vData = FiniteObjects? & StaticAttr? &
          AttributeAssociation? & BinaryAssociation? &
          CompositeAssociation? & MtoMAssociation? &
          QualifiedAssociation? & DerivedAssociation?;

  vControl = TypeSafeOps? & SingleOrActive? & ClassSingleInheritance? &
             StaticOpn?;

  SingleOrActive = SingleThread? ^ ActiveObjects?;

  vState = Query?;

  vSTS = Events?;

  constraints {
     vData.AttributeAssociation ->
                    (vData.BinaryAssociation & vObject.LiskovPrinciple);
     vData.MtoMAssociation -> vData.BinaryAssociation;
     vData.CompositeAssociation -> vData.BinaryAssociation;
     vData.QualifiedAssociation -> vData.BinaryAssociation;
     vData.DerivedAssociation -> vData.BinaryAssociation;
     vControl.ClassSingleInheritance -> !vObject.StrictSingleInheritance;
     vControl.StaticOpn -> vData.StaticAttr;
  }

}
```

Anhang C

UML/P-Syntax

Dieser Anhang enthält die syntaktischen Bestandteile der UML/P. In den einzelnen Abschnitten ist immer zuerst die vollständige MontiCore-Grammatik angegeben. Es folgt die hieraus abgeleitete abstrakte Syntax in Isabelle/HOL. Anschließend sind zum Teil Kontextbedingungen und Varianten von Kontextbedingungen angegeben. Die definierte Variabilität wird am Ende eines Abschnitts in einem Feature-Diagramm zusammengefasst. Falls Änderungen an der ursprünglichen Version 1.0 der MontiCore-Grammatik [Sch09] erfolgt sind, werden diese beschrieben.

C.1 Gemeinsam genutzte Sprachteile

C.1.1 Literals

Definition KS C.1 (Literals)

Literals

MontiCore-Grammatik

```
package mc.literals;

/**
@version 1.0a
*/
grammar Literals {

  /*========================================================================*/
  /*============================== OPTIONS =================================*/
  /*========================================================================*/

  options {
    parser lookahead=3
    lexer lookahead=3
    identrule Name
    nostring noident nows
  }

  /*========================================================================*/
  /*======================= INTERFACE DEFINITIONS ==========================*/
  /*========================================================================*/

  ast BooleanLiteral =
    method public boolean getValue(){
      return this.source == ASTConstantsLiterals.TRUE;
  };

  ast CharLiteral =
    method public char getValue() {
      try {
        return
          mc.literals.LiteralsHelper.getInstance().decodeChar(source);
      }
      catch (java.io.CharConversionException e) {
        return ’ ’;
      }
  };

  ast StringLiteral =
    method public String getValue() {
      try {
        return
          mc.literals.LiteralsHelper.getInstance().decodeString(source);
      }
      catch (Exception e) {
        return "";
      }
  };

  ast IntLiteral =
    method public int getValue() {
      try {
        return
          mc.literals.LiteralsHelper.getInstance().decodeInt(source);
      }
      catch (NumberFormatException e) {
        return 0;
      }
  };

  ast LongLiteral =
    method public long getValue() {
       try {
         return
           mc.literals.LiteralsHelper.getInstance().decodeLong(source);
       }
       catch (NumberFormatException e) {
         return 0;
       }
  };

  ast FloatLiteral =
    method public float getValue() {
      try {
        return
          mc.literals.LiteralsHelper.getInstance().decodeFloat(source);
      }
      catch (NumberFormatException e) {
        return 0f;
      }
  };

  ast DoubleLiteral =
    method public double getValue() {
      try {
        return
          mc.literals.LiteralsHelper.getInstance().decodeDouble(source);
      }
      catch (NumberFormatException e) {
        return 0d;
      }
```

```
};

/** ASTLiteral is the interface for all literals (NullLiteral,
    BooleanLiteral, CharLiteral, StringLiteral and all NumericLiterals)
*/
interface Literal;

/** The interface ASTNumericLiteral combines the numeric literal types for
    Integer, Long, Float and Double
*/
interface NumericLiteral extends Literal;


/*========================================================================*/
/*============================ PARSER RULES ==============================*/
/*========================================================================*/

/** ASTNullLiteral represents ’null’
*/
NullLiteral implements Literal =
  "null";

/** ASTBooleanLiteral represents "true" or "false"
    @attribute source String-representation (including ’"’).
*/
BooleanLiteral implements Literal =
  source:["true" | "false"];

/** ASTCharLiteral represents any valid character parenthesized with "’".
    @attribute source String-representation (including "’").
*/
CharLiteral implements Literal =
  source:CHAR;

/** ASTStringLiteral represents any valid character sequence parenthesized
    with ’"’.
    @attribute source String-representation (including ’"’).
*/
StringLiteral implements Literal =
  source:STRING;

/** ASTIntLiteral represents an Integer number.
    @attribute source String-representation (including ’"’).
*/
IntLiteral implements NumericLiteral =
  source:NUM_INT;

/** ASTLongLiteral represents a Long number.
    @attribute source String-representation (including ’"’).
*/
LongLiteral implements NumericLiteral =
  source:NUM_LONG;

/** ASTFloatLiteral represents a Float number.
    @attribute source String-representation (including ’"’).
*/
FloatLiteral implements NumericLiteral =
  source:NUM_FLOAT;

/** ASTDoubleLiteral represents a Double number.
    @attribute source String-representation (including ’"’).
*/
```

```
DoubleLiteral implements NumericLiteral =
  source:NUM_DOUBLE;

/*========================================================================*/
/*============================ LEXER RULES ===============================*/
/*========================================================================*/

// Definition of IDENTs (Java-style)
ident Name
  options {testLiterals=true;} = // check Literals first
  (’a’..’z’ | ’A’..’Z’ | ’_’ | ’$’)
  (’a’..’z’ | ’A’..’Z’ | ’_’ | ’0’..’9’ | ’$’)*;

// Character literals
ident CHAR =
  ’\’’ ( ESC | ~(’\’’ | ’\n’ | ’\r’ | ’\\’) ) ’\’’;

// String literals
ident STRING =
  ’"’ (ESC | ~(’"’ | ’\\’ | ’\n’ | ’\r’))* ’"’;

// Hexadecimal digit (used inside NUMBERS)
protected ident HEX_DIGIT =
  (’0’..’9’|’A’..’F’|’a’..’f’);

// Suffix of float numbers (used inside NUMBERS)
protected ident FLOAT_SUFFIX =
   ’f’|’F’|’d’|’D’;

// Exponent for decimal numbers, used inside NUMBERS
protected ident DECIMAL_EXPONENT =
   (’e’|’E’) (’+’|’-’)? (’0’..’9’)+;

// Exponent for hexadecimal numbers, used inside NUMBERS
protected ident HEX_EXPONENT =
  (’p’|’P’) (’+’|’-’)? (’0’..’9’)+;

// Numbers
ident NUMBERS
  options {testLiterals=true;}
  {boolean isDecimal=false, isHex=false, isDoubleDot=false;} =
  (’.’
    (
      (’.’)
      |
      (’.’ ’.’)
      |
      (
        (’0’..’9’)+ (DECIMAL_EXPONENT)? (t=FLOAT_SUFFIX)?
        {
           if (t != null && t.getText().toUpperCase().indexOf(’F’)>=0){
             _ttype = NUM_FLOAT;
           }else{
             _ttype = NUM_DOUBLE; // assume double
           }
        }
      )?
    )
  )
  |
  (
    (’0’ {isDecimal = true; _ttype = NUM_INT;} // special case for ’0’
        (
           // HEX
           ((’x’|’X’) ((HEX_DIGIT)+ | (’.’ (HEX_DIGIT)+) | HEX_EXPONENT)) =>
           (
             (’x’|’X’)
             // The decimal exponent and the float suffix look like
             // hex digits, hence the (...)* doesn’t know when to stop
             // (-> ambig). ANTLR resolves it correctly by matching
             // immediately. It is therefore ok to hush warning.
             (options {warnWhenFollowAmbig=false;}: HEX_DIGIT)*
             {isHex = true;}
           )
          |
          // FLOAT or DOUBLE with leading zero
          ((’0’..’9’)+ (’.’|DECIMAL_EXPONENT|FLOAT_SUFFIX)) => (’0’..’9’)+
          |
          // OCTAL
          (’0’..’7’)+
        )?
      )
      |
      // NON-ZERO DECIMAL
      ((’1’..’9’) (’0’..’9’)*  {isDecimal=true; _ttype = NUM_INT;})
    )
    {
      if (LA(1)==’.’ && LA(2)==’.’)
        isDoubleDot = true;
      else
        isDoubleDot = false;
      }
    (
       (’l’|’L’) {_ttype = NUM_LONG;}

       // only check to see if it’s a float if looks like decimal so far
       |
       {isDecimal && !isDoubleDot}?
       (
         {isHex}?
         ( // exponent is mandatory for floating point hex digits
           ’.’ (HEX_DIGIT)* HEX_EXPONENT (t=FLOAT_SUFFIX)?
           |
           HEX_EXPONENT (t=FLOAT_SUFFIX)?
           |
           t=FLOAT_SUFFIX
         )
         |
         (
           ’.’ (’0’..’9’)* (DECIMAL_EXPONENT)? (t=FLOAT_SUFFIX)?
           |
           DECIMAL_EXPONENT (t=FLOAT_SUFFIX)?
           |
           t=FLOAT_SUFFIX
         )
      )
      {
        if (t != null && t.getText().toUpperCase().indexOf(’F’) >= 0) {
          _ttype = NUM_FLOAT;
        }
        else {
          _ttype = NUM_DOUBLE; // assume double
        }
      }
```

```
   )?;

  // Escape sequence -- note that this is protected; it can only be called
  // from another lexer rule -- it will not ever directly return a token to
  // the parser.
  // There are various ambiguities hushed in this rule. The optional
  // ’0’...’9’ digit matches should be matched here rather than letting them
  // go back to STRING to be matched. ANTLR does the right thing by matching
  // immediately; hence, it’s ok to shut off the FOLLOW ambig warnings.
  protected ident ESC =
    ’\\’
    (
      ’n’ | ’r’ | ’t’ | ’b’ | ’f’ | ’"’ | ’\’’ | ’\\’ | (’u’)+
      HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT | ’0’..’3’
      (
        options {warnWhenFollowAmbig = false;} : ’0’..’7’
        (options {warnWhenFollowAmbig = false;} : ’0’..’7’)?
      )?
      |
      ’4’..’7’ (options {warnWhenFollowAmbig = false;} : ’0’..’7’)?
    );

  // Whitespace -- ignored
  ident WS =
    (
      ’ ’ | ’\t’ | ’\f’
      | (
          options {generateAmbigWarnings=false;}:
          "\r\n" | ’\r’ | ’\n’
        )
        {newline();}
    )+
    {_ttype = Token.SKIP;};

}
```

Definition AS C.2 (LiteralsAS)

LiteralsAS

Isabelle-Theorie (gen)

```
(*
Source: mc.literals.Literals V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:19 CEST 2010
*)
theory LiteralsAS
imports "Main"
begin

types NUM-INT = "int"

types CHAR = "char"

types NUM-FLOAT = "char list"

types STRING = "char list"

types NUM-LONG = "int"

types NUM-DOUBLE = "char list"

types NUMBERS = "char list"

types Name = "char list"

datatype CharLiteral =
    CharLiteral CHAR

datatype StringLiteral =
    StringLiteral STRING

datatype NullLiteral =
    NullLiteral

datatype BooleanLiteralsource =
    BooleanLiteralsourceFALSE
  | BooleanLiteralsourceTRUE

datatype BooleanLiteral =
    BooleanLiteral BooleanLiteralsource

datatype IntLiteral =
    IntLiteral NUM-INT

datatype LongLiteral =
    LongLiteral NUM-LONG

datatype FloatLiteral =
    FloatLiteral NUM-FLOAT

datatype DoubleLiteral =
    DoubleLiteral NUM-DOUBLE

datatype NumericLiteral =
    NumericLiteralIntLiteral IntLiteral
  | NumericLiteralLongLiteral LongLiteral
  | NumericLiteralFloatLiteral FloatLiteral
  | NumericLiteralDoubleLiteral DoubleLiteral

datatype Literal =
    LiteralNullLiteral NullLiteral
  | LiteralBooleanLiteral BooleanLiteral
  | LiteralCharLiteral CharLiteral
  | LiteralStringLiteral StringLiteral
  | LiteralNumericLiteral NumericLiteral

end
```

Variante KB C.3 (Keine Fließkomma-Werte)

RestrictFloatTypes

Isabelle-Theorie

```
theory RestrictFloatTypes
imports "$UMLP/gen/mc/literals/LiteralsAS"
begin

fun floatUnused :: "NumericLiteral ⇒ bool"
where
  "floatUnused (NumericLiteralIntLiteral i) = True"
| "floatUnused (NumericLiteralLongLiteral l) = True"
| "floatUnused (NumericLiteralFloatLiteral f) = False"
| "floatUnused (NumericLiteralDoubleLiteral d) = False"
```

```

fun wellformed-RestrictFloatTypes :: "Literal ⇒ bool"
where
  "wellformed-RestrictFloatTypes (LiteralNullLiteral n) = True"
| "wellformed-RestrictFloatTypes (LiteralBooleanLiteral n) = True"
| "wellformed-RestrictFloatTypes (LiteralCharLiteral c) = True"
| "wellformed-RestrictFloatTypes (LiteralStringLiteral s) = True"
| "wellformed-RestrictFloatTypes (LiteralNumericLiteral n) =
      floatUnused n"

end
```

Definition Variabilität C.4 (Spracheinschränkungen Literals)

LiteralsConstr

Feature-Diagramm

```
package mc.literals.syntax;

featurediagram LiteralsConstr {

 LiteralsConstr = RestrictFloatTypes?;

}
```

C.1.2 Types

Definition KS C.5 (Types)

Types

MontiCore-Grammatik

```
package mc.types;

/**
This grammar defines Java compliant types. The scope of this grammar is to
ease the reuse of type structures in Java-like sublanguages, e.g., by grammar
inheritance or grammar embedment.
The grammar contains types from Java, e.g., primitives, void, types with
dimensions, reference types, generics, and type parameters.

@version 1.0a
*/
grammar Types extends mc.literals.Literals {

  /*========================================================================*/
  /*============================== OPTIONS =================================*/
  /*========================================================================*/

  options {
    parser lookahead=3
    lexer lookahead=3
    nostring noident nows
  }

  concept antlr {
    parser java {
      /**
       * Counts the number of LT of type parameters and type arguments.
       * It is used in semantic predicates to ensure the right number
```

```
     * of closing ’>’ characters; which actually may have been
     * either GT, SR (GTGT), or BSR (GTGTGT) tokens.
    */
    public int ltCounter = 0;
  }
}


/*========================================================================*/
/*======================= INTERFACE DEFINITIONS ==========================*/
/*========================================================================*/
ast QualifiedName =
      method public String toString(){
          return mc.helper.NameHelper.dotSeparatedStringFromList(
              this.getParts());
      };

ast PrimitiveType =
  method public String toString(){
    if (this.getPrimitive()==ASTConstantsTypes.BOOLEAN){
      return "boolean";
    }
    if (this.getPrimitive()==ASTConstantsTypes.BYTE){
      return "byte";
    }
    if (this.getPrimitive()==ASTConstantsTypes.CHAR){
      return "char";
    }
    if (this.getPrimitive()==ASTConstantsTypes.SHORT){
       return "short";
    }
    if (this.getPrimitive()==ASTConstantsTypes.INT){
       return "int";
    }
    if (this.getPrimitive()==ASTConstantsTypes.FLOAT){
       return "float";
    }
    if (this.getPrimitive()==ASTConstantsTypes.LONG){
       return "long";
    }
    if (this.getPrimitive()==ASTConstantsTypes.DOUBLE){
       return "double";
    }
    return "";
  };

ast ReferenceType astextends Type;

ast ArrayType astimplements ReferenceType =
  dimensions:/int
  componentType:Type;

/** ASTType defines types like primitives, Set, List, Collection, or
    class types. It might also be an array or generic type.
*/
interface Type = ComplexArrayType | PrimitiveArrayType;

/** ASTReferenceType defines a reference type like arrays or complex types.
*/
interface ReferenceType = SimpleReferenceType;

/** ASTTypeArgument represents a type argument (generics).
```

```
*/
interface TypeArgument = WildcardType | Type;

/** ASTReturnType represents return types.
*/
interface ReturnType = Type | VoidType;


/*========================================================================*/
/*============================ PARSER RULES ==============================*/
/*========================================================================*/

/** The ASTQualifiedName represents a single or qualified name in the AST. The
    different parts of a qualified name are separated by ’.’; they are
    stored in an ASTStringList.
    @attribute parts A List of ASTStringList concludes all name parts
*/
QualifiedName =
  parts:Name (options{greedy=true;}: "." parts:Name)*;

/** The ASTArrayType represents an array of any type. The rule
    ComplexArrayType itself treats all arrays except the primitive
    ones. Especially it treats generic types.
    @attribute componentType The kind of type which is used for the array.
                             Could be every complex type.
    @attribute dimensions    Counts the number of ’[]’
*/
ComplexArrayType: ArrayType returns Type
{int ltLevel = ltCounter;} =
  // Things are getting ugly. We have to disambiguate between
  // Class<List<Anything>>[]
  // and
  // Class<List<Anything>[]>
  // So we have to make sure that we only take the array dimensions if we
  // are at the right level. If we would not, we wouldn’t know that ">>"
  // actually closes two levels and not just one as ">" does. It would
  // then count the array dimensions to the inner type resulting in the
  // same tree for each of the cases above.
  ret=ComplexType
  (
    {ltCounter==ltLevel}?
    (
       astscript{!(ComponentType=ret;);}
       (
          options{greedy=true;}: "[" "]"
          {a.setDimensions(a.getDimensions()+1);}
        )+
     )
     | /* let the dims for the enclosing type reference */
  );

/** The ASTArrayType represents an array of any type. The rule
    PrimitiveArrayType itself treats arrays of primitive types, such as
    ’int[]’.
    @attribute componentType The kind of which is used for the array.
                             Could be every primitive type.
    @attribute dimensions    Counts the number of ’[]’
*/
PrimitiveArrayType: ArrayType returns Type =
  ret=PrimitiveType
  (
     options{greedy=true;}:
        astscript{!(ComponentType=ret;);}
        (
           options{greedy=true;}: "[" "]"
           {a.setDimensions(a.getDimensions()+1);}
         )+
     )?;

   /** ASTVoidType represents the return type "void".
   */
   VoidType =
     "void";

   /** The BooleanType rule represents boolean primitive types (BOOLEAN).
       An instance will be of type ASTPrimitiveType.
   */
   BooleanType:PrimitiveType =
     primitive: ["boolean"];

   /** The IntegralType rule represents integral primitive types
       (BYTE, SHORT, INT, LONG, or CHAR).
       An instance will be of type ASTPrimitiveType.
   */
   IntegralType:PrimitiveType =
     primitive: ["byte" | "short" | "int" | "long" | "char"];

   /** The FloatingPointType rule represents floating point primitive types
       (LONG or DOUBLE).
       An instance will be of type ASTPrimitiveType.
   */
   FloatingPointType:PrimitiveType =
     primitive: ["float" | "double"];

   /** The NumericType rule represents numeric types
       (BOOLEAN, BYTE, CHAR, SHORT, INT, FLOAT, LONG, or DOUBLE).
       An instance will be of type ASTPrimitiveType.
   */
   NumericType:PrimitiveType =
     ret=FloatingPointType | ret=IntegralType;

   /** ASTPrimitiveType represents every primitive type supported by Java.
       @attribute primitive BOOLEAN, BYTE, CHAR, SHORT, INT, FLOAT, LONG,
                            or DOUBLE
   */
   PrimitiveType implements Type =
     ret=FloatingPointType | ret=IntegralType | ret=BooleanType;

   /** The ComplexType rule represents a complex type (in contrast to
       PrimitiveTypes; e.g. class or interface types). It handels
       SimpleReferenceTypes and QualifiedTypes. An instance will be of type
       ASTType.
   */
   ComplexType returns Type =
     ret=SimpleReferenceType
     (options{greedy=true;}: "." ret=QualifiedType->[ret])*;

   /** ASTSimpleReferenceType represents types like class or interface types
       which could have a qualified name like this: a.b.c<Arg>. The
       qualification stored in the name (a.b) could be package or a type name.
       The qualified name could contain type arguments only at the end.
       a.b.c<Arg>.d would be one ASTSimpelReferenceType (a.b.c<Arg>) and one
       ASTQualifiedType (d).
       @attribute name          Name of the type
```

```
                             Note: Allthough the class name contains the
                             word 'simple', the name could be a qualified
                             one. So it is saved in an ASTStringList
    @attribute typeArguments The types between '<...>'
*/
SimpleReferenceType implements ReferenceType =
  Name
  (options{greedy=true;}: "." Name)*
  (options {greedy=true;}: TypeArguments)?;

/** ASTQualifiedType represents types like class or interface types which
    always have another ASTQualifiedType or ASTSimpelReferenceType as
    qualification. So the qualification is in every case a type.
    For example:
    a.b.c<Arg>.d.e<Arg> would be one ASTSimpelReferenceType (a.b.c<Arg>)
    and two ASTQualifiedType (d; e<Arg>)
    @attribute name          Name of the type
    @attribute typeArguments The types between '<...>'
    @attribute qualification Another ASTQualifiedType or
                             ASTSimpelReferenceType.
*/
QualifiedType [qualification:Type] implements ReferenceType  =
  Name
  (options {greedy=true;}: TypeArguments)?;

/** ASTTypeArguments represents a list of generic arguments parenthesized
    by '<...>'. It is also possible to nest type arguments in each other
    like this <A<B<C>>>.
    @attribute typeArguments List of arguments
*/
TypeArguments {int currentLtLevel = 0;} =
  {currentLtLevel = ltCounter;}
  (
    "<" {ltCounter++;}
    typeArguments:TypeArgument
    (options{greedy=true;}:
      /*
       * The following semantic predicates are needed to construct
       * trees properly in case of ">>" or ">>>" tokens within nested
       * TypeArguments (e.g., "var<O1<I1<M1>>, O2<I2>> a;").
       */
      {inputState.guessing !=0 || ltCounter == currentLtLevel + 1}?
         "," typeArguments:TypeArgument
    )*
    (
      /*
       * The token stream contains GT, GTGT, or GTGTGT tokens,
       * so the parser has to expect one of these three possibilities.
       * Furthermore the angle brackets are counted to check if there
       * are some left open. After parsing the ltCounter has to be 0,
       * what is checked by the predicate below.
       */
      options{generateAmbigWarnings=false;}:
      ">" {ltCounter-=1;}
      | ">>" {ltCounter-=2;}
      | ">>>" {ltCounter-=3;}
    )?
  )
  // This predicate checks if we have a valid angle bracket structure.
  {(currentLtLevel != 0) || ltCounter == currentLtLevel}?;

/** ASTWildcardType represents a wildcard type in a type argument (generics)
```

```
      It could also contain an upper- or lower bound.
      @attribute upperBound Supertye of the type argument
      @attribute lowerBound Subtype of the type argument
  */
  WildcardType implements TypeArgument =
    "?" (
          options{greedy=true;}:
          ("extends" upperBound:Type) | ("super" lowerBound:Type)
         )?;

  /** ASTTypeParameters represents a list of generic parameter parenthesized
      by '<...>' in type declarations (e.g., class-, interface-, method-, or
      constructor declarations).
      @attribute typeVariableDeclarations List of parameters
  */
  TypeParameters {int currentLtLevel = 0;} =
    {currentLtLevel = ltCounter;}
    (
      options{greedy=true;}:
      "<" {ltCounter++;}
      typeVariableDeclarations:TypeVariableDeclaration
      (
        options{greedy=true;}:
        "," typeVariableDeclarations:TypeVariableDeclaration
      )*
      ( // inlined typeArgumentsEnd
         options{generateAmbigWarnings=false;}:
         ">" {ltCounter-=1;}
         | ">>" {ltCounter-=2;}
         | ">>>" {ltCounter-=3;}
      )?
    )
    // This predicate checks if we have a valid angle bracket structure
    // (if we are at the "top level" of nested TypeArgument productions).
    {(currentLtLevel != 0) || ltCounter == currentLtLevel}?
    | /* nothing! (instead of optionality) */;

  /** ASTTypeVariableDeclaration represents the generic variable declaration
      in '<'...'>' (e.g., in front of method or constructor declarations or
      behind the class or interface name).
      E.g.: public <T extends SuperClass> void test(T t)
      @attribute name        Name of the type variable
      @attribute upperBounds Optional list of required super classes
  */
  TypeVariableDeclaration =
    Name
    (
      options{generateAmbigWarnings=false;}:
      "extends" upperBounds:ComplexType
      ("&" upperBounds:ComplexType)*
    )?;

}
```

Definition AS C.6 (TypesAS)

TypesAS

Isabelle-Theorie (gen)

```
(*
Source: mc.types.Types V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:20 CEST 2010
*)
theory TypesAS
imports "$UMLP/gen/mc/literals/LiteralsAS"
begin

datatype PrimitiveTypeprimitive =
    PrimitiveTypeprimitiveCHAR
  | PrimitiveTypeprimitiveDOUBLE
  | PrimitiveTypeprimitiveBYTE
  | PrimitiveTypeprimitiveSHORT
  | PrimitiveTypeprimitiveFLOAT
  | PrimitiveTypeprimitiveBOOLEAN
  | PrimitiveTypeprimitiveINT
  | PrimitiveTypeprimitiveLONG

datatype PrimitiveType =
    PrimitiveType "PrimitiveTypeprimitive option"

datatype ComplexType =
    ComplexType

datatype QualifiedType =
    QualifiedType Name "TypeArguments option" "Type option"
and
WildcardType =
    WildcardType "Type option" "Type option"
and
TypeArgument =
    TypeArgumentWildcardType WildcardType
  | TypeArgumentType Type
and
TypeArguments =
    TypeArguments "TypeArgument list"
and
SimpleReferenceType =
    SimpleReferenceType "Name list" "TypeArguments option"
and
ReferenceType =
    ReferenceTypeArrayType ArrayType
  | ReferenceTypeSimpleReferenceType SimpleReferenceType
  | ReferenceTypeQualifiedType QualifiedType
and
ArrayType =
    ArrayType int Type
and
Type =
    TypeArrayType ArrayType
  | TypePrimitiveType PrimitiveType
  | TypeComplexType ComplexType
  | TypeReferenceType ReferenceType

datatype VoidType =
    VoidType

datatype ReturnType =
    ReturnTypeVoidType VoidType
  | ReturnTypeType Type

datatype TypeVariableDeclaration =
    TypeVariableDeclaration Name "Type list"

```

```
datatype TypeParameters =
    TypeParameters "TypeVariableDeclaration list"

datatype QualifiedName =
    QualifiedName "Name list"

end
```

Variante KB C.7 (Typeinschränkungen)

RestrictTypes

Isabelle-Theorie

```
theory RestrictTypes
imports "$UMLP/gen/mc/types/TypesAS"
begin

fun restrictPrimitiveTypes :: "PrimitiveType ⇒ bool"
where
 "restrictPrimitiveTypes
    (PrimitiveType (Some PrimitiveTypeprimitiveCHAR))    = True"
|"restrictPrimitiveTypes
    (PrimitiveType (Some PrimitiveTypeprimitiveDOUBLE))  = False"
|"restrictPrimitiveTypes
    (PrimitiveType (Some PrimitiveTypeprimitiveBYTE))    = False"
|"restrictPrimitiveTypes
    (PrimitiveType (Some PrimitiveTypeprimitiveSHORT))   = False"
|"restrictPrimitiveTypes
    (PrimitiveType (Some PrimitiveTypeprimitiveFLOAT))   = False"
|"restrictPrimitiveTypes
    (PrimitiveType (Some PrimitiveTypeprimitiveBOOLEAN)) = True"
|"restrictPrimitiveTypes
    (PrimitiveType (Some PrimitiveTypeprimitiveINT))     = True"
|"restrictPrimitiveTypes
    (PrimitiveType (Some PrimitiveTypeprimitiveLONG))    = False"

fun restrictReferenceTypes :: "ReferenceType ⇒ bool"
where
 "restrictReferenceTypes (ReferenceTypeArrayType t)     = False"
|"restrictReferenceTypes (ReferenceTypeQualifiedType t) = False"
|"restrictReferenceTypes (ReferenceTypeSimpleReferenceType
       (SimpleReferenceType idl typeArgs)) = (
         if (typeArgs = None) then True else False)"

fun wellformed-RestrictTypes :: "Type ⇒ bool"
where
  "wellformed-RestrictTypes (TypeArrayType x) = False"
| "wellformed-RestrictTypes (TypeReferenceType refT) =
    restrictReferenceTypes refT"
| "wellformed-RestrictTypes (TypePrimitiveType p) =
    restrictPrimitiveTypes p"

end
```

Definition Variabilität C.8 (Spracheinschränkungen Types)

TypesConstr

Feature-Diagramm

```
package mc.types.syntax;

featurediagram TypesConstr {

  TypesConstr = RestrictTypes?;

}
```

C.1.3 Common

Definition KS C.9 (Common)

Common

MontiCore-Grammatik

```
package mc.umlp.common;

/**
@version 1.0a
*/
grammar Common extends mc.types.Types {

  options {
    parser lookahead=3
    lexer lookahead=7
    nostring noident nows
  }

  /*========================================================================*/
  /*======================= INTERFACE DEFINITIONS ==========================*/
  /*========================================================================*/

  ast StereoValue =
    method public String getValue() {
      try {
        return
          mc.literals.LiteralsHelper.getInstance().decodeString(source);
      }
      catch (Exception e) {
        return "";
      }
    }
    method public void setValue(String value) {
      this.source = ’"’ + value + ’"’;
    };


  /*========================================================================*/
  /*============================== GRAMMAR =================================*/
  /*========================================================================*/

  /** ASTStereotype represents Stereotypes in the UML/P
      @attribute values List of Values of this Stereotype
  */
  Stereotype =
    "<<" values:StereoValue ("," values:StereoValue)* ">>";

```

```
/** ASTStereoValue represents a Value of a Steretype in the UML/P
    @attribute name  Name of the Stereotype-Value
    @attribute source Source of the Stereotype (String including '"';
                      use getValue() for decoded String)
*/
StereoValue =
  Name ("=" source:STRING)?;

/** ASTCardinality represents a Cardinality in the UML/P
    @attribute many         True if "*" is set as Cardinality
    @attribute lowerBound   Minimum number of associated Classes/Ojects
    @attribute upperBound   Maximum number of associated Classes/Ojects
    @attribute noUpperLimit True if no upper bound exists
*/
Cardinality =
  "["
  (
    many:["*"]
    |
    (lowerBound:NUM_INT {a.setUpperBound(a.getLowerBound());})
    |
    (
      lowerBound:NUM_INT
       ".."
      (upperBound:NUM_INT | noUpperLimit:["*"])
    )
  )
  "]";

/** ASTQualifier represents a Qualifier in the UML/P
    @attribute typeOrName Type of the Qualifier if not an abstract name
                          (i.e. attribute name)
    @attribute name       Abstract name of the Qualifier if not a type
*/
Qualifier =
  "[" typeOrName:Type "]";

/** ASTCompleteness represents the completeness in the UML/P
    Syntax: (left-completeness, right-completeness)
    Interpretation:
      CD: Diagramm: left: Classes, right: Assoziations
          Classes, Enums, Interfaces: left: Attributes, right: Methods
      OD: Diagramm: left=right: Links of visible Objects
          Objects: left=right: Attributes
      SC: Diagramm: left: state space coverage,
                    right: Events of visible States
          States: left: Inner-States, right: Inner-Transitions
                  (including Do-, Entry-, and ExitAction)
      SD: Diagramm: left: involved Objects
                    right: Interactions visible Objects
          Objects: left=right:  (c) = <<match:complete>>,
                                (...) = <<match:free>>
    @attribute incomplete    true if left and right side are
                             incomplete (...)
    @attribute complete      true if left and right side are
                             complete (c)
    @attribute rightComplete true if only right side is complete (...,c)
    @attribute leftComplete  true if only left is complete (c,...)
*/
enum Completeness =
      complete:"(c)"
    | incomplete:"(...)"
```

```
      | incomplete:"(...,...)"
      | complete:"(c,c)"
      | incomplete:"(..., ...)"
      | complete:"(c, c)"
      | rightComplete:"(...,c)"
      | leftComplete:"(c,...)"
      | rightComplete:"(..., c)"
      | leftComplete:"(c, ...)";

  /** ASTModifier represents a Modifier for Classes, Interfaces, Methods,
      Constructors and Attributes in the UML/P
      @attribute stereotype Optional Stereotype
      @attribute public     true if Modifier is public
                            (i.e. Modifier written as "public" or "+")
      @attribute private    true if Modifier is private
                            (i.e. Modifier written as "private" or "-")
      @attribute protected  true if Modifier is protected
                            (i.e. Modifier written as "protected" or "#")
      @attribute final      true if Modifier is final
                            (i.e. Modifier written as "final")
      @attribute abstract   true if Modifier is abstract
                            (i.e. Modifier written as "abstract")
      @attribute local      true if Modifier is local
                            (i.e. Modifier written as "local")
      @attribute derived    true if Modifier is derived
                            (i.e. Modifier written as "derived" or "/")
      @attribute readonly   true if Modifier is readonly
                            (i.e. Modifier written as "readonly" or "?")
      @attribute static     true if Modifier is static
                            (i.e. Modifier written as "static")
  */
  Modifier =
    Stereotype?
    ModifierVal*;

  enum ModifierVal =
     Public:"public" | Public:"+"
   | Private:"private" | Private:"-"
   | Protected:"protected" | Protected:"#"
   | Final:"final"
   | Abstract:"abstract"
   | Local:"local"
   | Derived:"derived" | Derived:"/"
   | Readonly:"readonly" | Readonly:"?"
   | Static:"static";

}
```

Die Grammatik aus Definition C.9 wurde im Vergleich zur ursprünglichen Version 1.0 [Sch09] angepasst. Die Änderungen und die Gründe hierfür sind:

- Die Produktion `Modifier` verwendet eine Enumeration, vorher wurden Konstantendefinitionen verwendet, die zu einer unintuitiven abstrakten Syntax führten. Die Anpassung ändert die Ausdrucksmächtigkeit der Sprache nicht.
- In allen UML/P-Diagrammarten können an verschiedenen Stellen Invarianten verwendet werden. Die Produktion zur Einbettung von Invarianten war ursprünglich in dieser

Grammatik definiert. Sie wurde in alle Untergrammatiken verschoben, damit die Parameterbelegung für die Einbettung in Isabelle separat für jede Sprache erfolgen kann. Damit ist methodisch auch etwas klarer geregelt, dass das Sprachfragment Common nur einmal konfiguriert und diese Konfiguration dann einheitlich verwendet wird. Zudem führt die Abhängigkeit von Java/P von Common dazu, dass sonst bei Einbettung von Java/P ein zirkulärer Import von Theorien entstünde, der in Isabelle nicht erlaubt ist.

Definition AS C.10 (CommonAS)

CommonAS

Isabelle-Theorie (gen)

```
(*
Source: mc.umlp.common.Common V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:23 CEST 2010
*)
theory CommonAS
imports "$UMLP/gen/mc/types/TypesAS"
begin

datatype StereoValue =
    StereoValue Name "STRING option"

datatype Stereotype =
    Stereotype "StereoValue list"

datatype ModifierVal =
    ModifierValDERIVED
  | ModifierValFINAL
  | ModifierValPROTECTED
  | ModifierValABSTRACT
  | ModifierValLOCAL
  | ModifierValREADONLY
  | ModifierValSTATIC
  | ModifierValPRIVATE
  | ModifierValPUBLIC

datatype Modifier =
    Modifier "Stereotype option" "ModifierVal list"

datatype CardinalitynoUpperLimit =
    CardinalitynoUpperLimitSTAR

datatype Cardinalitymany =
    CardinalitymanySTAR

datatype Qualifier =
    Qualifier Type

datatype Completeness =
    CompletenessLEFTCOMPLETE
  | CompletenessINCOMPLETE
  | CompletenessCOMPLETE
  | CompletenessRIGHTCOMPLETE

datatype Cardinality =
    Cardinality "NUM-INT option" "CardinalitynoUpperLimit option"
      "NUM-INT option" "Cardinalitymany option"

```

```
end
```

Variante KB C.11 (Nur eine Sichtbarkeit)

Modifier

Isabelle-Theorie

```
theory Modifier
imports "$UMLP/gen/mc/umlp/common/CommonAS"
begin

fun getModifiers :: "ModifierVal ⇒ ModifierVal list ⇒ ModifierVal list"
where
  "getModifiers x [] = []"
| "getModifiers x (y#ys) = (
     if (x = y)
     then (y#(getModifiers x ys))
     else (getModifiers x ys))"

fun wellformed-ModifierList :: "ModifierVal list ⇒ bool"
where
 "wellformed-ModifierList modList = (
     length (getModifiers ModifierValPUBLIC modList) +
     length (getModifiers ModifierValPRIVATE modList) +
     length (getModifiers ModifierValPROTECTED modList) <= 1)"

fun wellformed-Modifier :: "Modifier ⇒ bool"
where
 "wellformed-Modifier (Modifier stereoOpt modList)
    = wellformed-ModifierList modList"

end
```

Definition Variabilität C.12 (Spracheinschränkungen Common)

CommonConstr

Feature-Diagramm

```
package mc.umlp.common.syntax;

featurediagram CommonConstr {

  CommonConstr = Modifier?;

}
```

C.2 Java/P und OCL/P

C.2.1 Java/P

Definition KS C.13 (JavaP)

JavaP

MontiCore-Grammatik

```
package mc.javap;

/**
```

```
@version 1.0a
*/
grammar JavaP extends mc.umlp.common.Common {

  /** ASTStatement lists the different types of statements
  */
  interface Statement =
    BlockStatement
    | (VariableDeclarationStatement)=> VariableDeclarationStatement
    | AssignmentStatement
    | ReturnStatement;

  /** ASTVariableDeclarationStatement is a variable declaration
      @attribute Type         Type of the variable
      @attribute Name         Name of the variable
      @attribute Initializer  Initial value of the variable
  */
  VariableDeclarationStatement implements Statement =
    Type:Type Name ("=" Initializer:Expression )? ";";

  /** ASTBlockStatement is a block statement
      @attribute Statement Statements the block consists of
  */
  BlockStatement implements Statement  =
    "{" (Statements:Statement)* "}";

  /** ASTReturnStatement is a return statement
      @attribute Expression  an optional return value
  */
  ReturnStatement implements Statement  =
    "return" (Expression:Expression)? ";";

  /** ASTAssignmentStatement is a variable assignment
      @attribute Name  name of a variable
      @attribute Name  value to be assigned
  */
  AssignmentStatement implements Statement =
    Name "=" Expression ";";

  /** ASTExpression is the expression interface
  */
  interface Expression = LogicalOrExpression;

  ast InfixExpression =
    LeftOperand:Expression
    RightOperand:Expression;

  /** ASTLogicalOrExpression, logical OR (infix expression)
  */
  LogicalOrExpression:InfixExpression implements Expression returns Expression =
    ret=LogicalAndExpression //ret accords to the left operand
    (astscript{!(LeftOperand=ret;);}
    Operator:["||"] RightOperand:LogicalAndExpression)* ;

  /** ASTLogicalAndExpression, logical AND (infix expression)
  */
  LogicalAndExpression:InfixExpression implements Expression returns Expression =
    ret=EqualityExpression //ret accords to the left operand
    (astscript{!(LeftOperand=ret;);}
    Operator:["&&" ] RightOperand:EqualityExpression)*;

  /** ASTEqualityExpression, equal and not equal (infix expressions)
  */
  EqualityExpression:InfixExpression implements Expression returns Expression =
    ret=RelationalExpression //ret accords to the left operand
    (astscript{!(LeftOperand=ret;);}
    Operator:["!=" | "=="] RightOperand:RelationalExpression)*;

  /** ASTRelationalExpression, smaller or larger than (infix expressions)
  */
  RelationalExpression:InfixExpression implements Expression returns Expression =
    ret=AdditiveExpression //ret accords to the left operand
    (
      ( astscript{!(LeftOperand=ret;);}
      Operator:["<" | ">" ] RightOperand:AdditiveExpression
    )*
  );

  /** ASTAdditiveExpression, plus or minus (infix expressions)
  */
  AdditiveExpression:InfixExpression implements Expression returns Expression =
    ret=Primary //ret accords to the left operand
    (
    astscript{!(LeftOperand=ret;);}
    Operator:["+" | "-"] RightOperand:Primary
  )*;

  /** ASTPrimary, a literal or a name
  */
  Primary returns Expression  =
     ret = JLiteral
   | ret = JName;

  /** ASTJName, a name (from grammar Types) and also expression
  */
  JName implements Expression = QualifiedName;

  /** ASTJLiteral, a literal (from grammar Literals) and also expression
  */
  JLiteral implements Expression = Literal;

}
```

Definition AS C.14 (JavaPAS)

JavaPAS

Isabelle-Theorie (gen)

```
(*
Source: mc.javap.JavaP V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:19 CEST 2010
*)
theory JavaPAS
imports "$UMLP/gen/mc/umlp/common/CommonAS"
begin

datatype JLiteral =
    JLiteral Literal

datatype InfixExpressionOperator =
    InfixExpressionOperatorEQUALSEQUALS
  | InfixExpressionOperatorGT
  | InfixExpressionOperatorLT
  | InfixExpressionOperatorPLUS
  | InfixExpressionOperatorMINUS
  | InfixExpressionOperatorEXCLAMATIONMARKEQUALS
  | InfixExpressionOperatorPIPEPIPE
  | InfixExpressionOperatorANDAND

datatype Primary =
    Primary

datatype JName =
    JName QualifiedName

datatype InfixExpression =
    InfixExpression Expression Expression InfixExpressionOperator
and
Expression =
    ExpressionInfixExpression InfixExpression
  | ExpressionPrimary Primary
  | ExpressionJName JName
  | ExpressionJLiteral JLiteral

datatype VariableDeclarationStatement =
    VariableDeclarationStatement Type Name "Expression option"

datatype ReturnStatement =
    ReturnStatement "Expression option"

datatype AssignmentStatement =
    AssignmentStatement Name Expression

datatype Statement =
    StatementVariableDeclarationStatement VariableDeclarationStatement
  | StatementBlockStatement BlockStatement
  | StatementReturnStatement ReturnStatement
  | StatementAssignmentStatement AssignmentStatement
and
BlockStatement =
    BlockStatement "Statement list"

end
```

Definition KB C.15 (JavaPCC-base)

JavaPCC-base

Isabelle-Theorie

```
theory JavaPCC-base
imports "$UMLP/gen/mc/javap/JavaPAS"
        "$UMLP/gen/mc/types/syntax/TypesCC"
begin

fun wellformed-Expression :: "Expression ⇒ bool"
where
 "wellformed-Expression (ExpressionInfixExpression (InfixExpression l r opr)) =
    (wellformed-Expression l ∧ wellformed-Expression r)"
|"wellformed-Expression (ExpressionJName a) = True"
|"wellformed-Expression (ExpressionJLiteral (JLiteral a)) =
      wellformed-RestrictFloatTypes a"

fun wellformed-AssignmentStatement :: "AssignmentStatement ⇒ bool"
where
 "wellformed-AssignmentStatement (AssignmentStatement n expr)
```

```
    = wellformed-Expression expr"

fun wellformed-VariableDeclarationStatement
          :: "VariableDeclarationStatement ⇒ bool"
where
  "wellformed-VariableDeclarationStatement
    (VariableDeclarationStatement tp ide expr) = (
     (if (expr ≠ None) then (wellformed-Expression (the expr))
      else True)
      ∧ wellformed-RestrictTypes tp)"

fun wellformed-ReturnStatement :: "ReturnStatement ⇒ bool"
where
  "wellformed-ReturnStatement (ReturnStatement (Some expr)) =
      wellformed-Expression expr"
| "wellformed-ReturnStatement (ReturnStatement None) = True"

fun wellformed-Statement :: "Statement ⇒ bool"
and wellformed-BlockStatement :: "BlockStatement ⇒ bool"
where
 "wellformed-Statement
   (StatementVariableDeclarationStatement s) =
     wellformed-VariableDeclarationStatement s"
|"wellformed-Statement
   (StatementBlockStatement b) = wellformed-BlockStatement b"
|"wellformed-Statement
    (StatementAssignmentStatement e) = wellformed-AssignmentStatement e"
|"wellformed-Statement (StatementReturnStatement r) =
    wellformed-ReturnStatement r"
|"wellformed-BlockStatement (BlockStatement []) = True"
|"wellformed-BlockStatement (BlockStatement (x#xs)) = (
     wellformed-Statement x ∧ xs = [])"

end
```

Definition Variabilität C.16 (Spracheinschränkungen JavaP)

JavaPConstr

Feature-Diagramm

```
package mc.javap.syntax;

featurediagram JavaPConstr {

 constraints {
   JavaPConstr -> Literals.LiteralsConstr.RestrictFloatTypes;
 }
}
```

C.2.2 OCL/P

Definition KS C.17 (OCL)

OCL

MontiCore-Grammatik

```
package mc.umlp.ocl;

/**
@version 1.0a
```

```
*/
grammar OCL extends mc.umlp.common.Common {

  /** ASTOCLInvariant is an OCL invariant
      @attribute ContextDefinition  the context of the invariant
      @attribute name               an optional name
      @attribute OCLExpression      the invariant
  */
  OCLInvariant =
    "context" contextDefinitions:OCLContextDefinition
         ("," contextDefinitions:OCLContextDefinition)*
    "inv" Name? ":"
    OCLExpression ";";

  /** ASTOCLContextDefinition is the context of an invariant
      @attribute className the type of the context class
      @attribute name the name of the context instance
  */
  OCLContextDefinition =
    className:Type Name?;

  /** ASTOCLExpression is the OCL epxression interface
  */
  interface OCLExpression =
    OCLEquivalentExpr;

  ast OCLInfixExpression =
    leftHand:OCLExpression
    rightHand:OCLExpression;

  /** ASTOCLEquivalentExpr is the equivalence expression (infix)
  */
  OCLEquivalentExpr:OCLInfixExpression returns OCLExpression =
    (
    ret=OCLImpliesExpr
    (astscript{!(leftHand=ret;);}
    Operator:["<=>"] rightHand:OCLImpliesExpr)*
    );

  /** ASTOCLImpliesExpr is the implication expression (infix)
  */
  OCLImpliesExpr:OCLInfixExpression returns OCLExpression =
    ret=OCLDoubleLogicalORExpr
    (astscript{!(leftHand=ret;);}
    Operator:["implies"] rightHand:OCLDoubleLogicalORExpr)*;

  /** ASTOCLDoubleLogicalORExpr is the OR expression (infix)
  */
  OCLDoubleLogicalORExpr:OCLInfixExpression returns OCLExpression =
    ret=OCLDoubleLogicalANDExpr
    (astscript{!(leftHand=ret;);}
    Operator:["||"] rightHand:OCLDoubleLogicalANDExpr)*;

  /** ASTOCLDoubleLogicalANDExpr is the AND expression (infix)
  */
  OCLDoubleLogicalANDExpr:OCLInfixExpression returns OCLExpression =
    ret=OCLRelationalExpr
    (astscript{!(leftHand=ret;);}
    Operator:["&&"] rightHand:OCLRelationalExpr)*;

  /** ASTOCLRelationalExpr is the equals or not equals expression (infix)
  */
  OCLRelationalExpr:OCLInfixExpression returns OCLExpression =
    ret=OCLCompareExpr
    (astscript{!(leftHand=ret;);}
    Operator:["==" | "!="] rightHand:OCLCompareExpr)*;

  /** ASTOCLCompareExpr is the smaller or greater than expression (infix)
  */
  OCLCompareExpr:OCLInfixExpression returns OCLExpression =
    ret=OCLBinaryPlusMinusExpr
    (astscript{!(leftHand=ret;);}
    Operator:["<" | ">" ] rightHand:OCLBinaryPlusMinusExpr)*;

  /** ASTOCLBinaryPlusMinusExpr is the plus or minus expression (infix)
  */
  OCLBinaryPlusMinusExpr:OCLInfixExpression returns OCLExpression =
    ret=OCLPrimary
    (astscript{!(leftHand=ret;);}
    Operator:["+" | "-"] rightHand:OCLPrimary)*;

  /** ASTOCLPrimary is a method invocation, a literal or a name
  */
  OCLPrimary returns OCLExpression  =
    (OCLMethodInvocation) => (ret=OCLMethodInvocation)
    | ret=OCLLiteral
    | ret=OCLName;

  /** ASTOCLName is a name (from grammar Types)
  */
  OCLName implements OCLExpression = value:QualifiedName;

  /** ASTOCLLiteral is a literal (from grammar Literals)
  */
  OCLLiteral implements OCLExpression = value:Literal;

  /** ASTOCLMethodInvocation is a (qualified) method invocation
  */
  OCLMethodInvocation implements OCLExpression =
    Name
    (options{greedy=true;}: "." Name)*
    "(" (Arguments: OCLExpression ("," Arguments: OCLExpression)*)? ")";

}
```

Definition AS C.18 (OCLAS)

OCLAS

Isabelle-Theorie (gen)

```
(*
Source: mc.umlp.ocl.OCL V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:25 CEST 2010
*)
theory OCLAS
imports "$UMLP/gen/mc/umlp/common/CommonAS"
begin

datatype OCLLiteral =
    OCLLiteral Literal

datatype OCLInfixExpressionOperator =
    OCLInfixExpressionOperatorEQUALSEQUALS
```

```
  | OCLInfixExpressionOperatorGT
  | OCLInfixExpressionOperatorLT
  | OCLInfixExpressionOperatorPLUS
  | OCLInfixExpressionOperatorIMPLIES
  | OCLInfixExpressionOperatorMINUS
  | OCLInfixExpressionOperatorLTEQUALSGT
  | OCLInfixExpressionOperatorEXCLAMATIONMARKEQUALS
  | OCLInfixExpressionOperatorPIPEPIPE
  | OCLInfixExpressionOperatorANDAND

datatype OCLContextDefinition =
    OCLContextDefinition Type "Name option"

datatype OCLPrimary =
    OCLPrimary

datatype OCLName =
    OCLName QualifiedName

datatype OCLMethodInvocation =
    OCLMethodInvocation "Name list" "OCLExpression list"
and
OCLInfixExpression =
    OCLInfixExpression OCLExpression OCLExpression
      OCLInfixExpressionOperator
and
OCLExpression =
    OCLExpressionOCLInfixExpression OCLInfixExpression
  | OCLExpressionOCLPrimary OCLPrimary
  | OCLExpressionOCLName OCLName
  | OCLExpressionOCLLiteral OCLLiteral
  | OCLExpressionOCLMethodInvocation OCLMethodInvocation

datatype OCLInvariant =
    OCLInvariant "OCLContextDefinition list" "Name option" OCLExpression

end
```

Definition KB C.19 (OCLCC-base)

OCLCC-base

Isabelle-Theorie

```
theory OCLCC-base
imports "$UMLP/gen/mc/umlp/ocl/OCLAS"
        "$UMLP/gen/mc/types/syntax/TypesCC"
begin

fun wellformed-OCLExpression :: "OCLExpression ⇒ bool"
where
  "wellformed-OCLExpression
    (OCLExpressionOCLInfixExpression (OCLInfixExpression l r opr)) =
       (wellformed-OCLExpression l ∧ wellformed-OCLExpression r)"
 |"wellformed-OCLExpression (OCLExpressionOCLName n) = True"
 |"wellformed-OCLExpression (OCLExpressionOCLLiteral (OCLLiteral l)) =
    wellformed-RestrictFloatTypes l"
 |"wellformed-OCLExpression
    (OCLExpressionOCLMethodInvocation (OCLMethodInvocation idl exprl)) =
       (∀ expr ∈ set exprl . wellformed-OCLExpression expr)"

fun wellformed-OCLContextDefinition :: "OCLContextDefinition ⇒ bool"
where
```

```
 "wellformed-OCLContextDefinition (OCLContextDefinition tp idOpt) =
      wellformed-RestrictTypes tp"

fun wellformed-OCLInvariant :: "OCLInvariant ⇒ bool"
where
 "wellformed-OCLInvariant (OCLInvariant contextDefs idOpt expr) = (
    (∀ cd ∈ set contextDefs . wellformed-OCLContextDefinition cd) ∧
     wellformed-OCLExpression expr)"

end
```

Definition Variabilität C.20 (Spracheinschränkungen OCL)

OCLConstr

Feature-Diagramm

```
package mc.umlp.ocl.syntax;

featurediagram OCLConstr {

 constraints {
   OCLConstr -> Literals.LiteralsConstr.RestrictFloatTypes;
 }
}
```

C.3 Klassendiagramme

Definition KS C.21 (CD)

CD

MontiCore-Grammatik

```
package mc.umlp.cd;

/**
@version 1.0a
*/
grammar CD extends mc.umlp.common.Common {

  options {
    nostring noident nows
    compilationunit CDDefinition
  }

  /*========================================================================*/
  /*================== INTERFACES AND EXTERNAL SYMBOLS =====================*/
  /*========================================================================*/

  ast CDClass astimplements /mc.umlp.cd.IType;

  ast CDInterface astimplements /mc.umlp.cd.IType;

  ast CDEnum astimplements /mc.umlp.cd.IType;

  external Value;
  external Body;

  /** ASTCDElement represents all Elements of a UML Classdiagram */
  interface CDElement;
```

```

external InvariantExpression;

/** ASTInvariant represents an Invariant in the UML/P
    @attribute kind                Kind/Language of the Invariant
    @attribute invariantExpression Condition of the Invariant
*/
Invariant =
  (kind:Name":")?
  "[" InvariantExpression(parameter kind) "]";


/*=======================================================================*/
/*============================== GRAMMAR ================================*/
/*=======================================================================*/

/** ASTCDDefinition represents a UML Classdiagram
    @attribute completeness Optional Comleteness of this Classdiagramm
    @attribute stereotype   Optional Stereotype
    @attribute name         Name of this Classdiagram
    @attribute cDElements   List of the content of this Classdiagram
                            (Classes, Interfaces, and Associations)
    @attribute invaritants  List of Invaritants of this Classdiagram
*/
CDDefinition =
  Completeness?
  Stereotype?
  "classdiagram" Name
  "{"
  (cDElements:CDElement | (invariants:Invariant ";"))*
  "}";

/** ASTCDClass represents a Class in a UML Classdiagram
    @attribute completeness   Optional Comleteness of this Class
    @attribute modifier       Modifier of this Class
    @attribute name           Name of this Class
    @attribute typeParameters Generic type parameters of this Class
    @attribute superclasses   List of Superclasses of this Class
    @attribute interfaces     List of Interfaces implemented by this Class
    @attribute cDConstructors List of Constructors of this Class
    @attribute cDMethods      List of Methods of this Class
    @attribute cDAttributes   List of Attributes of this Class
*/
CDClass implements (Completeness? Modifier "class")=>CDElement =
    Completeness?
    Modifier
    "class" Name
    (options {greedy=true;}: TypeParameters)?
    ("extends" superclasses:ReferenceType
              ("," superclasses:ReferenceType)*)?
    ("implements" interfaces:ReferenceType
              ("," interfaces:ReferenceType)*)?
    (
      ("{" (
        (Modifier Type Name ("=" | ";"))
        => cDAttributes:CDAttribute
        |
        (Modifier (options {greedy=true;}: TypeParameters)? Name "(")
        => cDConstructors:CDConstructor
        |
        cDMethods:CDMethod
     )* "}")
     |
     ";"
      );

/** ASTCDInterface represents a Interface in a UML Classdiagram
    @attribute completeness   Optional Comleteness of this Interface
    @attribute modifier       Modifier of this Interface
    @attribute name           Name of this Interface
    @attribute typeParameters Generic type parameters of this Interface
    @attribute interfaces     List of Interfaces extended by this Interface
    @attribute constructors   List of Constructors of this Interface
    @attribute cDMethods      List of Methods of this Interface
    @attribute cDAttributes   List of Attributes of this Interface
*/
CDInterface implements (Completeness? Modifier "interface")=>CDElement =
  Completeness?
  Modifier
  "interface" Name
  (options {greedy=true;}: TypeParameters)?
  ("extends" interfaces:ReferenceType
            ("," interfaces:ReferenceType)*)?
  (
    ("{" (
      (Modifier Type Name ("=" | ";"))
        => cDAttributes:CDAttribute
       |
       cDMethods:CDMethod
      )* "}")
      |
       ";"
  );

/** ASTCDEnum represents an Enumeration (enum type) in a UML Classdiagram
    @attribute completeness    Optional Comleteness of this Enum
    @attribute modifier        Modifier of this Enum
    @attribute name            Name of this Enum
    @attribute interfaces      List of Interfaces implemented by this Enum
    @attribute cDEnumConstants List of the Enum Constants
    @attribute cDConstructors  List of Constructors of this Enum
    @attribute cDMethods       List of Methods of this Enum
    @attribute cDAttributes    List of Attributes of this Enum
*/
CDEnum implements (Completeness? Modifier "enum")=>CDElement =
    Completeness?
    Modifier
    "enum" Name
    ("implements" interfaces:ReferenceType
              ("," interfaces:ReferenceType)*)?
    (
      ("{"
        cDEnumConstants:CDEnumConstant
        ("," cDEnumConstants:CDEnumConstant)* ";"
        (
          (Modifier Type Name ("=" | ";"))
          => cDAttributes:CDAttribute
          |
          (Modifier ("<" TypeVariableDeclaration
          ("," TypeVariableDeclaration)* (">" | ">>" | ">>>")?)?
          Name "(")
          => cDConstructors:CDConstructor
          |
          cDMethods:CDMethod
        )*
      "}")
      |
      ";"
    );

/** ASTCDEnumConstant represents a Constant of an Enumeration (enum type)
    in a UML Classdiagram
    @attribute name             Name of this Constant
    @attribute cDEnumParameters List of optional parameters of this Constant
*/
CDEnumConstant =
  Name ("("
      cDEnumParameters:CDEnumParameter
      ("," cDEnumParameters:CDEnumParameter)*
  ")")?;

/** ASTCDEnumParameter represents a Parameter of a Enumeration Constant
    @attribute Value Value of this Parameter
*/
CDEnumParameter =
    Value;

/** ASTCDMethod represents a Method of a Class or Interface
    @attribute modifier       Modifier of this Method
    @attribute typeParameters Generic type parameters of this Method
    @attribute returnType     Return-Type of the return value of this Method
    @attribute name           Name of this Method
    @attribute cDParameters   List of Parameters of this Method
    @attribute exceptions     List of Exceptions thrown by this Method
    @attribute body           Body of this Method
*/
CDMethod =
  Modifier
  (options {greedy=true;}: TypeParameters)?
  ReturnType
  Name "(" (
    cDParameters:CDParameter
    (options {greedy=true;}: "," cDParameters:CDParameter)*
  )? ")"
  (("throws")=> "throws"
    exceptions:QualifiedName
       (options {greedy=true;}: "," exceptions:QualifiedName)*)?
  (("{")=>Body | ";");

/** ASTCDConstructor represents a Constructor of a Class or Interface
    @attribute modifier       Modifier of this Constructor
    @attribute typeParameters Generic type parameters of this Constructor
    @attribute name           Name of this Constructor
    @attribute cDParameters   List of Parameters of this Constructor
    @attribute exceptions     List of Exceptions thrown by this Constructor
    @attribute body           Body of this Constructor
*/
CDConstructor =
    Modifier
    (options {greedy=true;}: TypeParameters)?
    Name
    "(" (cDParameters:CDParameter ("," cDParameters:CDParameter)*)? ")"
    (("throws")=> "throws"
      exceptions:QualifiedName
         (options {greedy=true;}: "," exceptions:QualifiedName)*)?
    (("{")=> Body | ";");
```

```

/** ASTCDParameter represents a Parameter of a Constructor or Method
   @attribute type Type of this Parameter
   @attribute name Name of this Parameter
*/
CDParameter =
  Type Name;

/** ASTCDAttribute represents an Attribute of a Class or Interface
    @attribute modifier Modifier of this Attribute
    @attribute type     Type of this Attribute
    @attribute name     Name of this Attribute
     @attribute value    Value of this Attribute
*/
CDAttribute =
  Modifier
  Type
  Name
  ("=" Value)? ";";

/** ASTCDAssociation represents a Association between Classes or Interfaces
    @attribute stereotype       Optional Stereotype
    @attribute type             Type of the Association (Association,
                                Aggregation, or Composition)
    @attribute derived          True if this is a derived Association
    @attribute name             Name of this Association
    @attribute leftStereotype   Optional left side Stereotype
    @attribute leftCardinality  Cardinality of the left side of this
                                Association
    @attribute leftReference    Name of the Class or Interface on the left
                                side of this Association
    @attribute leftQualifier    Qualifier of the left side of this
                                Association
    @attribute leftRole         Role of the Class or Interface on the left
                                side of this Association
    @attribute arrow            Navigatable direction between left and right
                                side of this Association (lefttoright:"->",
                                righttoleft:"<-", bidirectional:"<->",
                                simple:"--")
    @attribute rightRole        Role of the Class or Interface on the right
                                side of this Association
    @attribute rightQualifier   Qualifier of the right side of this
                                Association
    @attribute rightReference   Name of the Class or Interface on the right
                                side of this Association
    @attribute rightCardinality Cardinality of the right side of this
                                Association
    @attribute rightStereotype  Optional right side Stereotype
*/
CDAssociation implements
(Stereotype? ("association"|"aggregation"|"composition"))=>CDElement =
  Stereotype?
  type:[Association:"association"
      | Aggregation:"aggregation"
      | Composition:"composition"]
  (Derived:[DERIVED:"/"])?
  //predicate necessary to prevent clash with LeftReference
  ((Name Stereotype? Cardinality? QualifiedName)=> Name |)
  leftStereotype:Stereotype?
  leftCardinality:Cardinality?
  leftReference:QualifiedName
  leftQualifier:Qualifier?
```

```
    ("(" leftRole:Name ")")?
    arrow:[lefttoright:"->"
        | righttoleft:"<-"
        | bidirectional:"<->"
        | simple:"--"]
    ("(" rightRole:Name ")")?
    rightQualifier:Qualifier?
    rightReference:QualifiedName
    rightCardinality:Cardinality?
    rightStereotype:Stereotype? ";";

}
```

Definition AS C.22 (CDAS)

CDAS

Isabelle-Theorie (gen)

```
(*
Source: mc.umlp.cd.CD V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:22 CEST 2010
*)
theory CDAS
imports "$UMLP/def/mc/umlp/cd/ExternalCDAS" "$UMLP/gen/mc/umlp/common/CommonAS"
begin

datatype CDEnumParameter =
    CDEnumParameter Value

datatype CDEnumConstant =
    CDEnumConstant Name "CDEnumParameter list"

datatype CDParameter =
    CDParameter Type Name

datatype CDMethod =
    CDMethod Modifier "TypeParameters option" ReturnType Name
      "CDParameter list" "QualifiedName list" "Body option"

datatype CDConstructor =
    CDConstructor Modifier "TypeParameters option" Name "CDParameter list"
      "QualifiedName list" "Body option"

datatype CDAttribute =
    CDAttribute Modifier Type Name "Value option"

datatype CDEnum =
    CDEnum "Completeness option" Modifier Name "ReferenceType list"
      "CDEnumConstant list" "CDMethod list" "CDConstructor list"
      "CDAttribute list"

datatype CDClass =
    CDClass "Completeness option" Modifier Name "TypeParameters option"
      "ReferenceType list" "ReferenceType list" "CDMethod list"
      "CDConstructor list" "CDAttribute list"

datatype CDInterface =
    CDInterface "Completeness option" Modifier Name "TypeParameters option"
      "ReferenceType list" "CDMethod list" "CDAttribute list"

datatype CDAssociationtype =
    CDAssociationtypeAGGREGATION
  | CDAssociationtypeCOMPOSITION
  | CDAssociationtypeASSOCIATION

datatype CDAssociationDerived =
    CDAssociationDerivedDERIVED

datatype CDAssociationarrow =
    CDAssociationarrowRIGHTTOLEFT
  | CDAssociationarrowBIDIRECTIONAL
  | CDAssociationarrowSIMPLE
  | CDAssociationarrowLEFTTORIGHT

datatype CDAssociation =
    CDAssociation "Stereotype option" CDAssociationtype
      "CDAssociationDerived option" "Name option"
      "Stereotype option" "Cardinality option" QualifiedName
      "Qualifier option" "Name option" CDAssociationarrow
      "Name option" "Qualifier option" QualifiedName
      "Cardinality option" "Stereotype option"

datatype CDElement =
    CDElementCDClass CDClass
  | CDElementCDInterface CDInterface
  | CDElementCDEnum CDEnum
  | CDElementCDAssociation CDAssociation

datatype Invariant =
    Invariant "Name option" InvariantExpression

datatype CDDefinition =
    CDDefinition "Completeness option" "Stereotype option" Name
      "Invariant list" "CDElement list"

datatype MCCompilationUnit =
    MCCompilationUnit "Name list" "Name list" "STRING option" CDDefinition

end
```

Definition KB C.23 (CDCC-base)

CDCC-base

Isabelle-Theorie

```
theory CDCC-base
imports "$UMLP/gen/mc/umlp/cd/CDAS"
        "$UMLP/gen/mc/umlp/common/CommonCC"
        "$UMLP/def/mc/umlp/cd/syntax/ExternalCDCC"
begin

fun noBody :: "CDMethod ⇒ bool"
where
 "noBody (CDMethod - - - - - - bodyOpt) =
    (bodyOpt = None)"

fun wellformed-InterfaceModifier :: "Modifier ⇒ bool"
where
 "wellformed-InterfaceModifier (Modifier stereoOpt modivals) = (
    Modifier.wellformed-Modifier (Modifier stereoOpt modivals) ∧
    (getModifiers ModifierValSTATIC modivals) = [] ∧
    (getModifiers ModifierValFINAL  modivals) = []
 )"
```

```

fun wellformed-CDInterface :: "CDInterface ⇒ bool"
where
 "wellformed-CDInterface
   (CDInterface complOpt modi n typParam refL1 mths attrs) = (
     (wellformed-InterfaceModifier modi ∧
      (∀ m ∈ set mths . noBody m))
 )"

fun wellformed-CDElement :: "CDElement ⇒ bool"
where
 "wellformed-CDElement (CDElementCDInterface x) = wellformed-CDInterface x"

fun exprOf :: "Invariant ⇒ InvariantExpression"
where
 "exprOf (Invariant ide expr) = expr"

fun wellformed-base :: "CDDefinition ⇒ bool"
where
 "wellformed-base (CDDefinition complOpt stereoOpt na invars elems) = (
    (∀ i ∈ set invars . wellformed-InvariantExpression (exprOf i)) ∧
    (∀ e ∈ set elems . wellformed-CDElement e)
  )"

end
```

Variante KB C.24 (Ausschluss von Enumerationen, Typparametern etc.)

RestrictCD

Isabelle-Theorie

```
theory RestrictCD
imports "$UMLP/gen/mc/umlp/cd/CDAS"
        "$UMLP/gen/mc/umlp/common/CommonCC"
begin

fun toType :: "ReturnType ⇒ Type option"
where
  "toType (ReturnTypeType t) = Some t"
| "toType (ReturnTypeVoidType t) = None"

fun restrict-TypeParameters :: "TypeParameters option ⇒ bool"
where
  "restrict-TypeParameters None = True"
| "restrict-TypeParameters (Some (TypeParameters x)) = (x = [])"

fun restrict-CDParameter :: "CDParameter ⇒ bool"
where
 "restrict-CDParameter (CDParameter tp n) =
      wellformed-RestrictTypes tp"

fun restrict-CDMethod :: "CDMethod ⇒ bool"
where
 "restrict-CDMethod (CDMethod modi tpParamOpt retT n paramL ex bodyOpt) =
  ( restrict-TypeParameters tpParamOpt ∧
    (if (toType retT = None) then True
     else wellformed-RestrictTypes (the (toType retT))) ∧
    (∀ p ∈ set paramL . restrict-CDParameter p) ∧
    ex = []
  )"

fun restrict-CDConstructor :: "CDConstructor ⇒ bool"
```

```
where
 "restrict-CDConstructor (CDConstructor modi tpParamOpt n paramL ex bodyOpt) =
   (restrict-TypeParameters tpParamOpt ∧
    (∀ p ∈ set paramL . restrict-CDParameter p) ∧
    ex = []
 )"

fun restrict-CDAttribute :: "CDAttribute ⇒ bool"
where
 "restrict-CDAttribute (CDAttribute modi tp n valOpt) =
    (wellformed-RestrictTypes tp)"

fun restrict-CDClass :: "CDClass ⇒ bool"
where
 "restrict-CDClass
   (CDClass complOpt modi n typParam refL1 refL2 mths constrs attrs) = (
     restrict-TypeParameters typParam ∧
     (∀ t1 ∈ set refL1 . wellformed-RestrictTypes (TypeReferenceType t1)) ∧
     (∀ t1 ∈ set refL2 . wellformed-RestrictTypes (TypeReferenceType t1)) ∧
     (∀ m ∈  set mths  . restrict-CDMethod  m) ∧
     (∀ c ∈ set constrs . restrict-CDConstructor c) ∧
     (∀ a ∈ set attrs . restrict-CDAttribute a)
   )"

fun restrict-CDInterface :: "CDInterface ⇒ bool"
where
 "restrict-CDInterface
   (CDInterface complOpt modi n typParam refL1 mths attrs) = (
     restrict-TypeParameters typParam ∧
     (∀ t1 ∈ set refL1 . wellformed-RestrictTypes (TypeReferenceType t1)) ∧
     (∀ m ∈  set mths  . restrict-CDMethod  m) ∧
     (∀ a ∈ set attrs . restrict-CDAttribute a)
 )"

fun restrict-CDElement :: "CDElement ⇒ bool"
where
  "restrict-CDElement (CDElementCDClass x) = restrict-CDClass x"
| "restrict-CDElement (CDElementCDInterface x) = restrict-CDInterface x"
| "restrict-CDElement (CDElementCDEnum x) = False"

fun wellformed-RestrictCD :: "CDDefinition ⇒ bool"
where
 "wellformed-RestrictCD (CDDefinition complOpt stereoOpt na invars elems) = (
    (∀ e ∈ set elems . restrict-CDElement e)
  )"

end
```

Variante KB C.25 (Keine Mehrfachvererbung)

CCSingleInheritance

Isabelle-Theorie

```
theory CCSingleInheritance
imports "$UMLP/gen/mc/umlp/cd/CDAS"
begin

fun singleInheritance :: "CDClass ⇒ bool"
where
 "singleInheritance (CDClass - - - - refL1 - - - -) = (length refL1 <= 1)"

fun getCDClasses :: "CDElement list ⇒ CDClass list"
```

```
where
  "getCDClasses ((CDElementCDClass x)#xs) = (x#(getCDClasses xs))"
| "getCDClasses ((CDElementCDInterface x)#xs) = getCDClasses xs"
| "getCDClasses ((CDElementCDEnum x)#xs) = getCDClasses xs"
| "getCDClasses ((CDElementCDAssociation x)#xs) = getCDClasses xs"

fun wellformed-CCSingleInheritance :: "CDDefinition ⇒ bool"
where
 "wellformed-CCSingleInheritance
   (CDDefinition complOpt stereoOpt na invars elems) =
    (∀ c ∈ set (getCDClasses elems) . singleInheritance c)"

end
```

Definition Variabilität C.26 (Syntax CD)

CDSyntax

Feature-Diagramm

```
package mc.umlp.cd.syntax;

featurediagram CDSyntax {

  CDSyntax = <<lparam>> LangParam? & <<constr>> CDConstr? & <<stereos>> Stereos?;

  LangParam = InvariantExpression? & Value? & Body?;

  InvariantExpression = OCLInvariant;

  Value = JavaExpression;

  Body = JavaBlockStatement;

  CDConstr = vInherit? & RestrictCD?;

  vInherit = CCSingleInheritance;

  Stereos = SingletonClass;

  constraints {
    Stereos.SingletonClass -> CDSemantics.vStereotypes.MapSingletonClass;
    CDConstr -> TypesConstr.RestrictTypes;
  }
}
```

C.4 Objektdiagramme

Definition KS C.27 (OD)

OD

MontiCore-Grammatik

```
package mc.umlp.od;

/**
@version 1.0a
*/
grammar OD extends mc.umlp.common.Common {

```

```
options {
  nostring noident nows
  compilationunit ODDefinition
}

/*========================================================================*/
/*================== INTERFACES AND EXTERNAL SYMBOLS =====================*/
/*========================================================================*/

external Value;

/** ASTODElement represents all Elements of a UML Objectdiagram
*/
interface ODElement;

external InvariantExpression;

/** ASTInvariant represents an Invariant in the UML/P
    @attribute kind                Kind/Language of the Invariant
    @attribute invariantExpression Condition of the Invariant
  */
Invariant =
  (kind:Name ":")?
  "[" InvariantExpression(parameter kind) "]";

/*========================================================================*/
/*============================== GRAMMAR =================================*/
/*========================================================================*/

/** ASTODDefinition represents a UML Objectdiagram
    @attribute completeness Optional Comleteness of this Objectdiagramm
    @attribute stereotype   Optional Stereotype
    @attribute name         Name of this Objectdiagram
    @attribute oDElements   List of the content of this Objectdiagram
                            (Objects and Links)
    @attribute invaritants  List of Invaritants of this Objectdiagram
*/
ODDefinition =
  Completeness?
  Stereotype?
  "objectdiagram" Name
  "{"
  (
    ((Name ":" "[") | "[" )=> (invariants:Invariant ";")
    |
    oDElements:ODElement
  )*
  "}";

/** ASTODObject represents an Object in a UML Objectdiagram
    @attribute completeness Optional Comleteness of this Object
    @attribute modifier     Modifier of this Object
    @attribute name         Optional Name of this Object
    @attribute type         Optional Type of this Object
    @attribute oDAttributes List of Attributes of this Object
*/
ODObject implements
(Completeness? Modifier
(Name (":" ReferenceType)? | (":" ReferenceType))
("{"|";"))=>ODElement =
  Completeness?
  Modifier
```

```
    (Name (":" type:ReferenceType)? | (":" type:ReferenceType))
    (
      ("{" oDAttributes:ODAttribute* "}")
      |
      ";"
    );

  /** ASTODAttribute represents an Attribute of an Object
      @attribute modifier Modifier of this Attribute
      @attribute type     Type of this Attribute
      @attribute name     Name of this Attribute
      @attribute value    Value of this Attribute
  */
  ODAttribute =
    Modifier
    Type?
    Name
    ("=" Value)? ";";

  /** ASTODLink represents a Link between Objects
      @attribute stereotype      Optional Stereotype
      @attribute type            Type of the Link (Link, Aggregation, or
                                 Composition)
      @attribute derived         True if this is a derived Link
      @attribute associationName Name of the Association of this Link
      @attribute leftReferences  List of References of the Objects on the left
                                 side of this Link
      @attribute leftQualifier   Qualifier of the left side of this Link
      @attribute leftRole        Role of the Objects on the left side of this
                                 Link
      @attribute arrow           Navigatable direction between left and right
                                 side of this Link (lefttoright:"->",
                                 righttoleft:"<-", bidirectional:"<->",
                                 simple:"--")
      @attribute rightRole       Role of the Objects on the right side of this
                                 Link
      @attribute rightQualifier  Qualifier of the right side of this Link
      @attribute rightReferences List of References of the Objects on the
                                 right side of this Link
  */
  ODLink implements
  (Stereotype? ("link"|"aggregation"|"composition"))=>ODElement =
    Stereotype?
    type:[link:"link"
        | aggregation:"aggregation"
        | composition:"composition"]
    (Derived:[DERIVED:"/"])?
    associationName:Name?
    leftReferences:Name ("," leftReferences:Name)*
    ("[" leftQualifier:Value "]")?
    ("(" leftRole:Name ")")?
    arrow:[lefttoright:"->"
         | righttoleft:"<-"
         | bidirectional:"<->"
         | simple:"--"]
    ("(" rightRole:Name ")")?
    ("[" rightQualifier:Value "]")?
    rightReferences:Name ("," rightReferences:Name)*
    ";";

```

```
}
```

Die Grammatik aus Definition C.27 wurde im Vergleich zur Version 1.0 [Sch09] wie folgt angepasst:

- Links sind die „Instanzen von Assoziationen“. Daher wurde für Links der Typ der Qualifikatoren von `Qualifier` (was einen Typ beschreibt) auf `Value` geändert. Der Wert ist der Qualifikator, der das Ende des Links identifiziert. Die Änderung behebt damit einen Fehler in der früheren Version.

Definition AS C.28 (ODAS)

ODAS

Isabelle-Theorie (gen)

```
(*
Source: mc.umlp.od.OD V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:26 CEST 2010
*)
theory ODAS
imports "$UMLP/def/mc/umlp/od/ExternalODAS" "$UMLP/gen/mc/umlp/common/CommonAS"
begin

datatype ODAttribute =
    ODAttribute Modifier "Type option" Name "Value option"

datatype ODObject =
    ODObject "Completeness option" Modifier "ReferenceType option"
      "Name option" "ODAttribute list"

datatype ODLinktype =
    ODLinktypeLINK
  | ODLinktypeAGGREGATION
  | ODLinktypeCOMPOSITION

datatype ODLinkDerived =
    ODLinkDerivedDERIVED

datatype ODLinkarrow =
    ODLinkarrowRIGHTTOLEFT
  | ODLinkarrowBIDIRECTIONAL
  | ODLinkarrowSIMPLE
  | ODLinkarrowLEFTTORIGHT

datatype ODLink =
    ODLink "Stereotype option" ODLinktype "ODLinkDerived option"
      "Name option" "Name list" "LeftQualifier option" "Name option"
      ODLinkarrow "Name option" "RightQualifier option" "Name list"

datatype ODElement =
    ODElementODObject ODObject
  | ODElementODLink ODLink

datatype Invariant =
    Invariant "Name option" InvariantExpression

datatype ODDefinition =
    ODDefinition "Completeness option" "Stereotype option" Name
```

```
      "ODElement list" "Invariant list"

datatype MCCompilationUnit =
    MCCompilationUnit "Name list" "Name list" "STRING option" ODDefinition

end
```

Definition Variabilität C.29 (Syntax OD)

ODSyntax

Feature-Diagramm

```
package mc.umlp.od.syntax;

featurediagram ODSyntax {

  ODSyntax =   <<lparam>> LangParam?;

  LangParam = InvariantExpression? & Value?;

  InvariantExpression = JavaExpression;

  Value = JavaExpression;

}
```

C.5 Statecharts

Definition KS C.30 (SC)

SC

MontiCore-Grammatik

```
package mc.umlp.sc;

/**
@version 1.0a
*/
grammar SC extends mc.umlp.common.Common {

  options {
    nostring noident nows
    compilationunit SCDefinition
  }

  /*=========================================================================*/
  /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
  /*=========================================================================*/
  external InvariantExpression;

  /** ASTInvariant represents an Invariant in the UML/P
      @attribute kind                Kind/Language of the Invariant
      @attribute invariantExpression Condition of the Invariant
  */
  Invariant =
    (kind:Name ":")?
    "[" InvariantExpression(parameter kind) "]";

  external Statements;
external Expression;

/** ASTSCElement represents all Elements of a UML Statechart diagram
*/
interface SCElement;

/** ASTSCEvent represents Events of Transitions in a UML Statechart diagram
*/
interface SCEvent;



/*========================================================================*/
/*============================== GRAMMAR =================================*/
/*========================================================================*/

/** ASTSCDefinition represents a UML Statechart diagram
    @attribute completeness Optional Comleteness of this Statechart diagram
    @attribute stereotype   Optional Stereotype
    @attribute name         Name of this Statechart diagram
    @attribute className    Optional name of the class modeled with this
                            Statechart diagram (if it differs from the name
                            of the Statechart diagram)
    @attribute sCMethod     Optional method for method Statecahrt diagrams
    @attribute sCElements   List of the content of this Statechart diagram
                            (States, Transitions, and Code)
    @attribute invaritants  List of Invaritants of this Statechart diagram
*/
SCDefinition =
  Completeness?
  Stereotype?
  "statechart" Name
  ((QualifiedName "(")=> SCMethod | className:ReferenceType)?
  "{"
    (
      ((Name ":") | "[" )=> (invariants:Invariant ";")
      |
      sCElements:SCElement
     )*
  "}";

/** ASTSCMethod represents the Method of a method Statechart diagram
    @attribute name         Name of this Method
    @attribute sCParameters Optional list of parameters
*/
SCMethod =
  QualifiedName "(" (
    sCParameters:SCParameter ("," sCParameters:SCParameter)*
  )? ")";

/** ASTSCParameter represents a Parameter of a Method
    @attribute type Type of this Parameter
    @attribute name Name of this Parameter
*/
SCParameter =
  Type Name;

/** ASTSCAction represents the general part of do-, entry-, and exit-Actions
    @attribute preCondition  Pre-Condition for this Action
    @attribute dtatements    Statements of this Action
    @attribute postCondition Post-Condition for this Action
*/
```

```
SCAction =
  (((Name ":") | "[")=> preCondition:Invariant)?
  (
    ";"
    |
    ("/" Statements
      (((Name ":") | "[")=> postCondition:Invariant ";")?)
  );

/** ASTSCDoAction represents a do-Action of States
    @attribute sCAction Action for this do-Action
*/
SCDoAction =
    "do" SCAction;

/** ASTSCEntryAction represents an entry-Action of States
    @attribute sCAction Action for this entry-Action
*/
SCEntryAction =
    "entry" SCAction;

/** ASTSCExitAction represents an exit-Action of States
    @attribute sCAction Action for this exit-Action
*/
SCExitAction =
  "exit" SCAction;

/** ASTSCModifier represents a Modifier for a State
    @attribute stereotype Optional Stereotpye
    @attribute initial    true if State is initial
    @attribute final      true if State is final
    @attribute local      true if State is local
*/
//own modifier needed as common-modifier causes nondeterminism
//because of derived "/"
SCModifier =
  Stereotype?
   ModifierVals*;

enum ModifierVals = "initial" |"final"|"local";

/** ASTSCState represents a State in a UML Statechart diagram
    @attribute completeness        Optional Comleteness of this State
    @attribute sCModifier          Modifier of this State
    @attribute name                Name of this State
    @attribute invariant           Invariant for this State
    @attribute sCEntryAction       entry-Action for this State
    @attribute sCDoAction          do-Action for this State
    @attribute sCExitAction        exit-Action for this State
    @attribute sCInternTransitions List of Intern Transitions for this State
    @attribute sCElements          List of Elements (States, Transitions,
                                   and Code) included in this State
                                   (hierarchical Statechart diagram)
*/
SCState implements (Completeness? SCModifier "state")=>SCElement =
  Completeness?
  SCModifier
  "state" Name
  (
    ("{"
      (((Name ":") | "[" )=> Invariant ";")?
      SCEntryAction?
      SCDoAction?
      SCExitAction?
      (
        (Stereotype? "->")=>
        sCInternTransitions:SCInternTransition
        |
        sCElements:SCElement
      )*
    "}")
   |
   ";"
  );

/** ASTSCInternTransition represents an Intern Transition of a State in a
    UML Statechart diagram
    @attribute stereotype       Optional Stereotype
    @attribute sCTransitionBody Body of this Transition
*/
SCInternTransition =
  Stereotype? "->" ":"? SCTransitionBody;

/** ASTSCTransition represents a Transition between two States in a
    UML Statechart diagram
    @attribute stereotype       Optional Stereotype
    @attribute source           Name of the source of this Transition
    @attribute target           Name of the target of this Transition
    @attribute sCTransitionBody Body of this Transition
*/
SCTransition implements (Stereotype? Name "->")=>SCElement =
  Stereotype?
  source:QualifiedName "->" target:QualifiedName
  (
    (":" SCTransitionBody)
    |
    ";"
  );

/** ASTSCTransitionBody represents the Body of a Transition in a
    UML Statechart diagram
    @attribute preCondition  Pre-Condition of this Transition Body
    @attribute sCEvent       Event for this Transition Body to take place
    @attribute statements    Actions of this Transition Body
    @attribute postCondition Post-Condition of this Transition Body
*/
SCTransitionBody =
  (((Name ":") | "[")=> preCondition:Invariant)?
  SCEvent?
  (
    ("/" Statements
      (((Name ":") | "[")=> postCondition:Invariant ";")?)
     |
    ";"
  );

/** ASTSCMethodOrExceptionCall represents a call of a method or exception
    of a Transition in a UML Statechart diagram
    @attribute name        Name of this method call
    @attribute sCArguments Optional Arguments of this method call
*/
SCMethodOrExceptionCall implements SCEvent =
  QualifiedName (("(")=> SCArguments)?;

  /** ASTSCReturnStatement represents a return statement of a Transition
      in a UML Statechart diagram
      @attribute incomplete True if return statement is
                            incomplete/underspecified
      @attribute expression Expression of this return statement
  */
  SCReturnStatement implements SCEvent =
    "return"
    (("(")=> "("
      (("...")=> incomplete:[INCOMPLETE:"..."]
      | Expression)
    ")")?;

  /** ASTSCArguments represents Arguments of an Event in a UML Statechart
      diagram
      @attribute incomplete  True if Arguments are incomplete (underspecified)
      @attribute expressions Specified Arguments as a list of Expressions
  */
  SCArguments =
    ("(" "...")=> ("(" incomplete:[INCOMPLETE:"..."] ")")
    | ("(" ")")=> ("(" ")")
    | ("(" expressions:Expression ("," expressions:Expression)* ")");

  /** ASTSCCode represents user added code to the Statechart diagram or to
      States
      @attribute statements The code added by the user
  */
  SCCode implements ("code")=>SCElement =
    "code" Statements;

}
```

Die Grammatik aus Definition C.30 wurde im Vergleich zur Version 1.0 [Sch09] wie folgt angepasst:

- Zur Vereinfachung der abstrakten Syntax wurde `association` als Metamodellierungskonzept für Zustände und Transitionen gelöscht. Wir benötigen keinen direkten Zugriff von Zuständen auf ein- oder ausgehende Transitionen oder umgekehrt, von Transitionen auf Quellzustand und Ziel.

Definition AS C.31 (SCAS)

SCAS

Isabelle-Theorie (gen)

```
(*
Source: mc.umlp.sc.SC V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:28 CEST 2010
*)
theory SCAS
imports "$UMLP/def/mc/umlp/sc/ExternalSCAS" "$UMLP/gen/mc/umlp/common/CommonAS"
begin

datatype Invariant =
    Invariant "Name option" InvariantExpression

datatype SCArgumentsincomplete =
    SCArgumentsincompleteINCOMPLETE
```

```

datatype SCArguments =
    SCArguments "Expressions list" "SCArgumentsincomplete option"

datatype SCMethodOrExceptionCall =
    SCMethodOrExceptionCall QualifiedName "SCArguments option"

datatype SCReturnStatementincomplete =
    SCReturnStatementincompleteINCOMPLETE

datatype SCReturnStatement =
    SCReturnStatement "Expression option"
      "SCReturnStatementincomplete option"

datatype SCEvent =
    SCEventSCMethodOrExceptionCall SCMethodOrExceptionCall
  | SCEventSCReturnStatement SCReturnStatement

datatype SCTransitionBody =
    SCTransitionBody "Invariant option" "SCEvent option"
      "Statements option" "Invariant option"

datatype ModifierVals =
    ModifierValsFINAL
  | ModifierValsLOCAL
  | ModifierValsINITIAL

datatype SCParameter =
    SCParameter Type Name

datatype SCMethod =
    SCMethod QualifiedName "SCParameter list"

datatype SCModifier =
    SCModifier "Stereotype option" "ModifierVals list"

datatype SCAction =
    SCAction "Invariant option" "Statements option" "Invariant option"

datatype SCEntryAction =
    SCEntryAction SCAction

datatype SCDoAction =
    SCDoAction SCAction

datatype SCExitAction =
    SCExitAction SCAction

datatype SCInternTransition =
    SCInternTransition "Stereotype option" SCTransitionBody

datatype SCTransition =
    SCTransition "Stereotype option" QualifiedName QualifiedName
      "SCTransitionBody option"

datatype SCCode =
    SCCode Statements

datatype SCState =
    SCState "Completeness option" SCModifier Name "Invariant option"
      "SCEntryAction option" "SCDoAction option"
      "SCExitAction option" "SCElement list"
      "SCInternTransition list"
and
SCElement =
    SCElementSCState SCState
  | SCElementSCTransition SCTransition
  | SCElementSCCode SCCode

datatype SCDefinition =
    SCDefinition "Completeness option" "Stereotype option" Name
      "ReferenceType option" "SCMethod option"
      "SCElement list" "Invariant list"

datatype MCCompilationUnit =
    MCCompilationUnit "Name list" "Name list" "STRING option" SCDefinition

end
```

Variante KB C.32 (Vereinfachte Statecharts)

SimplifiedSC

Isabelle-Theorie

```
theory SimplifiedSC
imports "$UMLP/gen/mc/umlp/sc/SCAS"
begin

fun initialOrFinal :: "SCModifier ⇒ bool"
where
 "initialOrFinal (SCModifier stereoOpt mVals) = (
   stereoOpt = None ∧
   ModifierValsLOCAL ∉ set mVals
  )"

fun simplifiedState :: "SCState ⇒ bool"
where
 "simplifiedState (SCState complOpt modifier name invOpt
                     entryOpt doOpt exitOpt subElems internTransL) =
   (initialOrFinal modifier ∧
    entryOpt = None ∧
    doOpt = None ∧
    exitOpt = None ∧
    subElems = [] ∧
    internTransL = [])"

fun simplifiedEvent :: "SCEvent ⇒ bool"
where
  "simplifiedEvent (SCEventSCReturnStatement r) = False"
| "simplifiedEvent (SCEventSCMethodOrExceptionCall m) = True"

fun simplifiedBody :: "SCTransitionBody ⇒ bool"
where
 "simplifiedBody (SCTransitionBody preOpt evOpt stmtOpt postOpt) = (
    if evOpt = None then True else simplifiedEvent (the evOpt)
 )"

fun simplifiedTransition :: "SCTransition ⇒ bool"
where
 "simplifiedTransition (SCTransition stereoOpt src trg bodyOpt) = (
     stereoOpt = None ∧
     (if bodyOpt = None then True else simplifiedBody (the bodyOpt))
 )"

```

```
fun simplifiedElems :: "SCElement list ⇒ bool"
where
 "simplifiedElems [] = True"
|"simplifiedElems ((SCElementSCState s)#xs) = (
     simplifiedState s ∧ simplifiedElems xs)"
|"simplifiedElems ((SCElementSCTransition t)#xs) = (
     simplifiedTransition t ∧ simplifiedElems xs)"
|"simplifiedElems ((SCElementSCCode c)#xs) = (simplifiedElems xs)"

fun wellformed-SimplifiedSC :: "SCDefinition ⇒ bool"
where
 "wellformed-SimplifiedSC
   (SCDefinition complOpt stereoOpt name refTypeOpt methOpt elemL invL)
     = (
    stereoOpt = None ∧
    simplifiedElems elemL
  )"

end
```

Variante KB C.33 (Ausschluss von beliebigem Code und Methoden-Statecharts)

NoCodeNoMethodSC

Isabelle-Theorie

```
theory NoCodeNoMethodSC
imports "$UMLP/gen/mc/umlp/sc/SCAS"
begin

fun simplifiedElems :: "SCElement list ⇒ bool"
where
 "simplifiedElems [] = True"
|"simplifiedElems ((SCElementSCState s)#xs) = (simplifiedElems xs)"
|"simplifiedElems ((SCElementSCTransition t)#xs) = (simplifiedElems xs)"
|"simplifiedElems ((SCElementSCCode c)#xs) = False"

fun wellformed-NoCodeNoMethodSC :: "SCDefinition ⇒ bool"
where
 "wellformed-NoCodeNoMethodSC
   (SCDefinition complOpt stereoOpt name refTypeOpt methOpt elemL invL)
     = (
    simplifiedElems elemL ∧ methOpt = None
  )"

end
```

Definition Variabilität C.34 (Presentation SC)

SCPresentation

Feature-Diagramm

```
package mc.umlp.sc.presentation;

featurediagram SCPresentation {

  SCPresentation = <<abb>> SCAbb;

  SCAbb = SimplifiedSC;
}
```

Definition Variabilität C.35 (Syntax SC)

SCSyntax

Feature-Diagramm

```
package mc.umlp.sc.syntax;

featurediagram SCSyntax {

  SCSyntax = <<lparam>> LangParam? & <<constr>> SCConstr?;

  LangParam = InvariantExpression? & Expressions?
             & Expression? & Statements?;

  InvariantExpression = JavaExpression;
  Expression = JavaExpression;
  Expressions = JavaExpression;
  Statements = JavaBlockStatement;

  SCConstr = NoCodeNoMethodSC?;
}
```

C.6 Sequenzdiagramme

Definition KS C.36 (SD)

SD

MontiCore-Grammatik

```
package mc.umlp.sd;

/**
@version 1.0a
*/
grammar SD extends mc.umlp.common.Common {

  options {
    nostring noident nows
    compilationunit SDDefinition
  }

  /*========================================================================*/
  /*================== INTERFACES AND EXTERNAL SYMBOLS =====================*/
  /*========================================================================*/
  external InvariantExpression;

  /** ASTInvariant represents an Invariant in the UML/P
      @attribute kind                Kind/Language of the Invariant
      @attribute invariantExpression Condition of the Invariant
  */
  Invariant =
    (kind:Name ":")?
    "[" InvariantExpression(parameter kind) "]";

  external Expression;

  /** ASTSDElement represents all Elements of a UML Sequencediagram
  */
  interface SDElement;

  /** ASTSDActivity represents all Activities of a UML Sequencediagram
```

```
*/
interface SDActivity;

/** ASTSDInteraction represents Interactions of a UML Sequencediagram
*/
interface SDInteraction;

/** ASTSDCallMessage represents the call of a message of an Interaction
*/
interface SDCallMessage;

/** ASTSDReturnMessage represents a return message of an Interaction
*/
interface SDReturnMessage;

/*========================================================================*/
/*============================== GRAMMAR =================================*/
/*========================================================================*/

/** ASTSDDefinition represents a UML Sequencediagram
    @attribute completeness Optional Comleteness of this Sequencediagramm
    @attribute stereotype   Optional Stereotype
    @attribute name         Name of this Sequencediagram
    @attribute sDElements   List of the content of this Sequencediagram
                            (Objects, ActivityStart and ActivityEnd,
                            Interactions, and Conditions)
    @attribute invaritants  List of Invaritants of this Sequencediagram
*/
SDDefinition =
  Completeness?
  Stereotype?
  "sequencediagram" Name
  "{"
    (
      ((Name ":" "[") | "[" )=> (invariants:Invariant ";")
      |
      sDElements:SDElement
    )*
  "}";

/** ASTSDObject represents an Object in a UML Sequencediagram
    @attribute completeness Optional Comleteness of this Object
    @attribute modifier     Modifier of this Object
    @attribute name         Name of this Object
    @attribute type         Optional type of this Object
*/
SDObject implements
(Completeness? Modifier Name (":" ReferenceType)? ";")=>SDElement =
  Completeness?
  Modifier
  Name (":" type:ReferenceType)? ";";

/** ASTSDComplexActivity represents a list of Activities in a UML
    Sequencediagram
    @attribute sDActivities List of Activities
*/
SDComplexActivity implements SDElement, SDActivity =
  "{" sDActivities:SDActivity+ "}";

/** ASTSDActivityStart represents the beginning of an Activation-Box
    (Activity) in a UML Sequencediagram
    @attribute name      Name of this Activity
    @attribute reference Name of the Object associated with this Activity
*/
SDActivityStart implements SDElement, SDActivity =
  "start" Name reference:Name ";";

/** ASTSDActivityEnd represents the end of an Activation-Box (Activity)
    in a UML Sequencediagram
    @attribute names List of Names of finished Activities
*/
SDActivityEnd implements SDElement, SDActivity =
  "end" names:Name ("," names:Name)* ";";

/** ASTSDCallInteraction represents an Interaction Call
    ("->"; new or MethodCall) in a UML Sequencediagram
    @attribute sourceReference Name of the Object which is the source of
                               this Interaction
    @attribute activityStart   Optional Name of the Activation-Box
                               (Activity) started with this Interaction
    @attribute targetReference Name of the Object which is the target of
                               this Interaction
    @attribute stereotype      Optional Stereotype of this Interaction
    @attribute sDCallMessage   Contains the message of the Interaction
*/
SDCallInteraction implements SDElement, SDInteraction, SDActivity =
  sourceReference:Name "->" activityStart:Name? targetReference:Name
  ":"
  (("<<")=> Stereotype)?
  SDCallMessage;

/** ASTSDReturnInteraction represents an Interaction Return
    ("<-"; return or Exception) in a UML Sequencediagram
    @attribute sourceReference Name of the Object which is the source of
                               this Interaction
    @attribute activityEnd     Optional Name of the Activation-Box
                               (Activity) stoped with this Interaction
    @attribute targetReference Name of the Object which is the target of
                               this Interaction
    @attribute stereotype      Optional Stereotype of this Interaction
    @attribute sDReturnMessage Contains the message of the Interaction
*/
SDReturnInteraction implements SDElement, SDInteraction, SDActivity =
  targetReference:Name "<-" activityEnd:Name? sourceReference:Name
  ":"
  (("<<")=> Stereotype)?
  SDReturnMessage;

/** ASTSDMethodCall represents a method call of an Interaction in a
    UML Sequencediagram
    @attribute methodName  Name of this method call
    @attribute sDArguments Optional Arguments of this method call
*/
SDMethodCall implements SDCallMessage =
  methodName:QualifiedName SDArguments? ";";

/** ASTSDNewStatement represents a new statement of an Interaction
    in a UML Sequencediagram
    @attribute objectName  Name of the new object to create
    @attribute sDArguments Optional Arguments of the constructor
*/
SDNewStatement implements SDCallMessage =
  "new" objectName:QualifiedName SDArguments? ";";

  /** ASTSDException represents an exception of an Interaction in
      a UML Sequencediagram
      @attribute exceptionName Name of the exception
      @attribute sDArguments   Optional Arguments of the exception
  */
  SDException implements SDReturnMessage =
     exceptionName:QualifiedName SDArguments? ";";

  /** ASTSDReturnStatement represents a return statement of an
      Interaction in a UML Sequencediagram
      @attribute incomplete True if return statement is incomplete
                            (underspecified)
      @attribute expression Expression of this return statement
  */
  SDReturnStatement implements SDReturnMessage =
    (("return" "...")=> ("return" incomplete:[INCOMPLETE:"..."])
    | ("return" Expression)) ";";

  /** ASTSDArguments represents Arguments of an Interaction/Message in a
      UML Sequencediagram
      @attribute incomplete  True if Arguments are incomplete (underspecified)
      @attribute expressions Specified Arguments as a list of Expressions
  */
  SDArguments =
    ("(" "...")=> ("(" incomplete:[INCOMPLETE:"..."] ")")
    | ("(" ")")=> ("(" ")")
    | ("(" expressions:Expression ("," expressions:Expression)* ")");

  /** ASTSDCondition represents a Condition in a UML Sequencediagram
      @attribute scope     List of Object-names included in this Condition
      @attribute invariant The Condition itself
  */
  SDCondition implements
  ("<" Name ("," Name)* ":" ((Name ":") | "["))=> SDElement, SDActivity =
    "<" scope:Name ("," scope:Name)* ":" Invariant ">" ";";

}
```

Definition AS C.37 (SDAS)

SDAS

Isabelle-Theorie (gen)

```
(*
Source: mc.umlp.sd.SD V. 1.0a
Generator: GenASWorkflow2 V. 0.3
Tue Jun 08 16:25:30 CEST 2010
*)
theory SDAS
imports "$UMLP/def/mc/umlp/sd/ExternalSDAS" "$UMLP/gen/mc/umlp/common/CommonAS"
begin

datatype SDArgumentsincomplete =
    SDArgumentsincompleteINCOMPLETE

datatype SDArguments =
    SDArguments "Expressions list" "SDArgumentsincomplete option"

datatype SDMethodCall =
    SDMethodCall QualifiedName "SDArguments option"

datatype SDNewStatement =
    SDNewStatement QualifiedName "SDArguments option"

datatype SDException =
    SDException QualifiedName "SDArguments option"

datatype SDReturnStatementincomplete =
    SDReturnStatementincompleteINCOMPLETE

datatype SDReturnStatement =
    SDReturnStatement "Expression option"
      "SDReturnStatementincomplete option"

datatype SDReturnMessage =
    SDReturnMessageSDException SDException
  | SDReturnMessageSDReturnStatement SDReturnStatement

datatype SDReturnInteraction =
    SDReturnInteraction Name "Name option" Name "Stereotype option"
      SDReturnMessage

datatype SDCallMessage =
    SDCallMessageSDMethodCall SDMethodCall
  | SDCallMessageSDNewStatement SDNewStatement

datatype SDObject =
    SDObject "Completeness option" Modifier Name "ReferenceType option"

datatype SDActivityStart =
    SDActivityStart Name Name

datatype SDActivityEnd =
    SDActivityEnd "Name list"

datatype SDCallInteraction =
    SDCallInteraction Name "Name option" Name "Stereotype option"
      SDCallMessage

datatype Invariant =
    Invariant "Name option" InvariantExpression

datatype SDCondition =
    SDCondition "Name list" Invariant

datatype SDActivity =
    SDActivitySDComplexActivity SDComplexActivity
  | SDActivitySDActivityStart SDActivityStart
  | SDActivitySDActivityEnd SDActivityEnd
  | SDActivitySDCallInteraction SDCallInteraction
  | SDActivitySDReturnInteraction SDReturnInteraction
  | SDActivitySDCondition SDCondition
and
SDComplexActivity =
    SDComplexActivity "SDActivity list"

datatype SDElement =
    SDElementSDObject SDObject
  | SDElementSDComplexActivity SDComplexActivity
  | SDElementSDActivityStart SDActivityStart
  | SDElementSDActivityEnd SDActivityEnd
  | SDElementSDCallInteraction SDCallInteraction
  | SDElementSDReturnInteraction SDReturnInteraction
  | SDElementSDCondition SDCondition
```

```

datatype SDDefinition =
    SDDefinition "Completeness option" "Stereotype option" Name
      "SDElement list" "Invariant list"

datatype MCCompilationUnit =
    MCCompilationUnit "Name list" "Name list" "STRING option" SDDefinition

datatype SDInteraction =
    SDInteractionSDCallInteraction SDCallInteraction
  | SDInteractionSDReturnInteraction SDReturnInteraction

end
```

Variante KB C.38 (Vereinfachte Sequenzdiagramme)

SimplifiedSD

Isabelle-Theorie

```
theory SimplifiedSD
imports "$UMLP/gen/mc/umlp/sd/SDAS"
begin

fun wellformed-SimplifiedObject :: "SDObject ⇒ bool"
where
 "wellformed-SimplifiedObject
   (SDObject compOpt (Modifier stereoOpt valList) n refTOpt) =
    (valList = [])"

fun wellformed-SimplifiedCallMsg :: "SDCallMessage ⇒ bool"
where
  "wellformed-SimplifiedCallMsg (SDCallMessageSDMethodCall mcall) = True"
| "wellformed-SimplifiedCallMsg (SDCallMessageSDNewStatement x) = False"

fun wellformed-SimplifiedCall :: "SDCallInteraction ⇒ bool"
where
 "wellformed-SimplifiedCall
   (SDCallInteraction src actStart trg stereoOpt callMsg) =
    (actStart = None ∧ stereoOpt = None ∧ wellformed-SimplifiedCallMsg callMsg)"

fun wellformed-SimplifiedReturn :: "SDReturnInteraction ⇒ bool"
where
 "wellformed-SimplifiedReturn
   (SDReturnInteraction src actStart trg stereoOpt retMsg) =
    (actStart = None ∧  stereoOpt = None)"

fun wellformed-SimplifiedElems :: "SDElement ⇒ bool"
where
  "wellformed-SimplifiedElems (SDElementSDObject x)
     = wellformed-SimplifiedObject x"
| "wellformed-SimplifiedElems (SDElementSDComplexActivity a) = False"
| "wellformed-SimplifiedElems (SDElementSDActivityStart a)   = False"
| "wellformed-SimplifiedElems (SDElementSDActivityEnd a)     = False"
| "wellformed-SimplifiedElems (SDElementSDCallInteraction c)
     = wellformed-SimplifiedCall c"
| "wellformed-SimplifiedElems (SDElementSDReturnInteraction c)
     = wellformed-SimplifiedReturn c"
| "wellformed-SimplifiedElems (SDElementSDCondition c)       = True"

fun wellformed-SimplifiedSD :: "SDDefinition ⇒ bool"
where
 "wellformed-SimplifiedSD
```

```
    (SDDefinition complOpt stereoOpt name elems invars) = (
      foldr (λ x y . x ∧ y) (map wellformed-SimplifiedElems elems) True
    )"

end
```

Definition Variabilität C.39 (Presentation SD)

```
SDPresentation                                          Feature-Diagramm
package mc.umlp.sd.presentation;

featurediagram SDPresentation {

  SDPresentation = <<abb>> SDAbb;

  SDAbb = SimplifiedSD;
}
```

Definition Variabilität C.40 (Syntax SD)

```
SDSyntax                                                Feature-Diagramm
package mc.umlp.sd.syntax;

featurediagram SDSyntax {

  SDSyntax = <<lparam>> LangParam? & <<stereos>> vStereos?;

  LangParam = InvariantExpression? & Expressions?;

  InvariantExpression = JavaExpression;
  Expressions = JavaExpression;

  vStereos = CompleteVisibleInitialFree;
}
```

Anhang D

Glossar

Die folgenden Begriffe wurden bei der Entwicklung dieser Arbeit geprägt oder orientieren sich an den Definitionen in [Kra10].

Abkürzung
: Eine (modellierungssprachliche) Abkürzung bezeichnet ein Sprachkonstrukt, das semantikerhaltend durch andere Konstrukte ausgedrückt werden kann. Varianten von Abkürzungen sind Teil der →Präsentationsvariabilität einer Sprache. Synonym wird auch der Begriff „Präsentationserweiterung“ gebraucht.

Abstrakter Syntaxbaum (AST)
: Der abstrakte Syntaxbaum (engl. Abstract Syntax Tree) ist eine baumartige Struktur, die von einem Parser beim Verarbeiten eines →Modells erstellt wird. Der AST ist eine abstrahierte Darstellung des Modells und wird für dessen Weiterverarbeitung (zum Beispiel durch einen →Generator) verwendet. Der AST kann durch zusätzliche Links und Knoten erweitert sein, die eine effiziente Navigation erlauben oder eine Zusammenfassung bzw. Aufbereitung der AST-Informationen beinhalten (zum Beispiel eine Symboltabelle). Der Kern-AST bleibt dabei als aufspannender Baum erhalten.

Abstrakte Syntax
: Die abstrakte Syntax einer →(Modellierungs)Sprache beschreibt die strukturelle Essenz einer Sprache [Wil97]. Diese Struktur legt die Menge der möglichen →abstrakten Syntaxbäume fest.

Domänenspezifische Sprache (DSL)
: Eine domänenspezifische Sprache (engl. Domain Specific Language) ist eine Programmier- oder →Modellierungssprache, die auf eine Anwendungsdomäne zugeschnitten ist. Dadurch können die Problemstellungen oder Lösungen in einer Domäne kompakt beschrieben werden.

Einbettung

1. Bezeichnet in →MontiCore den Mechanismus zur Definition und Verwendung von →Sprachparametern. Technisch werden externe Produktionen mit Produktionen einer weiteren Sprache verbunden.
2. Bezeichnet die Kodierung einer Theorie mit Hilfe einer weiteren Theorie. Bei einer tiefen Einbettung (engl. deep embedding) werden alle Konstrukte der eingebetteten

Theorie in der einbettenden Theorie explizit repräsentiert. Bei der flachen Einbettung (engl. shallow embedding) hingegen werden die Konstrukte implizit durch passende Verwendung von bereits existierenden Konstrukten umgesetzt. Soll zum Beispiel wie in [WN04] eine spezielle *safety*-Logik in Logik höherer Stufe (HOL) kodiert werden, können Formeln der safety-Logik entweder flach als HOL-Formeln oder tief als explizite Repräsentation in Form eines Datentyps eingebettet werden.

Feature-Diagramm

Ein Feature-Diagramm [CE00] ist eine Baumstruktur und enthält →Variationspunkte und →Varianten und deren Abhängigkeiten. Ein Feature-Diagramm zeigt die →Variabilität beispielsweise eines Produkts oder einer Sprache.

Generator

Ein Generator ist eine Software und erzeugt aus einem →Modell Artefakte wie beispielsweise Quellcode oder andere Modelle.

Grammatikvererbung

Die Grammatikvererbung ist ein Mechanismus zur Spezialisierung von →MontiCore-Grammatiken. Die spezialisierende Grammatik erbt die Produktionen ihrer spezialisierten Grammatiken.

Isabelle/HOL

Isabelle ist ein generischer Theorembeweiser. Die Logik höherer Stufe (engl. Higher Order Logic) HOL ist die am besten entwickelte Objektlogik für Isabelle [NPW02].

Komposition

Komposition ist eine Operation, die mehrere Artefakte zu einem Artefakt zusammenführt. Das neue Artefakt kann dabei nur konzeptuell vorhanden sein. Die Operation muss typischerweise bestimmte Eigenschaften erfüllen, beispielsweise die Semantik der komponierten Artefakte im Kompositum reflektieren.

Konfiguration

Eine Konfiguration ist eine Auswahl von →Varianten eines →Feature-Diagramms.

Konkrete Syntax

Die konkrete Syntax legt fest, wie →Modelle einer →Sprache aufgebaut werden können. Mit ihr interagiert ein Benutzer der Sprache.

Kontextbedingung

Kontextbedingungen für eine Sprache schränken die Menge der erlaubten Modelle ein und ergänzen damit die Definition der konkreten oder abstrakten Syntax. Ein Modell, das auch die Kontextbedingungen erfüllt ist wohlgeformt (→Wohlgeformtheit). Als optionale Kontextbedingungen können sie →Abkürzungen ausschließen oder zusätzliche →Spracheinschränkungen definieren und dienen damit zur Erzeugung von →Varianten einer →Sprachdefinition.

Modell

Ein Modell ist ein Element einer →Modellierungssprache.

Modellbasierte Entwicklung

Bei der modellbasierten Entwicklung stehen →Modelle als primäre Entwicklungsartefakte im Zentrum eines Entwicklungsprozesses. Modelle einer →Modellierungssprache kommen bei verschiedenen Aktivitäten zum Einsatz. Sie werden beispielsweise automatisiert in Testfälle oder Produktcode transformiert oder dienen Analysezwecken.

Modellierungssprache

Eine Modellierungssprache ist eine →Sprache und dient zur abstrakten Spezifikation des zu entwickelnden Systems. Im Vergleich zu Programmen einer Programmiersprache können die →Modelle auch unvollständig oder unterspezifiziert sein und müssen nicht ausführbar sein. Durch die abstrakte Beschreibung ist oft eine Analyse wichtiger Systemeigenschaften formal möglich.

Modelltransformation

Eine Modelltransformation überführt →Modelle einer →Modellierungssprache in Modelle einer gegebenenfalls anderen Modellierungssprache.

MontiCore

MontiCore [Kra10] ist ein Framework zur Entwicklung →domänenspezifischer Sprachen. MontiCore erzeugt für eine →MontiCore-Grammatik Komponenten zur Sprachverarbeitung (Lexer, Parser, etc.) und stellt Funktionalität zur einfachen Entwicklung von generativen Werkzeugen zur Verfügung.

MontiCore-Grammatik

Die MontiCore-Grammatik ist das Eingabeformat für den MontiCore-Generator. Mit ihr wird die →konkrete und abstrakte Syntax einer →Sprache definiert.

Präsentationsoption

Eine Präsentationsoption bezeichnet eine alternative Darstellung eines Sprachkonstrukts in →konkreter Syntax, wobei die →abstrakte Syntax unverändert bleibt. Varianten von Präsentationsoptionen sind Teil der →Präsentationsvariabilität einer Sprache.

Präsentationsvariante

Eine Präsentationsvariante ist Teil der →Präsentationsvariabilität einer Sprache und ist entweder eine →Abkürzung oder eine →Präsentationsoption.

Präsentationsvariabilität

Präsentationsvariabilität umfasst alle →Präsentationsvarianten (→Präsentationsoptionen und →Abkürzungen) einer →Sprache.

Semantik

Die Semantik einer →Sprache ist deren Bedeutung. Semantik umfasst die →semantische Domäne und die →semantische Abbildung, die Elemente der →abstrakten Syntax mit Elemente der semantischen Domäne in Beziehung setzt [HR04].

Semantische Abbildung

Die semantische Abbildung ist Teil der Definition der →Semantik einer Sprache. Sie setzt die Elemente der →abstrakten Syntax mit Elemente der →semantischen Domäne in Beziehung.

Semantische Domäne

Auf Elemente der semantischen Domäne werden Elemente der →abstrakten Syntax mit Hilfe der →semantischen Abbildung abgebildet. Sie enthält also die Elemente, die die Bedeutung der Sprache bilden. Daher muss die semantische Domäne präzise definiert und gut verstanden sein.

Semantische Variabilität

Semantische Variabilität umfasst alle semantischen Varianten einer Sprache. Semantische Varianten können in der →semantischen Abbildung oder in der →semantischen Domäne auftreten.

Sprache

Eine Sprache bezeichnet eine Menge von wohlgeformten →Modellen, denen zudem eine →Semantik zugeordnet werden kann. Die Sprache wird durch die →Sprachdefinition festgelegt.

Sprachdefinition

Ein Sprachdefinition umfasst die Definition der →konkreten und abstrakten Syntax, der notwendigen →Kontextbedingungen und der →Semantik. Zusätzlich kann die Sprachdefinition durch →Varianten ergänzt werden.

Spracheinschränkungen

Spracheinschränkungen sind ein Teil der →syntaktischen Variabilität einer Sprache und schränken die Menge der erlaubten →Modelle beispielsweise durch optional nutzbare →Kontextbedingungen ein.

Sprachparameter

Sprachparameter gehören zur →syntaktischen Variabilität einer Sprache und bezeichnen Parameter einer Sprachdefinition, für die (Teile einer) unabhängige Sprachen (Parametersprachen) eingesetzt werden können. Die Syntax einer parametrischen Sprache ist also nicht vollständig festlegt und wird erst durch Belegung der Parameter fixiert. Im Kontext von →MontiCore wird der Begriff →Einbettung verwendet.

Stereotyp

Der aus der UML [OMG09b] entlehnte Begriff Stereotyp bezeichnet die Annotation eines Modellelements mit bestimmten, festlegten Namen oder Name-Wert-Paaren, um hierdurch die Bedeutung des Elements anzupassen. Stereotypen sind Teil der →syntaktischen Variabilität.

Syntaktische Variabilität

Die syntaktische Variabilität einer Sprache umfasst syntaktischen Varianten. Hierzu zählen →Stereotypen, →Sprachparameter und →Spracheinschränkungen/-erweiterungen.

Syntax

Die Syntax einer Sprache umfasst die →konkrete Syntax, die →abstrakte Syntax und die →Kontextbedingungen.

Systemmodell

Das Systemmodell ist eine Sammlung mathematischer Theorien, die objektbasierte Systeme abstrakt charakterisieren. Das Systemmodell wird als →semantische Domäne für objektbasierte →Modellierungssprachen verwendet. Ein Element des Systemmodells ist ein unterspezifiziertes objektbasiertes →Transitionssystem und beschreibt die Struktur von Klassen und das Verhalten und die Interaktion der Objekte.

Theorie

Eine Theorie bezeichnet im Allgemeinen eine Sammlung mathematischer Definitionen und Beweise und im Speziellen die von →Isabelle/HOL verarbeiteten Dateien.

Transitionssystem

Ein Transitionssystem im →Systemmodell legt zustandsbasiert das Verhalten von Objekten fest. Aufgrund einer Eingabe und basierend auf dem aktuellen Zustand wird ein Folgezustand eingenommen und gegebenenfalls eine Ausgabe getätigt.

Variabilität

Variabilität bezeichnet die Menge der möglichen →Varianten und die Abhängigkeiten zwischen ihnen. Sie wird in →Feature-Diagrammen dokumentiert. Die Variabilität einer →Sprachdefinition ist durch →Präsentationsvariabilität, →syntaktische Variabilität und →semantische Variabilität beschreibbar.

Variante

Eine Variante entspricht einem Blatt in einem →Feature-Diagramm. Eine Variante kann in einer →Konfiguration ausgewählt werden.

Variationspunkt

Ein Variationspunkt organisiert einen bestimmten Aspekt der Variabilität und entspricht einem Zwischenknoten im →Feature-Diagramm. Zusammengehörende Varianten werden unter einem Variationspunkt gruppiert.

Verifikation

Verifikation bezeichnet den Vorgang des Nachweises einer Behauptung. Mit Hilfe von →Isabelle/HOL lassen sich Verifikationsaufgaben (Beweise) rechnerunterstützt durchführen.

Wohlgeformtheit

Wohlgeformtheit ist eine Eigenschaft eines →Modells. Ein Modell ist wohlgeformt, falls es syntaktisch korrekt ist und insbesondere die →Kontextbedingungen erfüllt.

Anhang E

Abkürzungen

API	Application Programming Interface
ASF	Algebraic Specification Formalism
ASM	Abstract State Machine
AST	Abstract Syntax Tree
DFG	Deutsche Forschungsgemeinschaft
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
GReAT	Graph Rewriting and Transformation
HOL	Higher Order Logic
HTML	HyperText Markup Language
IPSEN	Integrated Project Support ENvironment
MOF	Meta Object Facility
OAW	OpenArchitectureWare
OCL	Object Constraint Language
OMG	Object Management Group
PROGRES	PROgrammed Graph REwriting Systems
PSG	Programming System Generator
QVT	Query/Views/Transformations
PVS	Prototype Verification System

RTC	Run-To-Completion
rUML	Rigorous UML
SDF	Syntax Specification Formalism
SOS	Structured Operational Semantics
SQL	Structured Query Language
SYSLAB	Forschungslabor für System und Softwareentwicklung
UML	Unified Modeling Language
UML/P	Unified Modeling Language / Programmier-geeignet

Anhang F

Index der Systemmodelldefinitionen

Die erste der angegebenen Seitenzahlen (dritte und vierte Spalte) bezieht sich auf die Definition in Kapitel 4 oder Anhang A, die zweite Seitenzahl verweist auf die entsprechende Isabelle-Theorie in Anhang B. Die Markierung V zeigt an, dass es sich um eine Variante handelt.

$.^{\&}$	Typkonstruktor für Klassen (Objektidentifikatoren sind Werte)	55	252	
$.^{*}$	Typkonstruktor für Klassen (Instanzen sind Werte)	56	253	V
$*$	Wildcard	241		
$. \otimes .$	Komposition von Zustandsmaschinen	243		
		77		
addobj	Objekt in Datenzustand speichern	58	254	
attr	Attribute einer Klasse	53	250	
attributeAssoc	Realisierungsvariante für Assoziationen	61	257	V
attrOid	Attribute eines Objektidentifikators	54	251	
binaryAssoc	Binäre Assoziation	61	256	V
binaryAssocMultis	Multiplizität für Assoziationen	61	256	V
binaryRelOf	Extrahieren von Links für binäre Assoziationen	61	256	V
Bool	Typname für Boolean	50	247	
callsOf	Menge der Methodenaufrufe	72	268	
callsOfO	Menge der Methodenaufrufe eines Objekts	72	268	
CAR	Trägermenge eines Typs	49	247	
Char	Typname für Zeichen	51	249	V
CharSet	Menge der Zeichen	51	249	V
classes	An einer Assoziation beteiligte Klassen	60	255	
classOf	Klasse eines Objektidentifikators	53	250	
	Klasse einer Operationen	65	260	
compositeAssoc	Komposition (Assoziation)	62	258	V
ControlStore	Speicher für den Kontrollzustand	69	264	
created	Wurde Objekt im Zustand erzeugt	78	274	
csOf	Kontrollzustand eines Gesamtzustands	75	270	
DataStore	Speicher für Datenzustand	57	254	
definedIn	Klasse einer Methode	66	261	
derivedAssoc	Abgeleitete Assoziationen	62	259	V

destinations	Assoziationspartner (Ziele)	61	256	V
dsOf	Datenzustand eines Gesamtzustands	75	270	
esOf	Ereignisspeicher	79	274	V
extraVals	Zusätzliche Werte für Assoziationen	60	255	
false	Wert "falsch" für Bool	50	247	
following	Ist Zustand ein Folgezustand	78	274	
FRAME	Menge der Stack-Frames	68	264	
framesOf	Menge der Stack-Frames einer Methode	68	264	
getAttr	Attributzugriff	54	251	
hasMsg	Ist eine Nachricht in Objektzustand vorhanden	75	270	
impl	Implementierung einer Operation	67	262	
INSTANCE	Menge der Instanzen	53	250	
Int	Typname für Integer	50	247	
isInterface	Ist Klasse ein Interface	67	262	
localNames	Lokale Variablen einer Methode	66	261	
localsOf	Mögliche lokale Variablenwerte einer Methode	66	261	
MESSAGE	Menge der Nachrichten	72	267	
messages	Nachrichten eines Objekts	72	267	
MessageStore	Speicher für den Nachrichtenzustand	72	267	
msgIn	Eingehende Nachrichten eines Objekts	72	267	
msgOut	Ausgehende Nachrichten eines Objekts	72	267	
msgReceivedNow	Nachricht in Zustand empfangen	79	274	V
msgSentNow	Nachricht in Zustand gesendet	79	274	V
msOf	Nachrichtenzustand eines Gesamtzustands	75	270	
mtomAssoc	Realisierungsvariante für Assoziationen	62	258	V
Name	Menge von Namen	51	245	
nameOf	Name einer Operationen	65	260	
	Name einer Methode	66	261	
Nil	Spezieller Objektidentifikator	54	251	
notExecuting	Wird eine Methoden in einem Objektzustand *nicht* ausgeführt	75	270	
objects	Instanzen eines Objekts	53	250	
objectsOfClass	Instanzen einer Klasse	53	250	
oids	Objektidentifikatoren einer Klasse	53	250	
	Aktuelle Objektmenge im Datenzustand	57	254	
	Aktuelle Objektmenge eines Zustands	75	270	
OSTATE	Zustandsraum der Objekte	75	270	
osts	Zustandsmaschine eines Objekts	77	272	
OSTS	Zustandsmaschine aller Objekten	78	273	
params	Mögliche Parameterwerte einer Operation	65	260	
parNames	Parameter einer Methode	66	261	
parOf	Mögliche Parameterwerte einer Methode	66	261	
parTypes	Formale Parameter einer Operation	65	260	

pcOf	Mögliche Programmzähler einer Methode	66	261	
qualifiedRelOf	Extrahieren von Links für qualifizierte Assoziationen	63	259	V
query	Abfrage über Systemzustand	76	271	V
reachable	Erreichbare Zustände	78	274	
ReceivedEvent	Empfangsereignis	79	274	V
relOf	Extrahieren von Links aus Datenzustand	60	255	
resType	Rückgabetyp einer Operation	65	260	
	Rückgabetyp einer Methode	66	261	
returnsOf	Menge der Returns	73	269	
returnsOfO	Menge der Returns eines Objekts	73	269	
rightQualifiedAssoc	Qualifizierte Assoziation	63	259	V
running	Läuft eine Methoden in Objektzustand	75	270	
SentEvent	Sendeereignis	79	274	V
setval	Werte setzen für Datenzustand	58	254	
sources	Assoziationspartner (Quellen)	61	256	V
state	Zustand eines Objekts	75	270	
STATE	Zustandsraum des Systems	75	270	
statesO	Mögliche Zustände eines Objekts	75	270	
StaticAttr	Menge der statischen Attribute	59	256	V
StaticC	Statische Klasse	59	256	V
staticOid	Objektidentifikator zu StaticC	59	256	V
StaticOpn	Menge der statischen Operationen	68	266	V
String	Typname für Zeichenketten	51	249	V
sts	Zustandsmaschine einer Menge von Objekten	78	273	
STS(S, I, O)	Zustandsmaschinen	243	272	
sub	Subklassenrelation	55	252	
SYSSTS	Zustandsmaschine des Gesamtsystems	78	273	
SystemModel	Systemmodell	49	275	
this	Objektidentifikator einer Instanz	54	251	
TI	Typkonstruktor für Klassen (Instanzen sind Werte)	56	253	V
TO	Typkonstruktor für Klassen (Objektidentifikatoren sind Werte)	55	252	
trace	Systemablauf		274	
TRACE	Menge der Systemabläufe	78	274	
true	Wert “wahr” für Bool	50	247	
typeOf	Typ eines Wertes	50	249	V
UASSOC	Universum der Assoziationen	60	255	
UCLASS	Universum der Klassennamen	53	250	
UEVENT	Universum der Ereignisse	79	274	V
UMETH	Universum der Methoden	66	261	
UOID	Universum der Objektidentifikatoren	53	250	

UOPN	Universum der Operationen	65	260
UPC	Universum der Programmzähler	66	261
USIGNAL	Universum der Signale	74	269
UTHREAD	Universum der Threads	69	264
UTYPE	Universum der Typenamen	49	247
UVAL	Universum der Werte	49	247
UVAR	Universum der Variablennamen	51	248
val	Wertabfrage für Datenzustand	58	254
VarAssign	Menge der Variablenbelegungen	51	248
vname	Name einer Variable	51	248
void	Wert void	50	247
Void	Typname für Void	50	247
vtype	Typ einer Variable	51	248

Anhang G

Lebenslauf

Name	Grönniger
Vorname	Hans
Geburtstag	15.05.1979
Geburtsort	Meppen
Staatsangehörigkeit	deutsch

seit 2009	Wissenschaftlicher Mitarbeiter am Lehrstuhl Informatik 3 (Software Engineering), RWTH Aachen
2005-2009	Wissenschaftlicher Mitarbeiter am Institut für Software Systems Engineering, TU Braunschweig
2005	Abschluss als Diplom-Informatiker
1999 - 2005	Studium der Informatik an der TU Braunschweig
1998 - 1999	Zivildienst
1998	Abitur
1985 - 1998	Grundschule, Orientierungsstufe und Gymnasium in Meppen

Aachener Informatik-Berichte

This is the list of all technical reports since 1987. To obtain copies of reports please consult `http://aib.informatik.rwth-aachen.de/` or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

1987-01 * Fachgruppe Informatik: Jahresbericht 1986
1987-02 * David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes
1987-03 * Manfred Nagl: A Software Development Environment based on Graph Technology
1987-04 * Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment
1987-05 * Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
1987-06 * Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*
1987-07 * Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method
1987-08 * Manfred Nagl: Set Theoretic Approaches to Graph Grammars
1987-09 * Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents
1987-10 * Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions
1987-11 * Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory
1987-12 J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata
1988-01 * Gabriele Esser, Johannes Rückert, Frank Wagner Gesellschaftliche Aspekte der Informatik
1988-02 * Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
1988-03 * Thomas Welzel: Simulation of a Multiple Token Ring Backbone
1988-04 * Peter Martini: Performance Comparison for HSLAN Media Access Protocols
1988-05 * Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 * Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 * Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 * Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 * W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 * Kai Jakobs: Towards User-Friendly Networking
1988-11 * Kai Jakobs: The Directory - Evolution of a Standard

1988-12 *	Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 *	Martine Schümmer: RS-511, a Protocol for the Plant Floor
1988-14 *	U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing
1988-15 *	Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation
1988-16 *	Fachgruppe Informatik: Jahresbericht 1987
1988-17 *	Wolfgang Thomas: Automata on Infinite Objects
1988-18 *	Michael Sonnenschein: On Petri Nets and Data Flow Graphs
1988-19 *	Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers
1988-20 *	Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen
1988-21 *	Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment
1988-22 *	Joost Engelfriet, Heiko Vogler: Modular Tree Transducers
1988-23 *	Wolfgang Thomas: Automata and Quantifier Hierarchies
1988-24 *	Uschi Heuter: Generalized Definite Tree Languages
1989-01 *	Fachgruppe Informatik: Jahresbericht 1988
1989-02 *	G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
1989-03 *	Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions
1989-04 *	Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language
1989-05	J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)
1989-06 *	Kai Jakobs: OSI - An Appropriate Basis for Group Communication?
1989-07 *	Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems
1989-08 *	Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control
1989-09 *	Peter Martini: High Speed Local Area Networks - A Tutorial
1989-10 *	P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation
1989-11 *	Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science
1989-12 *	Peter Martini: The DQDB Protocol - Is it Playing the Game?
1989-13 *	Martine Schümmer: CNC/DNC Communication with MAP
1989-14 *	Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements
1989-15 *	M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

1989-16 *	G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model
1989-17 *	J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling
1989-18	A. Maassen: Programming with Higher Order Functions
1989-19 *	Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL
1989-20	H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language
1990-01 *	Fachgruppe Informatik: Jahresbericht 1989
1990-02 *	Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)
1990-03 *	Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas
1990-04	R. Loogen: Stack-based Implementation of Narrowing
1990-05	H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies
1990-06 *	Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication
1990-07 *	Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work
1990-08 *	Kai Jakobs: Directory Names and Schema - An Evaluation
1990-09 *	Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke
1990-11	H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine
1990-12 *	Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit
1990-13 *	Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains
1990-14	A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)
1990-15 *	Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars
1990-16	A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars
1990-17 *	Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit
1990-18 *	Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen
1990-20	Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations
1990-21 *	Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences
1990-22	H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming
1991-01	Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990

1991-03	B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence
1991-04	M. Portz: A new class of cryptosystems based on interconnection networks
1991-05	H. Kuchen, G. Geiler: Distributed Applicative Arrays
1991-06 *	Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension
1991-07 *	Ludwig Staiger: Syntactic Congruences for w-languages
1991-09 *	Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System
1991-10	K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming
1991-11	R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages
1991-12 *	K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming
1991-13 *	Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline
1991-14 *	Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes
1991-15	J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability
1991-16	J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design
1991-17	A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems
1991-18 *	Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems
1991-19	M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification
1991-20	G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs
1991-21 *	Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint
1991-22	H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL
1991-23	S. Graf, B. Steffen: Compositional Minimization of Finite State Systems
1991-24	R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems
1991-25 *	Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks
1991-26	M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases
1991-27	J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem
1991-28	J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion
1991-30	T. Margaria: First-Order theories for the verification of complex FSMs
1991-31	B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications
1992-01	Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991

1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen

1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability

1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories

1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes

1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems

1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line

1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme

1992-10 Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management

1992-11 Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines

1992-12 W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking

1992-13 * Matthias Jarke, Thomas Rose: Specification Management with CAD

1992-14 Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars

1992-15 A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)

1992-16 * Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik

1992-17 M. Jarke (ed.): ConceptBase V3.1 User Manual

1992-18 * Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems

1992-19-00 H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages

1992-19-01 G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)

1992-19-02 H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine

1992-19-03 C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus

1992-19-04 T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions

1992-19-05 S. Kahrs: Polymorphic Type Checking by Interpretation of Code

1992-19-06 M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine

1992-19-07 J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)

1992-19-08	D. Gärtner, A. Kimms, W. Kluge: pi-Redˆ+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus
1992-19-09	D. Howe, G. Burn: Experiments with strict STG code
1992-19-10	J. Glauert: Parallel Implementation of Functional Languages Using Small Processes
1992-19-11	M. Joy, T. Axford: A Parallel Graph Reduction Machine
1992-19-12	A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction
1992-19-13	K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)
1992-19-14	K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer
1992-19-15	S. Mintchev: Using Strictness Information in the STG-machine
1992-19-16	D. Rushall: An Attribute Grammar Evaluator in Haskell
1992-19-17	J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation
1992-19-18	W.S. Martins: Parallel Implementations of Functional Languages
1992-19-19	D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)
1992-19-20	J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment
1992-19-21	S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)
1992-19-22	G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers
1992-19-23	S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)
1992-19-24	W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)
1992-19-25	M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting
1992-19-26	Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)
1992-19-27	H. Doerr: Monitoring with Graph-Grammars as formal operational Models
1992-19-28	J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures
1992-19-29	G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)
1992-20	H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language
1992-21	H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language
1992-22	H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing
1992-23	T. Margaria, B. Steffen: Distinguishing Formulas for Free
1992-24	K. Pohl: The Three Dimensions of Requirements Engineering

1992-25 *	R. Stainov: A Dynamic Configuration Facility for Multimedia Communications
1992-26 *	Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification
1992-27	W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety
1992-28 *	Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design
1992-29	B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik
1992-30	A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic
1992-32 *	Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems
1992-33 *	B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: Euro-Bridge: Communication Services for Multimedia Applications
1992-34	C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice
1992-35	J. Börstler: Feature-Oriented Classification and Reuse in IPSEN
1992-36	M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis
1992-37 *	K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models
1992-38	A. Zuendorf: Implementation of the imperative / rule based language PROGRES
1992-39	P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme
1992-40 *	Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks
1992-41 *	Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems
1992-42 *	P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components
1992-43	W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages
1992-44	N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications
1993-01 *	Fachgruppe Informatik: Jahresbericht 1992
1993-02 *	Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems
1993-03	G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments
1993-05	A. Zündorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES
1993-06	A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels

1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages

1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager

1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept

1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain

1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes

1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing

1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel

1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control

1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle

1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993

1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications

1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse

1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations

1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics

1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations

1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository

1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments

1994-09 *	Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems
1994-11	A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems
1994-12	A. Schürr: Specification of Graph Translators with Triple Graph Grammars
1994-13	A. Schürr: Logic Based Programmed Structure Rewriting Systems
1994-14	L. Staiger: Codes, Simplifying Words, and Open Set Condition
1994-15 *	Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents
1994-16	P. Klein: Designing Software with Modula-3
1994-17	I. Litovsky, L. Staiger: Finite acceptance of infinite words
1994-18	G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction
1994-19	M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas
1994-20 *	R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)
1994-21	M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras
1994-22	H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry
1994-24 *	M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach
1994-25 *	M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach
1994-26 *	St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment
1994-27 *	M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments
1994-28	O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes
1995-01 *	Fachgruppe Informatik: Jahresbericht 1994
1995-02	Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES
1995-03	Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction
1995-04	Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries
1995-05	Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types
1995-06	Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases
1995-07	Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies
1995-08	Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09	Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages
1995-10	Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency
1995-11 *	M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases
1995-12 *	G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology
1995-13 *	M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views
1995-14 *	P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work
1995-15 *	Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems
1995-16 *	W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming
1996-01 *	Jahresbericht 1995
1996-02	Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees
1996-03 *	W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
1996-04	Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability
1996-05	Klaus Pohl: Requirements Engineering: An Overview
1996-06 *	M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
1996-07	Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs
1996-08 *	S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
1996-09	Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming
1996-09-0	Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents
1996-09-1	Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines
1996-09-2	Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation
1996-09-3	Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell
1996-09-4	Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems
1996-09-5	Alexandre Tessier: Declarative Debugging in Constraint Logic Programming
1996-10	Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management
1996-11 *	C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement

1996-12 *	R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
1996-13 *	K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools
1996-14 *	R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996
1996-15 *	H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases
1996-16 *	M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization
1996-17	Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet
1996-18	Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation
1996-19 *	P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations
1996-20	Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems
1996-21 *	G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto
1996-22 *	S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality
1996-23 *	M.Gebhardt, S.Jacobs: Conflict Management in Design
1997-01	Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996
1997-02	Johannes Faassen: Using full parallel Boltzmann Machines for Optimization
1997-03	Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems
1997-04	Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler
1997-05 *	S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge
1997-06	Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries
1997-07	Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen
1997-08	Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting
1997-09	Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets
1997-10	Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases
1997-11 *	R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations

1997-13	Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs
1997-14	Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System
1997-15	George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms
1998-01 *	Fachgruppe Informatik: Jahresbericht 1997
1998-02	Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes
1998-03	Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr
1998-04 *	O. Kubitz: Mobile Robots in Dynamic Environments
1998-05	Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems
1998-06 *	Matthias Oliver Berger: DECT in the Factory of the Future
1998-07	M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
1998-09 *	Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
1998-10 *	M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases
1998-11 *	Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML
1998-12 *	W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
1998-13	Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
1999-01 *	Jahresbericht 1998
1999-02 *	F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
1999-03 *	R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager
1999-04	María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
1999-05 *	W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference
1999-06 *	Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie
1999-07	Thomas Wilke: CTL+ is exponentially more succinct than CTL
1999-08	Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 *	Jahresbericht 1999
2000-02	Jens Vöge, Marcin Jurdzinski A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-03	D. Jäger, A. Schleicher, B. Westfechtel: UPGRADE: A Framework for Building Graph-Based Software Engineering Tools
2000-04	Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05	Mareike Schoop: Cooperative Document Management
2000-06	Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 *	Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08	Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 *	Jahresbericht 2000
2001-02	Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03	Thierry Cachat: The power of one-letter rational languages
2001-04	Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05	Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08	Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07	Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08	Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09	Markus Mohnen: Interfaces with Default Implementations in Java

2002-10	Martin Leucker: Logics for Mazurkiewicz traces
2002-11	Jürgen Giesl, Hans Zantema: Liveness in Rewriting
2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)
2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"
2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22	Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23	Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24	Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 *	Fachgruppe Informatik: Jahresbericht 2005
2006-02	Michael Weber: Parallel Algorithms for Verification of Large Systems
2006-03	Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04	Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation
2006-05	Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F
2006-06	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color
2006-07	Thomas Colcombet, Christof Löding: Transforming structures by set interpretations
2006-08	Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs
2006-09	Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking
2006-10	Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11	Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers
2006-12	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08	Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches
2007-09	Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption
2007-10	Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes
2007-11	Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke
2007-12	Uwe Naumann: An L-Attributed Grammar for Adjoint Code
2007-13	Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs
2007-14	Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes
2007-15	Volker Stolz: Temporal assertions for sequential and concurrent programs
2007-16	Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks
2007-17	René Thiemann: The DP Framework for Proving Termination of Term Rewriting
2007-18	Uwe Naumann: Call Tree Reversal is NP-Complete
2007-19	Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control
2007-20	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems
2007-21	Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains
2007-22	Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets
2008-01 *	Fachgruppe Informatik: Jahresbericht 2007
2008-02	Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination
2008-04	Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05	Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations
2008-06	Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07	Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition
2008-08	George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09	George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10	George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11	George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13	Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14	Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15	Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16	Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17	Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving
2008-18	Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2008-19	Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems
2009-02	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03	Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-04	Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation
2009-05	George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06	George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07	Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08	Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09	Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10	Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11	Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12	Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes

2009-13	Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14	Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15	Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems
2009-16	George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17	Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18	Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03	Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04	René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05	Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07	George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08	Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09	George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
2010-10	Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
2010-11	Martin Zimmermann: Parametric LTL Games
2010-12	Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
2010-13	Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems
2010-14	Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
2010-15	Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.