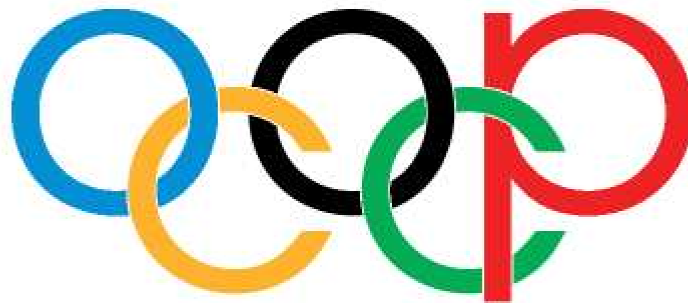

# PG437

## Object-Oriented Proof-Carrying Code


Sven Cordes, Damian Daniluk
Dominik Eisenberg, Achour Elmrabti
Roman Kaczmarczyk, Britta Porsche
Tim Rädisch, Abdelouahab Samet
Said Sghouri, Sebastian Steinfort

Betreuer:

Prof. Dr. Peter Padawitz, Dr. Hubert Wagner

Alle Warenzeichen und eingetragenen Warenzeichen sind Eigentum ihrer jeweiligen Inhaber.

# Inhaltsverzeichnis


**I Überblick** **5**

**1 Zielsetzung** **6**
1.1 Ziele der Projektgruppe . . . . . 6
1.2 Ablauf der PG . . . . . 7

**2 Grundlagen** **9**
2.1 *Expander2* . . . . . 9
2.1.1 Überblick . . . . . 9
2.1.2 Die gesamte Struktur des *Expander2* -Code . . . . . 10
2.1.3 Starten des *Expander2* -Systems . . . . . 11
2.2 PCC . . . . . 15
2.2.1 Überblick . . . . . 15
2.2.2 Definition einer Safety Policy . . . . . 16
2.2.3 Zertifizierung der Sicherheit von Programmen . . . . . 20
2.2.4 Validierung des Safety Proofs . . . . . 24
2.2.5 Zusammenfassung . . . . . 25
2.3 OOPCC . . . . . 26

**II Transformation** **27**

**3 UML** **28**
3.1 XML-Parser . . . . . 28
3.2 UML-Parser . . . . . 29
3.3 UML-Compiler . . . . . 42
3.3.1 Allgemeines Übersetzungsschema von UML . . . . . 43
3.3.2 Aufbau und Funktion des Compilers . . . . . 43

**4 OCL** **48**
4.1 Überblick . . . . . 48
4.2 Grammatik . . . . . 49
4.3 interne Funktionen . . . . . 49
4.3.1 Operationen für alle Kollektionstypen . . . . . 49

4.3.2 Zusätzliche Operationen für Sequenzen . . . . . . . . . . . . . . . . 51
4.3.3 *Expander2* - Umsetzung . . . . . . . . . . . . . . . . . . . . . . . . 51
4.4 Primärparser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.4.1 Initialisierung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.4.2 abstrakte Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.4.3 Parserinitialisierung . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.4.4 Hilfsparser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.4.5 Hauptparser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.5 Sekundärparser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.5.1 Datenstruktur . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.5.2 Arbeitsweise . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.6 Compiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.6.1 Grundlagen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.6.2 Deklaration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.6.3 Hilfsfunktionen . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.6.4 Elementarcompiler . . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.6.5 Invarianten, Preconditions & Postconditions . . . . . . . . . . . . 72
4.6.6 Propertycall . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.6.7 Postfixexpressions . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.6.8 Methodenaufrufe . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.7 Bedienung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

**III Beispiele** **77**

**5 Autorennspiel** **78**
5.1 Einleitung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.2 Die Rennstrecke Version 1 . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.2.1 Umsetzung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.2.2 *Expander2* -Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.3 Entwicklung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.4 Die Rennstrecke Version 2 . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.4.1 Umsetzung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.4.2 *Expander2* -Code . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.5 Beweis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**6 Bank** **86**
6.1 Einleitung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
6.2 Umsetzung und Verwurf . . . . . . . . . . . . . . . . . . . . . . . . . . 86

**7 Mobile Phone System** **88**
7.1 Einleitung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
7.2 Beschreibung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

7.2.1 mobileState . . . . . 89
7.2.2 freqAllocator . . . . . 90
7.3 Probleme . . . . . 90
7.4 Fazit . . . . . 90

**8 H-Bahn** **92**
8.1 Einleitung . . . . . 92
8.2 Beschreibung . . . . . 92
8.2.1 Das Bahnsystem . . . . . 92
8.2.2 bahn und hbahn . . . . . 93
8.2.3 faehrt . . . . . 93
8.2.4 wegfrei, isFrei, frei und check . . . . . 93
8.2.5 fahrschleife . . . . . 94
8.3 Beweis . . . . . 94

**IV Schlusswort** **96**

**9 Zukünftige Arbeit** **97**
9.1 Fehlende Features . . . . . 97
9.2 Ausblick . . . . . 97

**10 Erfahrungen während der PG-Zeit** **99**

**V Anhang** **100**

**A OCL-Grammatik** **101**

# Teil I

# Überblick

# Kapitel 1

# Zielsetzung

## 1.1 Ziele der Projektgruppe

Mitte der 90er Jahre entwickelten Necula und Lee die Idee des proof carrying code (PCC). Die Idee von PCC ist es Programme, und die zu dem Programm gehörenden Korrektheitsbeweise, zu entwickeln. Der Anwender dieser Programme soll mit Hilfe eines proof checkers herausfinden, ob seine Anforderungen, die er an das Programm stellt erfüllt worden sind. Der proof checker wertet Beweisterme aus, d.h. die Beweise werden zuerst in Beweisterme umgeschrieben. Die Vorteile für den Anwender sind leicht zu erkennen: Der Anwender muss seine eigenen Programme nicht mehr verifizieren und er braucht sich, aufgrund des proof checkers, nicht vollständig auf den Produzenten zu verlassen. Der proof checker erkennt außerdem noch wer Änderungen an dem Produkt vorgenommen hat. Bis zum heutigen Zeitpunkt wird PCC nur bei Assemblerprogrammen angewendet, die Korrektheitsbeweise setzten sich deshalb aus Regeln der Zusicherungslogik zusammen. In unserer Projektgruppe wurde der klassische PCC Ansatz nur in der Seminarphase vorgestellt. Ein Ziel der Projektgruppe war es hingegen ein proof checker für objektorientierte (UML-) Spezifikationen für den *Expander2* zu entwickeln. Grundlage dafür war die Programmiersprache O´Haskell. Mit Hilfe des *Expander2* kann man funktionale oder logische Programme, konstruktbasierte Datentypen oder verschiedene Darstellungen zustandsbasierter Systeme verifizieren. Die UML Spezifikationen müssen zuerst in mehrsortige Prädikatenlogik umgeschrieben werden. Der *Expander2* besitzt drei verschiedene Klassen von Inferenzregeln:

- Simplifikationen sind Äquivalenztransformationen die Terme und Formeln so weit wie möglich auswerten.

- Rewriting und Narrowing sind ebenfalls Äquivalenztransformationen. Sie wenden prädikatenlogische Axiome wie funktionale bzw. logische Programme an. Die Axiome werden vorher ausgewählt, und alle Definitionen, Attribute, Methoden und Constraints werden axiomatisiert dargestellt.

- Die Beziehung zwischen Prämisse und Konklusion. Dies ist keine Äquivalenztransformation sondern eine Implikation. Diese Klasse umfasst die Anwendung von Induktions-, Coinduktions-, Generalisierungs- und Spezialisierungsbeweisen.

Für die oben erläuterten Inferenzregeln realisiert *Expander2* die Idee des PCC bereits, d.h. das während der Konstruktion des Beweises automatisch der dazugehörige Beweisterm erzeugt wird.

In unserer Projektgruppe sollten weitere Regeln bzw. Regelkombinationen entwickelt und in den *Expander2* eingebaut werden. Ziel ist es einen Übersetzer von einer baumartigen oder linearen Darstellung in Signaturen bzw. Axiome prädikatenlogischer Spezifikationen, die vom *Expander2* verarbeitet werden können, zu entwerfen und zu implementieren. Gegeben sind UML-Klassendiagramme oder OCL-Constraints. Ein weiteres Minimalziel unserer Projektgruppe ist die Auswahl eines Anwendungsbeispiels und die geeignete Anpassung des Regelsystems und der Beweisdarstellung vom *Expander2* .

## 1.2 Ablauf der PG

Begonnen hat unsere Projektgruppe mit einer Seminarphase. In dieser Zeit wurden insgesamt 6 Vorträge gehalten, die zur Einführung und zur Vertiefung des Verständnisses dienen sollten. Die Vorträge beinhalteten folgende Themen:

- Proof-Carrying Code (PCC) Referenten: Sven Cordes u. Dominik Eisenberg
- UML/OCL Referenten: Britta Porsche u. Ashour Elmrabti
- Algebraische Spezifikation und Verifikation Referent: Damian Daniluk
- Coalgebraische Spezifikation von UML und Swinging UML Referent: Said Sghouri
- Haskell und O'Haskell Referenten: Roman Kaczmarczyk u. Abdelouahab Samet
- Expander2: Beweisfeatures Referenten: Tim Rädisch u. Sebastian Steinfort

Die Seminarphase wurde Anfang November beendet. Nach Abschluß der Seminarphase haben wir uns Beispiele, die man am Ende der PG beweisen kann, überlegt. Die ersten beiden Beispiele befassten sich mit einem Bankkonto und einem Autorennspiel. Bei dem Bankkontobeispiel hatten wir die Idee ein Konto zu verwalten, d.h. ein Konto kann eröffnet werden, es finden verschiedene Transaktionen auf dem Konto statt und der Bankkunde und sein Konto sind eindeutig zu identifizieren. Bei diesem Beispiel gab es jedoch keine relevanten Aussagen die wir beweisen konnten. Wir verwarfen daher dieses Beispiel. Parallel dazu wurde das Autorennbeispiel bearbeitet und alle PG-Teilnehmer haben sich mit dem *Expander2* und dessen Arbeitsweise vertraut gemacht. Erläuterungen zu dem Autorennbeispiel sind weiter hinten im Endbericht beschrieben. Nach dieser Zeit wurde das Autorennbeispiel in UML bzw. OCL umgesetzt. Die OCL Constraints mussten

so umgesetzt werden, dass das Autorennbeispiel in den *Expander2* eingegeben werden konnte.

Im 2. Projektgruppensemester wurde ein weiteres Beispiel, das Mobile Phone System, behandelt. Die Projektgruppe hat sich in drei verschiedene kleine Untergruppen aufgeteilt. Die erste Gruppe beschäftigte sich mit dem Mobile Phone System Beispiel. Hierbei ging es darum das Beispiel zu axiomatisieren und OCL Constraints zu erstellen. Die zweite Gruppe beschäftigte sich mit dem UML Parser und dem UML Compiler. Die dritte Gruppe fertigte den OCL Parser und den OCL Compiler an. Diese Aufgaben deckten das gesamte 2. Semester ab. Am Ende der Projektgruppenzeit wurde nochmals ein weiteres Beispiel entwickelt: Das H-Bahn Beispiel. Bei diesem Beispiel sollten die Beweise eindeutiger und leichter zu führen sein. In den Semesterferien wurde ein Beispielbeweis durchgeführt und der Endbericht angefertigt.

Die gesamte PG traf sich während der zwei Semester montags und mittwochs. Montags gab es ein Gruppentreffen mit den PG-Betreuern. Für jedes dieser Gruppentreffen wurde immer ein Protokoll von jeweils einem der PG- Teilnehmer angefertigt. Weitere Treffen in kleineren Gruppen wurden spontan und mit den jeweiligen PG Teilnehmern vereinbart.

# Kapitel 2

# Grundlagen

## 2.1 *Expander2*

*Expander2* stellt eine Beweis- und Testumgebung für algebraische Datentypspezifikationen und funktional-logische Programme dar.

### 2.1.1 Überblick

Die Hauptkomponenten von *Expander2* sind: der *Solver*, der *Painter*, der *Simplifier*, der *Enumerator* und der *Recorder* von Beweis- und Berechnungssequenzen.

Der Benutzer des *Expander2* steuert den Prozess durch Anwenden bzw. Ändern von z.B. Signaturen, Theoremen, Axiomen oder Knoten durch z.B. Betätigen von Schaltflächen oder Texteingabe. Diese Aktionen stellen Eingaben für den *Solver* dar, die auch diverse Zustandsvariablen ändern können.

Der *Solver* wird nach dem Starten vom *Expander2* über die erscheinende graphische Oberfläche angesprochen. Anhand der graphischen Oberfläche lassen sich die durch Bäume und Graphen repräsentierten Disjunktionen oder Konjunktionen von logischen Formeln oder die Summe von funktionalen Termen eingeben, bearbeiten und darstellen. Eine richtige Summe resultiert aus einer Berechnung, die durch ein nicht deterministisches Rewriting erreicht wird (das Rewriting-Verfahren wird über eine Schaltfläche an der *Solver*-Oberfläche angeboten).

Nachdem ein Widget-Interpreter über das Menü von *pict type* selektiert wurde, verursacht das Drücken der Schaltfläche *Paint* das Öffnen eines *Painter*-Fensters und damit werden die grahischen Darstellungen von allen interpretierbaren Unterbäumen der aktuellen Bäume auf dem *Solver* angezeigt. Bilder sind Listen von Widgets, die über dem *Painter*-Fenster editiert und zu Widget-Graphen vervollständigt werden können.

Der *Solver* und sein entsprechender *Painter* sind völlig synchronisiert: das Selektieren eines Baumes auf dem *Solver*-Fenster wird automatisch zur Selektion der graphischen Darstellung des Baumes übertragen und umgekehrt. Daher können *Rewriting*-, *Narrowing*- und *Simplification*-Schritte gleichzeitig auf dem *Solver*- und *Painter*-Fenster

durchgeführt werden.

Der *Enumerator* stellt Algorithmen zur Verfügung, um Bäume oder Graphen aufzuzählen und deren Ergebnisse zum *Solver* und *Painter* zu übertragen.

*Expander2* erlaubt es dem Benutzer den Verlauf des Beweises bzw. der Berechnung auf drei unterschiedlichen Interaktionsleveln zu steuern bzw. zu beeinflussen.

Auf dem obersten Level werden analytische und synthetische Inferenzregeln sowie syntaktische Transformationen individuell auf lokal ausgewählte Teilbäume angewandt. Die Regeln umfassen zum Beispiel die Anwendung einzelner Axiome, Substitutions- oder Unifikationsschritte, Noethersche Induktion, Hoare-Induktion, Subgoal-Induktion, Fixpunktinduktion oder Co-Induktion.

Im mittleren Level realisieren Narrowing und Rewriting die vollständige Anwendung aller Axiome für die definierten Funktionen, Prädikate und Co-Prädikate der aktuellen Signatur. Das Rewriting endet mit Normalformen, z.B. Terme bestehend aus Konstruktoren und Variablen. Narrowing endet entweder mit True oder mit der Lösung der Ausgangsformel. Sollten die Axiome ein funktional-logisches Programm repräsentieren, so stimmt das Narrowing entlang der Axiome mit der Ausführung des Programms überein.

Das mittlere Level erlaubt es also Programme zu testen, während die Inferenzregeln auf dem obersten Level die Programme „nur“ verifizieren. Alle zulässigen und „passenden“ Transformationen des mittleren und oberen Levels werden in der Zustandsvariable *proof* kontrolliert.

Auf dem untersten Level werden die Terme und Formeln mittels eingebauter Haskell-Funktionen vereinfacht, und soweit automatisiert möglich, ausgewertet. Viele dieser Beweis- und Berechnungsschritte geschehen automatisch und werden nicht explizit aufgeführt.

### 2.1.2 Die gesamte Struktur des *Expander2* -Code

Der Code von *Expander2* besteht aus vier O´Haskell-Modulen:

- *Eterm*: enthält Datentypen und Funktionen zum Generieren, Manipulieren und Prüfen von Termen und Formeln.
- *Epaint*: stellt Haskell-Funktionen zum Parsen von Termen und Formeln dar sowie zur Berechnung und Darstellung ihrer graphischen Darstellungen.
- *Esolve*: ermöglicht die Übersetzung zwischen textueller und graphischer Darstellung von Termen und Formeln. Es verfügt auch über den *Simplifier*, der die partielle Auswertung von Termen und Formeln durchführt.
- *Ecom*: gestaltet die GUI und stellt zum Generieren, Manipulieren oder Übersetzen von Texten und Bäumen alle Befehle dar, die der Benutzer aufrufen kann, um Beweise oder Berechnungen und die interaktive Darstellung ihrer Ergebnisse durchzuführen.

### 2.1.3 Starten des *Expander2* -Systems

Nach der Ausführung der main-Methode startet der *Expander2* , indem zwei Solver und ihre entsprechenden Painter erzeugt werden.

#### Solver-Oberfläche

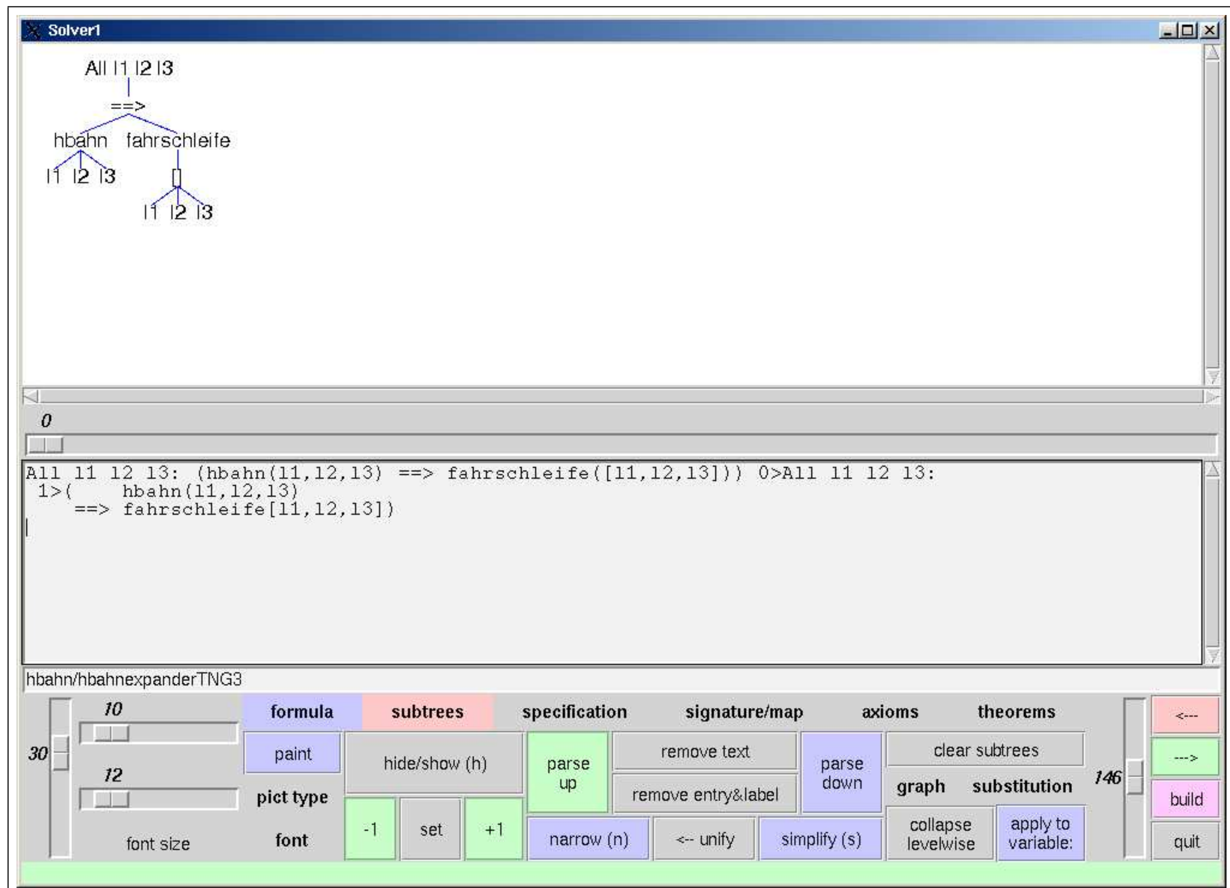


Abbildung 2.1: Oberfläche des *Expander2*

Abwärts enthält die *Solver*-Oberfläche die folgenden Widgets:

- ein scrollbarer graphischer Bildbereich
- ein waagerechter Schieber zum Selektieren des Baumes, der angezeigt werden soll
- ein scrollbares und editierbares Textfeld
- eine Eingabezeile: ein einzeiliges Textfeld für das Eintragen von Dateinamen, Knotenstartadressen der (Teil-)Bäume, ganzen oder rationalen Zahlen

- ein senkrechter und waagerechter Schieber zum horizontalen bzw. vertikalen Ausdehnen und Schrumpfen des Baumes
- ein waagerechter Schieber zum Ändern der Schriftgröße vom Knotennamen
- 9 Menüs in hervorgehobener Schrift, einige werden im Nachhinein beschrieben
- 18 Schaltflächen
- ein vertikaler Schieber zur relativen Änderung der vertikalen Größe zwischen dem graphischen Bildbereich und Textfeld
- ein Statusfeld zum Anzeigen von Hinweisen

Die aktuelle Menge der Bäume und Graphen wird in dem Textfeld in textueller Form und im graphischen Bildbereich in zwei-dimensionaler Form repräsentiert. In dem Textfeld können auch Axiome hineingeschrieben werden, die direkt angewendet werden können. Mit den Schaltflächen „Parse Up“ bzw. „Parse Down“ im Bedienungsteil kann zwischen den beiden unterschiedlichen Repräsentationen, die gleichzeitig editierbar sind, umgeschaltet werden. Dabei ist anzumerken, dass beim Umschalten die gesamte Baumdarstellung reinitialisiert wird, so werden z.B. sämtliche Protokolle gelöscht.

**Einige Zustandsvariablen vom Solver**

Der Zweck der Zustandsvariablen ist es, komplizierte Funktionsparameter zu verbegen. Die aktuellen Werte dieser Parameter werden vom *Simplifier* zum Auswerten der eingebauten interaktiven Haskell-Funktionen benötigt.

- *actions*, *boolFun*, *dissects*, *finals*, *finalsL*, *labels*, *fixPositions*, *matrixU*, *matrixL*, *transitions* und *transitionsL* können als Funktionssymbole in Termen und Formeln auftreten. Ihre Werte werden anhand von *rewrite*-, *simplify*-Schritten abgespeichert bzw. abgerufen und modifiziert. Bei Anwenden des passenden Widget-Interpreters werden sie graphisch dargestellt.
- Die aktuellen Axiome und Theoreme werden auf Vermutungen/Beschränkungen (*conjectures/constraints*) angewandt und bilden die Schritte des oberen oder mittleren Levels einer Berechnung oder eines Beweises. Axiome und Theoreme können mittels Rewriting und Narrowing angewandt werden.
- *curr* enthält die Position des gerade angezeigten Baumes von der Liste der aktuellen Bäume.
- *formula* zeigt an, ob die Liste von aktuellen Bäumen eine Disjunktion oder Konjunktion von Formeln bzw. eine Summe von Termen ist.
- *rule* zeigt an, ob die Liste der aktuellen Bäume aus Narrowing-, Rewriting-, Simplifikationsschritten oder aus anderen Regelanwendungen resultiert.

- *matching* zeigt die aktuelle Strategie an, die für Narrowing/Rewriting verwendet wird.

Nach jeder Regelanwendung wird der aktuelle Beweisterm in das Textfeld so eingetragen, dass der konstante Zeiger (*POINTER*) dem Befehl vorangeht, der zunächst durchgeführt wird. Wenn ein neuer Schritt während der Beweisüberprüfung durchgeführt worden ist, wird der Beweisterm dementsprechend angepasst, d.h. der Rest des zu überprüfenden Beweis wird durch den neuen Schritt ersetzt. Der aktuelle Beweisterm wird zu einer leeren Liste ([]) gesetzt, wann immer der Inhalt des Textfeldes geparsed und folglich zu einer neuen Liste der aktuellen Bäume gemacht wird.

**Einige Menüs**

- Trees Menü: die Befehle des tree-Menüs erzeugen oder wandeln die aktuellen Bäume oder den aktuellen Beweis um.

  *call enumerator*: öffnet ein Untermenü, dass die Algorithmen von Bäumen auflistet.

  *split tree(m)*: zerlegt eine Konjunktion, Disjunktion oder Summe jeweils in ihre Faktoren, Summanden oder Terme, vorausgesetzt dass die Liste von aktuellen Bäumen nur einen Baum enthält.

- Subtrees Menü: die Befehle dieses Menüs wandeln die Unterbäume um, die mit der linken Maustaste selektiert wurden. Wenn kein Unterbaum selektiert wurde, wird davon ausgegangen, dass der gesamte Baum selektiert wurde. Die meisten Befehle rufen Schlussfolgerungsregeln auf und liefern Hinweise, die uns erklären, ob die durchgeführte Regelanwendung bezüglich des Ausgangsmodells stichhaltig ist oder nicht. Dieses Ausgangsmodell wird durch die aktuelle Signatur und Axiome induziert.

- Axioms Menü:

  *remove all*: entfernt das Set von aktuellen Axiomen.

  *show*: trägt alle aktuellen Axiome in das Textfeld ein.

  *load text from*: öffnet ein Untermenü von Dateien. Der Inhalt der selektierten Datei wird in das Textfeld eingetragen.

  *add from*: öffnet ein Untermenü von Dateien. Die in der Datei deklarierten Signaturelemente werden zu der aktuellen Signatur, während Formeln zu der Menge von aktuellen Axiomen hinzugefügt werden. Die Formeln müssen als Disjunktion von höchstens zwei Konjunktionen der bewachten Horn- oder Co-Horn-Klauseln eingegeben werden. Dateien, die nur Signaturelemente enthalten, werden nicht erlaubt.

- Theorems Menü:

  *remove all*: entfernt alle aktuellen Theoreme.

  *save to file*: speichert die aktuellen Theoreme in der Datei ab, deren Name in dem Eingabefeld steht.

- Graph Menü:

  *redraw*: der angezeigte Baum wird neu gezeichnet.

  *expand*: dereferenziert alle Zeiger des angezeigten Baumes oder der selektierten Unterbäume. Wenn das Eingabefeld eine positive Zahl n (Standard: n=0) enthält, wird jeder Kreis in den Bäumen n-mal entfaltet.

**Weitere Schaltflächen**

- *remove text*: entfernt den Text im Textfeld.
- *remove entry & label*: löscht die Einträge im Eingabe- und Statusfeld.
- *narrow/rewrite*: führt Narrowing-/Rewriting-Schritte durch, indem zwischen jeweils zwei Schritten höchstens 100 Simplification-Schritte ausgeführt werden.
- *unify/match*: schaltet zwischen dem initialen unification-Modus, dem matching-Modus und den Greedy-Versionen von diesen Modi um und zwar, wenn Narrowing oder Rewriting auf Formeln bzw. Termen angewandt werden.
- *clear subtrees*: hebt die Selektierung aller selektierten Unterbäume auf.
- *collapse levelwise*: legt die gemeinsamen Unterbäume des aktuellen Baumes Schicht für Schicht zusammen.
- *quit*: beendet den Solver.

## 2.2 PCC

### 2.2.1 Überblick

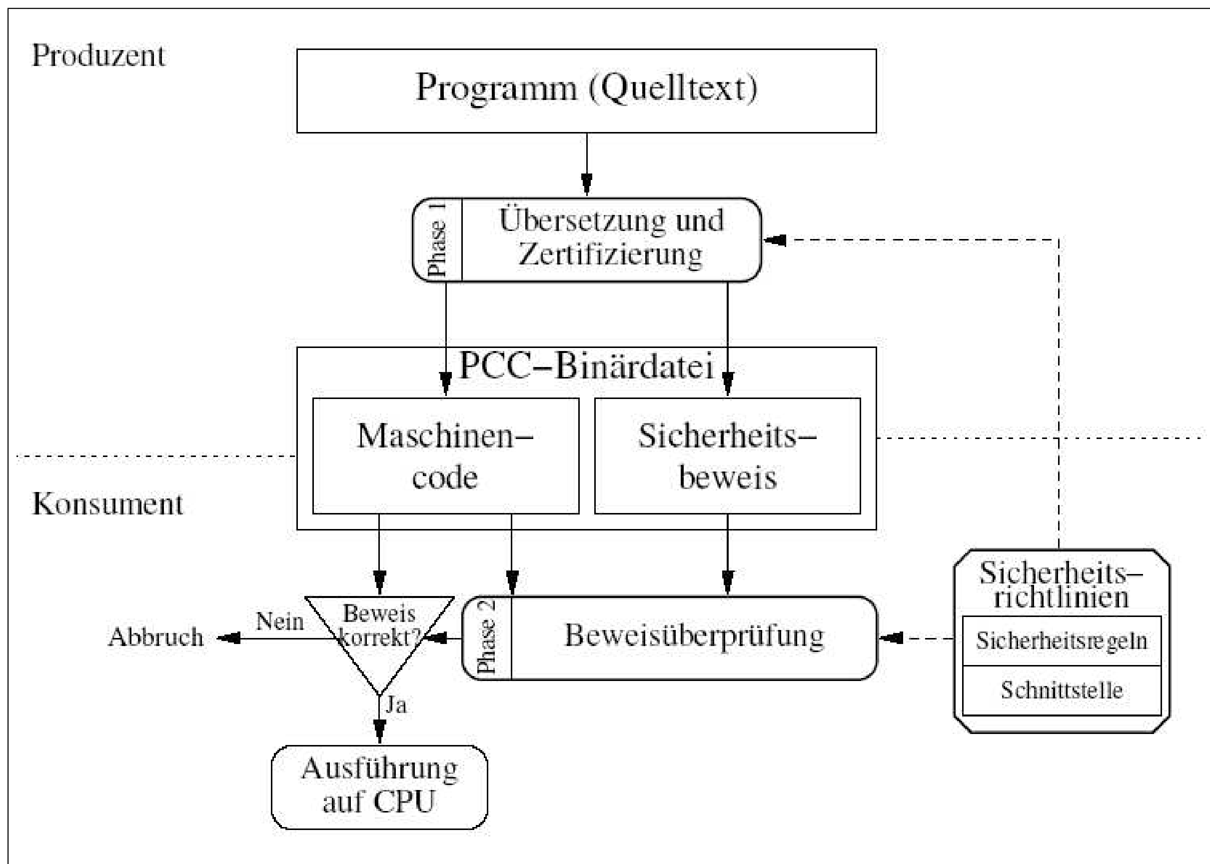


Abbildung 2.2: Übersicht über Proof-Carrying Code (PCC)

Die Idee von Proof Carrying Code (PCC) ist es Programme mit einem Beweis zu versehen. Die sichere Ausführung des Programms wird nachgewiesen, indem der Beweis vor der Programmausführung überprüft wird.

In der Abbildung 2.2 wird der Prozess von der Erstellung bis hin zu der Verwendung einer PCC-Binärdatei (*PCC binary*) beschrieben. Der Prozess beginnt damit, dass der Konsument (*Code Consumer*) Sicherheitsrichtlinien (*Safety Policy*) definiert und publiziert. Die *Safety Policy* definiert formal in den Sicherheitsregeln (*Safety Rules*), was „Sicherheit" bedeutet und spezifiziert eine Schnittstelle (*Interface*)zwischen dem *Code Consumer* und irgend einer vom Produzenten (*Code Producer*) erstellten Binärdatei. In der ersten Phase übersetzt und beweist der *Code Producer* die Sicherheit des Programms unter Berücksichtigung der *Safety Policy*. Dieser Prozess wird Zertifizierung (*Certification*) genannt. Das resultierende *PCC-Binary* wird an den *Code Consumer* geliefert. Nach dem Empfang überprüft der *Code Consumer* in der zweiten Phase den im *PCC Binary* ent-

haltenen Sicherheitsbeweis (*Safety Proof*) gemäß der *Safety Policy*. Dieser Prozess wird Beweisüberprüfung (*Proof Validation*) genannt. Wenn der Beweis korrekt ist, kann der *Code Consumer* den im *PCC Binary* enthaltenen Maschinencode (*Native Code*) sicher ausführen. Im folgenden wird auf die einzelnen Phasen des Prozesses näher eingegangen.

### 2.2.2 Definition einer Safety Policy

In der vom *Code Consumer* festgelegten *Safety Policy* wird definiert, wie sich ein sichereres Programm verhalten muss. Diese besteht aus drei Teilen:

1. Ein *Verification-Condition* Generator, im folgenden VC Generator genannt. Der VC Generator ist eine Funktion, die ein Prädikat in „First-Order“-Logik für das zu zertifizierende Programm berechnet. Dieses Prädikat wird als *Safety Predicate* bezeichnet.

2. Eine Menge von Axiomen, mit denen das *Safety Predicate* bewiesen werden kann.

3. Eine Vorbedingung (*Precondition*). Die *Precondition* ist eine Konvention, die definiert, wie der *Code Consumer* das *PCC Binary* aufzurufen hat.

Man erhält den VC Generator, indem eine abstrakte Maschine spezifiziert wird, welche die Ausführung eines sicheren Programms simuliert. Die abstrakte Maschine wird nicht unbedingt benötigt, sie vereinfacht aber den Entwurf der *Safety Policy* und bietet eine Basis für die Sicherheitsüberprüfung an. Im folgenden Unterabschnitt wird als Beispiel eine abstrakte Maschine vorgestellt, die sicheren Speicherzugriff für den DEC Alpha Prozessor spezifiziert.

#### Eine abstrakte Maschine für DEC Alpha Programme

Das nachfolgende Beispiel benutzt die DEC Alpha Assemblersprache, daher ist die abstrakte Maschine eine formale „High-Level“ Beschreibung der Alpha-Architektur. Um zu sehen, wie die abstrakte Maschine beschrieben wird, sollte man zunächst eine Untermenge der DEC Alpha Assemblersprache betrachten (siehe Abbildung 2.3):

- $n$ ist eine ganze Zahl und $r_i$ ist ein Maschinenregister $i$.

- Alle Instruktionen arbeiten mit 64 Bit Werten.

- Aus Vereinfachung werden nur 11 temporäre „caller-save“ Register verwendet, mit $r_0$ bis $r_{10}$ bezeichnet.

Um zu definieren, wie das Programm ausgeführt werden soll, wird die abstrakte Maschine als Zustandsübergangsfunktion festgelegt (siehe Abbildung 2.4):

- $\Pi$ ist ein Vektor von Instruktionen.

$$
\begin{array}{lcl}
op & ::= & n \mid r_i \quad i \in 0 \ldots 10 \\
al & ::= & \text{ADDQ} \mid \text{SUBQ} \mid \text{AND} \mid \text{OR} \mid \text{SLL} \mid \text{SRL} \\
br & ::= & \text{BEQ} \mid \text{BNE} \mid \text{BGE} \mid \text{BLT} \\
instr & ::= & \text{LDQ}\ r_d, n(r_s) \mid \text{STQ}\ r_s, n(r_d) \mid \ al\ r_s, op, r_d \mid br\ r_s, n \mid \text{RET}
\end{array}
$$

Abbildung 2.3: Die Untermenge der DEC Alpha Assemblersprache

$$
(p,pc) \rightarrow \begin{cases}
(p\,[r_d \leftarrow r_s \oplus op]\,, pc+1)\,, & if\ \Pi_{pc} = \text{ADDQ}\ r_s, op, r_d \\
(p\,[r_d \leftarrow sel\,(r_m, r_s \oplus n)]\,, pc+1)\,, & if\ \Pi_{pc} = \text{LDQ}\ r_d, n(r_s) \quad and \quad \boxed{rd(r_s \oplus n)} \\
(p\,[r_m \leftarrow upd\,(r_m, r_d \oplus n, r_s)]\,, pc+1)\,, & if\ \Pi_{pc} = \text{STQ}\ r_s, n(r_d) \quad and \quad \boxed{wr(r_d \oplus n)} \\
(p, pc+n+1)\,, & if\ \Pi_{pc} = \text{BEQ}\ r_s, n \quad and \quad r_s = 0 \\
(p, pc+1)\,, & if\ \Pi_{pc} = \text{BEQ}\ r_s, n \quad and \quad r_s \neq 0
\end{cases}
$$

Abbildung 2.4: Die abstrakte Maschine

- $\Pi_{pc}$ ist die aktuelle Instruktion, wobei $pc$ der Programm-Counter ist.
- $p$ ist eine Variable und beschreibt den Zustand der Maschinenregister und des Speichers.
- Die Zustandsübergangsfunktion überführt einen Maschinenzustand $(p, pc)$ in einen neuen Zustand $(p', pc')$ durch Ausführung der aktuellen Instruktion $\Pi_{pc}$.
- $p\,[r_i]$ bezeichnet den Inhalt vom Register $r_i$ im Zustand $p$. [1]
- Der Ausdruck $p\,[r_d \leftarrow r_d + 1]$ beschreibt den neuen Zustand, der vom Zustand $p$ durch Erhöhung des Wertes des Registers $r_d$ erhalten wird.

Zum Beispiel hat die Instruktion ADDQ $r_s, op, r_d$ (siehe Abbildung 2.4) die folgende Semantik:

$$(p\,[r_d \leftarrow r_s \oplus op]\,, pc+1)$$

[1] Der Wert eines Registers ist eine ganze Zahl mit dem Wertebereich von 0 bis $2^{64}-1$ in Zweierkomplement-Darstellung.

- $p$ ist der aktuelle Register- und Speicherzustand.
- Durch die ADDQ-Instruktion wird das Register $r_d$ durch die Summe von $r_s$ und $op$ aktualisiert.
- Der Programm-Counter $pc$ wird erhöht.

Das umkreiste Plus-Zeichen $\oplus$ bedeutet eine Zweierkomplement-Addition von 64 Bit Werten. Diese Operation wird durch die gewöhnliche arithmetische Operation für ganze Zahlen wie folgt definiert:

$$e_1 \oplus e_2 = (e_1 + e_2) \bmod 2^{64}$$

Um den Speicher zu modellieren, wird ein Pseudoregister $r_m$ verwendet, welches den Inhalt jeder Speicherstelle wiedergibt. Dabei gilt:

- $sel(r_m, a)$ beschreibt den Inhalt von der Speicheradresse $a$.
- $upd(r_m, a, r_s)$ beschreibt den neuen resultierenden Speicherzustand, der durch das Schreiben des Registers $r_s$ in die Adresse $a$ resultiert.

Die Speicherzugriffe arbeiten auch mit 64 Bit und die angesprochenen Adressen müssen ein Vielfaches von 8 Byte sein (*8-Byte-Aligned*). Bei den Speicherzugriffen gibt es einen entscheidenden Unterschied zwischen der Untermenge der DEC Alpha Assemblersprache (siehe Abbildung 2.3) und der abstrakten Maschine (siehe Abbildung 2.4). Der Unterschied besteht darin, dass die abstrakte Maschine zusätzliche Sicherheitsüberprüfungen bei den Speicherzugriffen durchführt. Betrachten wir zum Beispiel die Definition der
LDQ $r_d, n(r_s)$ Instruktion:

$$(p\,[r_d \leftarrow sel\,(r_m, r_s \oplus n)]\,, pc+1)\,,\ if\ \boxed{rd(r_s \oplus n)}$$

- Das Prädikat $rd(a)$ ist wahr, wenn es sicher ist, den Wert an der Speicheradresse $a$ zu lesen, was für den DEC Alpha Prozessor bedeutet, dass die Adresse $a$ *8-Byte-Aligned* ist.
- Das Prädikat $wr(a)$ ist wahr, wenn die *8-Byte-Aligned* Adresse $a$ sicher gelesen oder geschrieben werden kann.

Die Prädikate $rd(a)$ und $wr(a)$ werden durch die *Safety Policy* über die *Precondition* wie im nächsten Unterabschnitt beschrieben definiert. Mathematisch betrachtet gibt die abstrakte Maschine keine Fehler aus, wenn eine Überprüfung der Prädikate $rd(a)$ und $wr(a)$ scheitert. Stattdessen wird die Ausführung blockiert, da es keine Regeln bei der Zustandsübergangsfunktion für das Behandeln der Fehler gibt. Daher ist ein Programm nur sicher, wenn es ohne eine Blockierung der abstrakten Maschine ausgeführt wird. Das

Vorhandensein der Sicherheitsüberprüfungen bedeutet aber, dass die abstrakte Maschine keine genaue Abstraktion des DEC Alpha Prozessors ist.

Aber es ist die Absicht der Zertifizierung zu beweisen, dass alle Sicherheitsüberprüfungen erfolgreich sind. Wenn wir einen gültigen Sicherheitsbeweis (*Safety Proof*) für ein Programm haben, dann können wir das Programm mit Sicherheit auf einem echten DEC Alpha Prozessor ausführen und erhalten dasselbe Verhalten wie mit der abstrakten Maschine, obwohl der DEC Alpha Prozessor keine Sicherheitsüberprüfungen implementiert hat.

### Anwendungsbeispiel und seine Precondition

Die abstrakte Maschine beschreibt Sicherheit in Termen der abstrakten Notation von lesbaren und beschreibbaren Speicherstellen. Daher muss der *Code Consumer* eine Schnittstelle zu den *PCC Binaries* spezifizieren, die die lesbaren und beschreibbaren Speicher-Positionen kennzeichnet. Dies wird durch eine Spezifizierung einer *Precondition* erreicht, welches ein Prädikat in der *First-Order* Logik ist, wofür der *Code Consumer* die Gültigkeit garantiert, wenn das *PCC Binary* aufgerufen wird.

Als Beispiel wird ein Betriebssystem angenommen, welches eine Tabelle mit Daten, zu verschiedenen Benutzer-Prozessen gehörend, unterstützt. Jeder Tabelleneintrag besteht aus zwei aufeinander folgenden Speichereinträgen (*Memory Words*), das erste Wort heißt *Tag*, das zweite Word heißt *Data*. Der *Tag* beschreibt die Zugriffsrechte. Weiterhin gibt es einen Dienst für Speicherzugriffe (*Resource Access Service*), welcher Prozessen den Zugriff auf ihre Tabelleneinträge durch Installation des *Native Code* im Kernel ermöglicht. Um dies zu ermöglichen weist der Kernel dem installierten Maschinencode (*User Code*) die Adresse des zum Vater-Prozess $r_0$ gehörenden Tabelleneintrag zu. Die Gültigkeit und das *8-Byte-Aligned* werden durch den Kernel garantiert. Der *User Code* soll vollen Zugriff auf den Tabelleneintrag haben, der Rest der Tabelle und der komplette Kernelstatus müssen geschützt sein.

Informell erfordert die *Safety Policy* für den *Resource Access Service* die folgende Eigenschaften:

1. Der *User Code* kann nicht auf andere Tabelleneinträge zugreifen, außer dem vom Register $r_0$ verwiesenen Tabelleneintrag.

2. *Tag* kann nur gelesen werden.

3. *Data* kann auch nur gelesen werden, solange der *Tag* ungleich Null ist.

4. Die letzte Bedingung stellt sicher, dass *caller Save* Register und reservierte Register das Programm nicht modifizieren können.

Formaler, der Kernel spezifiziert eine *Precondition* $Pre_r$, die angibt, dass es sicher ist, den *Tag*, auf den $r_0$ zeigt, zu lesen, und dass es auch sicher ist, *Data* an der Stelle mit

dem Offset 8 von $r_0$ zu schreiben, wenn der Inhalt von *Tag* ungleich 0 ist. In der formalen Darstellung wird die *Precondition* wie folgt beschrieben:

$$Pre_r = r_0 \bmod 2^{64} = r_0 \wedge rd(r_0) \wedge rd(r_0 \oplus 8) \wedge sel(r_m, r_0) \neq 0 \Rightarrow wr(r_0 \oplus 8)$$

Es bleibt zu beweisen, dass für einen bestimmten Client des *Resource Access Service* alle Überprüfungen der Prädikate $rd(a)$ und $wr(a)$ für die gegebene *Precondition* und abstrakte Maschine erfolgreich sind. Im Allgemeinen kann auch eine Nachbedingung (*Postcondition*) als Teil der *Safety Policy* spezifiziert werden, welche bestimmte gültige Invarianten erfordert, wenn der *User Code* beendet wird. Bei diesem Beispiel ist die *Postcondition* das Prädikat *True* und bedeutet, dass keine zusätzlichen Bedingungen den endgültigen Maschinenzustand beeinflussen. Bevor in dem nächsten Unterabschnitt zu der Zertifizierung übergegangen wird, soll an dieser Stelle festgehalten werden, dass die *Safety Policy*, die hier beschrieben wurde, eine tief greifende Speichersicherheit erzwingt.

### 2.2.3 Zertifizierung der Sicherheit von Programmen

Um den *Safety Proof* für ein Programm zu erzeugen, muss bewiesen werden, dass die Ausführung keine Sicherheitsüberprüfungen verletzt und die *Postconditions*, falls gegeben, ebenfalls erfüllt sind. Es gibt Standardtechniken, mit denen solche Beweise erzeugt werden können. Im Allgemeinen wird eine Technik verwendet, die auf Floyds Verifikationsbedingungen (*Floyd Verification Condition*) basiert. Diese Technik ist leistungsfähig genug, um unstrukturierte Assemblerprogramme und Sicherheitsinvarianten behandeln zu können.

Die Zertifizierung erfolgt in den folgenden beiden Schritten:

1. Das *Safety Predicate* wird für das Programm berechnet, welches die semantische Bedeutung in logischer Form beschreibt und einen formalen Ausdruck erzeugt, dass bei der Ausführung des Programms keine Sicherheitsverletzungen auftreten.

2. Der *Safety Proof* für das *Safety Predicate* wird in einer formal prüfbaren Form erzeugt.

Nachfolgend werden die beiden Schritte erläutert.

#### Berechnen des Safety Predicate

Das *Safety Predicate* eines Programms ist eine Funktion vom Programm selbst, der *Precondition* und der *Postcondition*, die der *Code Consumer* definiert, und von der Menge der Invarianten, die im Programm enthalten sind. Zur Berechnung des *Safety Predicate* wird ein Vektor $VC$ von Prädikaten (*Verification Condition Vector*), für jede Instruktion ein Prädikat nach der Regeln in der Abbildung 2.5, erstellt:

- $VC_{pc}$ bezeichnet das Prädikat für die aktuelle Anweisung.

$$VC_{pc} \rightarrow \begin{cases} VC_{pc+1}\,[r_d \leftarrow r_s \oplus op]\,, & if\ \Pi_{pc} = \text{ADDQ}\ r_s, op, r_d \\ rd(r_s \oplus n) \wedge VC_{pc+1}\,[r_d \leftarrow sel\,(r_m, r_s \oplus n)]\,, & if\ \Pi_{pc} = \text{LDQ}\ r_d, n(r_s) \\ wr(r_d \oplus n) \wedge VC_{pc+1}\,[r_m \leftarrow upd\,(r_m, r_d \oplus n, r_s)]\,, & if\ \Pi_{pc} = \text{STQ}\ r_s, n(r_d) \\ (r_s = 0 \Rightarrow VC_{pc+n+1}) \wedge (r_s \neq 0 \Rightarrow VC_{pc+1})\,, & if\ \Pi_{pc} = \text{BEQ}\ r_s, n \\ Post\,, & if\ \Pi_{pc} = \text{RET} \end{cases}$$

Abbildung 2.5: Der *Verification Condition* Generator

- Das Prädikat $VC_0$ für den Programmanfang kann durch Rückwärtsdurchlauf des Programms vom Programmende berechnet werden, weil die Regeln $VC_{pc}$ in Ausdrücken von $VC_{pc+1}$ spezifizieren.

Die Regeln in Abbildung 2.5 werden direkt von der Spezifikation der abstrakten Maschine abgeleitet. Tatsächlich wird die Spezifikation der abstrakten Maschine von erfahrenen Kernel- und *Safety Policy*-Designer außer Acht gelassen und nur die Regeln des *Verification Condition* Generators angegeben. Die Notation $P[r_d \leftarrow r_s \oplus op]$ steht für das Prädikat $P$, dass durch die Substitution von $r_d$ durch $r_s \oplus op$ erhalten wird.

Nachdem man den *Verification Condition Vector* berechnet hat, wird das *Safety Predicate* durch Einfügen des Programms Π, der *Precondition Pre* und der *Postcondition Post* in die folgende Formel berechnet:

$$SP(\Pi,\ Pre,\ Post) = \forall r_0 \ldots \forall r_{10} \forall r_m.\ Pre \Rightarrow VC_0$$

Die Absicht hinter einem gültigen *Safety Predicate* ist, dass für jeden initialen Zustand, der die *Precondition* erfüllt, dass das Programm Π bei der ersten Instruktion startend ohne Blockierung ausgeführt wird und, wenn es beendet wird, der Endzustand die *Postcondition* erfüllt.

Als konkretes Beispiel für ein Client Programm für den *Resource Access Service* wird das kleine Programm in Abbildung 2.6 betrachtet. Die Aufgabe dieses Programms ist es, den Wert von *Data* zu erhöhen, wenn *Data* beschreibbar ist. Zuerst wird $VC_0$ für dieses Programm unter Benutzung der Regeln in Abbildung 2.5 berechnet. Danach wir das *Safety Predicate* mit der oben genannten Formel berechnet, indem die *Precondition Pre* und die gültige *Postcondition* $Post = true$ eingesetzt werden. Nach einigen Vereinfachungsschritten lässt sich das *Safety Predicate* folgendermaßen darstellen:

$$\begin{aligned} SP_r = \quad & \forall r_0.\forall r_m.\ Pre_r \Rightarrow rd(r_0 \oplus 8) \wedge rd(r_0 \oplus 8 \ominus 8) \\ & \wedge\ sel(rm, r_0 \oplus 8 \ominus 8) = 0 \Rightarrow true \\ & \wedge\ sel(rm, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8) \end{aligned}$$

| | | | |
|---|---|---|---|
| 1 | ADDQ | $r_0, 8, r_1$ | // Zeiger auf tag in $\mathrm{r}_0$, Zeiger auf data in $\mathrm{r}_1$ |
| 2 | LDQ | $r_0, 8(r_0)$ | // data in $\mathrm{r}_0$ |
| 3 | LDQ | $r_2, -8(r_1)$ | // tag in $\mathrm{r}_2$ |
| 4 | ADDQ | $r_0, 1, r_0$ | // inkrementiere data in $\mathrm{r}_0$ um 1 |
| 5 | BEQ | $r_2, \mathrm{L}_1$ | // verzweige, wenn tag $==$ 0 |
| 6 | STQ | $r_0, 0(r_1)$ | // schreibe data |
| $\mathrm{L}_1$ | RET | | // Fertig |

Abbildung 2.6: DEC Alpha Assemblerprogramm für Speicherzugriffe

Das Prädikat $SP_r$ sagt aus, dass für alle Werte des Registers $r_0$ und alle Speicherinhalte $r_m$, die die Precondition $Pre_r$ erfüllen, die Speicherstellen $r_0 \oplus 8$ und $r_0 \oplus 8 \ominus 8$ lesbar sein müssen. Weiterhin muss *Data* an der Speicherstelle $r_0 \oplus 8$ beschreibbar sein, falls der *Tag* an der Speicherstelle $r_0 \oplus 8 \ominus 8$ nicht Null ist. Alle diese Bedingungen müssen *true* sein, um die *Safety Policy* des *Resource Access Service* einzuhalten. Hiermit wurde der erste Schritt der Zertifizierung erledigt und im nächsten Unterabschnitt wird die Berechnung des *Safety Proofs* gezeigt.

**Beweisen des Safety Predicate**

Das Programm in Abbildung 2.6 wurde in einer etwas komplizierteren Weise geschrieben, um zu zeigen, dass *Low-Level* Optimierungen keine bedeutenden Probleme bei der Erzeugung und Validierung des *Safety Proof* verursachen. Einige wichtige Eigenschaften des Programms sind:

1. Die Anweisungen beinhalten spekulative Ausführungen des Speicherzugriffs in Zeile 2 und der Addition in Zeile 4, um der DEC Alpha *pipeline latency*[1] Rechnung zu tragen.

2. Das Register $r_0$ in Zeile 2 wird verwendet, um *Data* anstelle der Adresse von *Tag* zu halten.

3. Obwohl die *Precondition* als Werte-Funktion in Register $r_0$ ausgedrückt ist, erfolgen einige der aktuellen Speicher-Zugriffe (*Memory Accesses*) durch das Register $r_1$.

Wenn man sich mit der Assemblersprache auskennt, kann man nachvollziehen, dass das Programm korrekt ist und die *Safety Policy* eingehalten wird. Das Problem ist nun, wie man den misstrauischen Kernel überzeugt, dass der Code absolut sicher ist. Um dies zu tun, muss das *Safety Predicate* geprüft werden. Diese Überprüfung verläuft nach den Regeln des *First-Order* Prädikaten-Kalküls, das um die Zweierkomplement-Arithmetik für ganze Zahlen erweitert wurde. Hier sind zwei der verwendeten Regeln:

[1]Diese Operationen sind spekulativ, da sie nicht benötigt werden, wenn die Verzweigung ausgeführt wird.

$$\vdash_\Sigma Q, \qquad\qquad if \vdash_\Sigma P \Rightarrow Q \quad and \quad \vdash_\Sigma P$$

$$\vdash_\Sigma e_1 \oplus e_2 \ominus e_2 = e_1, \quad if \vdash_\Sigma e_1 \bmod 2^{64} = e_1$$

- $\Sigma$ ist die Menge Beweisregeln
- $\vdash_\Sigma SP$ bedeutet, dass das *Safety Predicate SP* basierend auf der Menge der Beweisregeln $\Sigma$ bewiesen werden kann

Die erste Regel beinhaltet die implizierte Elimination (*Implication Elimination*). Die zweite Regel ist ein wenig überraschend, weil $e_1 + e_2 - e_2 = e_1$ in der Arithmetik für die ganzen Zahlen unbedingt zutreffend ist. Bei der maschinellen Implementierung dieser Arithmetik ist diese Aussage aber nur zutreffend, wenn der ursprüngliche Wert von $e_1$ ein gültiger Registerwert ist.

$$\frac{\dfrac{\begin{matrix}Pre_r\\ \vdots\\ rd(r_0)\end{matrix} \quad \begin{matrix}Pre_r\\ \vdots\\ \dfrac{r_0 \bmod 2^{64}=r_o}{r_0=r_0\oplus 8\ominus 8}\end{matrix}}{rd(r_0\oplus 8\ominus 8)} \qquad \dfrac{\begin{matrix}Pre_r\\ \vdots\\ sel(r_m,r_0)\neq 0 \Rightarrow wr(r_0\oplus 8)\end{matrix} \quad \dfrac{\dfrac{u}{sel(r_m,r_0\oplus 8\ominus 8)\neq 0} \quad \begin{matrix}Pre_r\\ \vdots\\ \dfrac{r_0 \bmod 2^{64}=r_o}{r_0=r_0\oplus 8\ominus 8}\end{matrix}}{sel(r_m,r_0)\neq 0}}{\dfrac{wr(r_0\oplus 8)}{sel(r_m,r_0\oplus 8\ominus 8)\neq 0\Rightarrow wr(r_0\oplus 8)}\,u}}{\dfrac{\dfrac{rd(r_0\oplus 8\ominus 8)\wedge(sel(r_m,r_0\oplus 8\ominus 8)\neq 0\Rightarrow wr(r_0\oplus 8))\wedge\ldots}{Pre_r\Rightarrow rd(r_0\oplus 8\ominus 8)\wedge(sel(r_m,r_0\oplus 8\ominus 8)\neq 0\Rightarrow wr(r_0\oplus 8))\wedge\ldots}\,Pre_r}{\forall r_0.\forall r_m.\ Pre_r\Rightarrow rd(r_0\oplus 8\ominus 8)\wedge(sel(r_m,r_0\oplus 8\ominus 8)\neq 0\Rightarrow wr(r_0\oplus 8))\wedge\ldots}}$$

Abbildung 2.7: Teil eines formalen *Safety Proof* von $SP_r$

In der Abbildung 2.7 ist ein großer Teil des Beweises des *Safety Predicate* von dem Programm in Abbildung 2.6 dargestellt.

Der Beweisbaum (*Proof tree*) wird von oben nach unten gelesen, dabei wird jeder Knoten als gültige Folgerung des Prädikats unterhalb des Strichs interpretiert, der die Annahmen oberhalb des Strichs berücksichtigt.

Beispielsweise wird das Prädikat $r_0 = r_0 \oplus 8 \ominus 8$ in der oberen rechten Ecke unter Verwendung der arithmetischen Regel mit der aus der *Precondition* extrahierten Annahme $r_0 \bmod 2^{64} = r_0$ bewiesen. Dann wird $wr(r_0 \oplus 8)$ mit der *Implication Elimination* Regel und der Hypothese $u$ von dem Prädikat $sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0$ bewiesen. Die Hypothese $u$ wird weiterhin in einer unteren Ebene des Beweis-Baumes verwendet, um das Prädikat $sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)$ zu beweisen.

### Die Garantie der Sicherheit

Der Beweis des *Safety Predicate* wird als Beweis dafür benutzt, dass der Code die *Safety Policy* befolgt. Dies wird in dem folgenden Theorem beschrieben:

**Theorem (*Safety*)** Für jedes Programm $\Pi$, Precondition *Pre* und Postcondition *Post*, wenn $\vdash_\Sigma \ SP(\Pi, Pre, Post)$ gilt, dann sind für jeden initialen Zustand $p_0$, der die Precondition erfüllt, und für jeden aus dem initialen Zustand $(p_0, 0)$ erreichten abstrakten Maschinenzustand $(p, pc)$ eine der folgenden Aussagen wahr:

1. Der Zustand $(p, pc)$ ist ein Endzustand, der die Postcondition erfüllt, oder
2. Die Ausführung ist nicht angehalten, z.B gibt es einen neuen Zustand $(p', pc')$, so dass $(p, pc) \to (p', pc')$ gilt.

Wenn die abstrakte Maschine aufgrund einer Verletzung einer Sicherheitsüberprüfung stehen bleibt, dann liefert dieses Theorem eine absolute Garantie dafür, dass ein zertifiziertes Programm keine solcher Verletzungen hat, solange seine Ausführung in einem Zustand beginnt, das die Precondition erfüllt.

### 2.2.4 Validierung des Safety Proofs

Die *Proof Validation* wird für ein *PCC Binary* nur einmal ausgeführt und zwar unabhängig von der Zahl der Programmausführungen. Um die *PCC Binary* zu validieren, extrahiert der *Code Consumer* zunächst den *Native Code* und berechnet dann das *Safety Predicate* mit einem *Proof Checker*, der die *Verification Condition*-Regeln anwendet. Am Ende überprüft der *Code Consumer*, ob der *Safety Proof* ein gültiger Beweis für das berechnete *Safety Predicate* ist.

Diese Methode gewährleistet auch dann Sicherheit, wenn der *Native Code* oder der *Safety Proof* in dem *PCC Binary* verfälscht wurden. Im Fall der Änderung des *Native Codes* wird sich mit aller Wahrscheinlichkeit auch das *Safety Predicate* ändern, so dass der *Safety Proof* diesem *Safety Predicate* nicht mehr entspricht. Wird aber der *Safety Proof* modifiziert, dann ist der *Safety Proof* entweder ungültig oder entspricht wieder nicht dem *Safety Predicate*. Wird aber der *Native Code* so modifiziert, dass das *Safety Predicate* unverändert bleibt, oder wenn der *Native Code* und der Beweis alle beide geändert wurden, so dass der *Safety Proof* für das neue *Safety Predicate* noch gültig ist, dann verläuft die Validierung erfolgreich und man erhält weiterhin eine Garantie für die Sicherheit.

Um die Validierung zu automatisieren, muss eine konkrete Darstellungssprache für die Prädikate und ihre Beweise ausgewählt werden. Hier wurde das *Edinburgh Logical Framework* LF verwendet. LF ist eine Verlängerung des einfach geschriebenen Lambda Kalküls und wurde als Meta-Sprache für *High-Level* Spezifikation von Sprachen in der Logik entworfen. Die attraktivste Eigenschaft von LF ist, dass es einen leistungsfähigen und einfachen *Type Checking* Algorithmus hat, der für die Validierung der *Safety Proofs* benutzt werden kann.

Die Prädikate und die Beweise werden in LF dargestellt, so dass die Validierung eines Beweises durch die gültige Typisierung der Beweisdarstellung impliziert wird. Die Beweisüberprüfung erfolgt dann mittels *Type Checking*. Die Implementierung des *Type*

*Checking* ist so einfach, dass diejenigen Programmierer, die nicht einer öffentlich vorhandenen *Type Checking* Implementierung vertrauen, sich ihren eigenen *Proof Checker* programmieren können.

### 2.2.5 Zusammenfassung

Jede PCC-Implementierung besteht mindestens aus vier Elementen:

1. eine formale Sprache, um die *Safety Policy* zu spezifizieren
2. eine formale Semantik bzw. Repräsentation des zu beweisenden Codes
3. eine formale Sprache, um Beweise zu führen
4. einen Algorithmus zur Beweisüberprüfung

Die PCC-Vorteile lassen sich in den folgenden Punkten zusammenfassen:

1. der *Code Consumer* muss nicht mehr dem *Code Producer* vertrauen, sondern nur noch einem *Proof Checker*. Daher muss er auch nicht wissen oder verstehen wie die Beweise konstruiert sind, oder sie gar selbst führen.
2. PCC ist von der Programmiersprache unabhängig und somit sehr flexibel.
3. PCC lässt sich effizient durchführen, weil der Code nicht interpretiert werden muss und keine Virtuelle Maschine zur Ausführung benötigt wird.
4. Durch PCC kann die Prüfung des Codes vollautomatisch realisiert werden.

PCC hat die folgenden Nachteile:

1. Der Ablauf bis zur endgültigen Ausführung ist sehr komplex.
2. Alle Sicherheitsbedrohungen können nicht durch PCC überprüft werden.
3. Der Beweis kann mit hohem Aufwand verbunden sein und möglicherweise nicht durchgeführt werden.

## 2.3 OOPCC

Bei OOPCC handelt es sich um eine Erweiterung der PCC-Idee. Es sollen nun mehr nicht Beweise für sicheres Verhalten erstellt und verifiziert werden, sondern Beweise für korrektes Verhalten für objektorientierte Programme.

Die PG befasst sich hier jedoch nicht mit konkreten Programmen einer oder mehrerer konkreter Sprachen, sondern nimmt als Basis Modelle von Programmen, die durch UML-Klassendiagramme und OCL-Constraints beschrieben sind. Diese werden dann durch den erstellten Compiler in eine logische Repräsentation überführt.

Entsprechend der PCC-Grundlage, beinhaltet auch OOPCC die vier Grundelemente:

1. Expander-Logik, als eine formale Sprache um die *Safety Policy* zu spezifizieren
2. UML und OCL, als eine formale Semantik bzw. Repräsentation des zu beweisenden Codes
3. Expander-Logik, als eine formale Sprache um Beweise zu führen
4. Expander, als einen Algorithmus zur Beweisprüfung

# Teil II

# Transformation

# Kapitel 3

# UML

Um die relevanten Teile für eine *Expander2* - Logik aus einem UML-Diagramm zu extrahieren, war zunächst eine geeignete textuelle Darstellung eines UML-Diagramms zu wählen, welche sich effizient parsen lässt. In der Diskussion waren Human Readable UML und XML. Die Wahl fiel auf die XML-Darstellung, da gängige UML-Modelling-Tools UML-Diagramme in dieser Form exportieren können. Weiterhin weist XML eine baumartige Struktur auf, welche das Parsen erleichtert. Im Speziellen wurde die UML-XMI-Darstellung, in der aktuellen Version 1.4, gewählt.

Der Aufbau des Parsers wurde in zwei Segmente unterteilt, einen XML-Parser und einen UML-Parser. Der XML-Parser liest im Wesentlichen die XML-Struktur ein und überträgt diese in eine erste Hilfsstruktur. Der UML-Parser extrahiert daraus alle relevanten Informationen und überführt diese in eine verfeinerte Datenstruktur, welche als Ausgangspunkt für den UML-Compiler dient. Im Compiler wird dann aus dieser Struktur der UML-Teil der *Expander2* - Logik erzeugt. Zur Umsetzung wurde die funktionale Programmiersprache Haskell verwendet. Diese Bausteine werden in den folgenden Abschnitten genauer beschrieben.

## 3.1 XML-Parser

Der XML-Parser liest die XML-Struktur ein, welche aus einer Menge von XML-Knoten besteht. Ein Knoten setzt sich aus einem Knotenbezeichner und einer Menge von XML-Attributen zusammen und beinhaltet optional entweder reinen Text oder eine Menge von Kindknoten. Der XML-Parser überliest eventuelle Kommentare und erkennt Fehler in der XML-Syntax.

Der Parser ist monadisch aufgebaut und liefert bei korrektem Aufbau der XML-Datei ein Ergebnis folgender Struktur:

```
1 -- Definition eines XML-Knotens
2 data XmlNode
3  = Node Nodename [Attribute] [XmlNode]
4  | TextNode Nodename [Attribute] Text deriving(Eq,Show,Read)
```

```
5 type Attribute = (String, String)
6 type Nodename = String
7 type Text = String
```

*Ein Beispiel:*
Aus

```
1 <UML:Model xmi.id = 'S.1' name = 'Project' visibility = 'public'>
2   <UML:Class xmi.id = 'S.5' name = 'Klassenname' visibility = 'public'
        isSpecification = 'false' isAbstract = 'false' isActive = 'false
        '/>
3 </UML:Model>
```

wird

```
1 Model [("xmi.id","S.1"),("name","Project"),("visibility","public")] [
    Class [("xmi.id","S.5"),("name","Klassenname"),("visibility","
    public"),("isSpecification","false"),("isAbstract","false"),("
    isActive","false")] [] ]
```

Auf den Knotenbezeichner folgt die Attributliste und dann die Kindknotenliste. In diesem Fall hat der „UML:Model-Knoten“ einen Kindknoten, den „UML:Class-Knoten“. Dieser hat keine Kindknoten und somit eine leere Kindknotenliste. Die XML-Attribute tauchen als Tupel von zwei Strings (Attributbezeichner,Attributwert) in der Attributliste auf. Es gibt zwei Möglichkeiten einen Knoten zu schließen: „/>“ schließt den „UML:Class-Knoten“ direkt (nur möglich, wenn der Knoten keine Kindknoten und keinen Text enthält), während „</UML:Model>“ den „UML:Model-Knoten“ beendet. Dass jeder Knoten ordnungsgemäß geschlossen wird, ist bei einem korrekten Parserdurchlauf sichergestellt.

## 3.2 UML-Parser

Der UML-Parser erhält als Ausgangspunkt die vom XML-Parser erzeugte XmlNode-Datenstruktur und liefert im Ergebnis eine Liste von Klassen. Dabei geht der UML-Parser in zwei Schritten vor, um zum Ergebnis zu gelangen. Im ersten Schritt wird eine Hilfsstruktur erzeugt, mit deren Hilfe sich im zweiten Schritt die Zielstruktur (Liste von Klassen) erstellen lässt. Abbildung 3.1 veranschaulicht diesen Sachverhalt. Die Hilfsstruktur stellt ein 4-Tupel dar und setzt sich zusammen aus:

- einer Liste von Klassen
- einer Liste von Interfaces
- einer Liste von ID/Name - Tupeln, welche der Zuordnung von Ids zu Namen dienen
- einer Liste von Assoziationen

Die genaue Definition der Hilfsstruktur *Aux* ist nachfolgend aufgeführt.

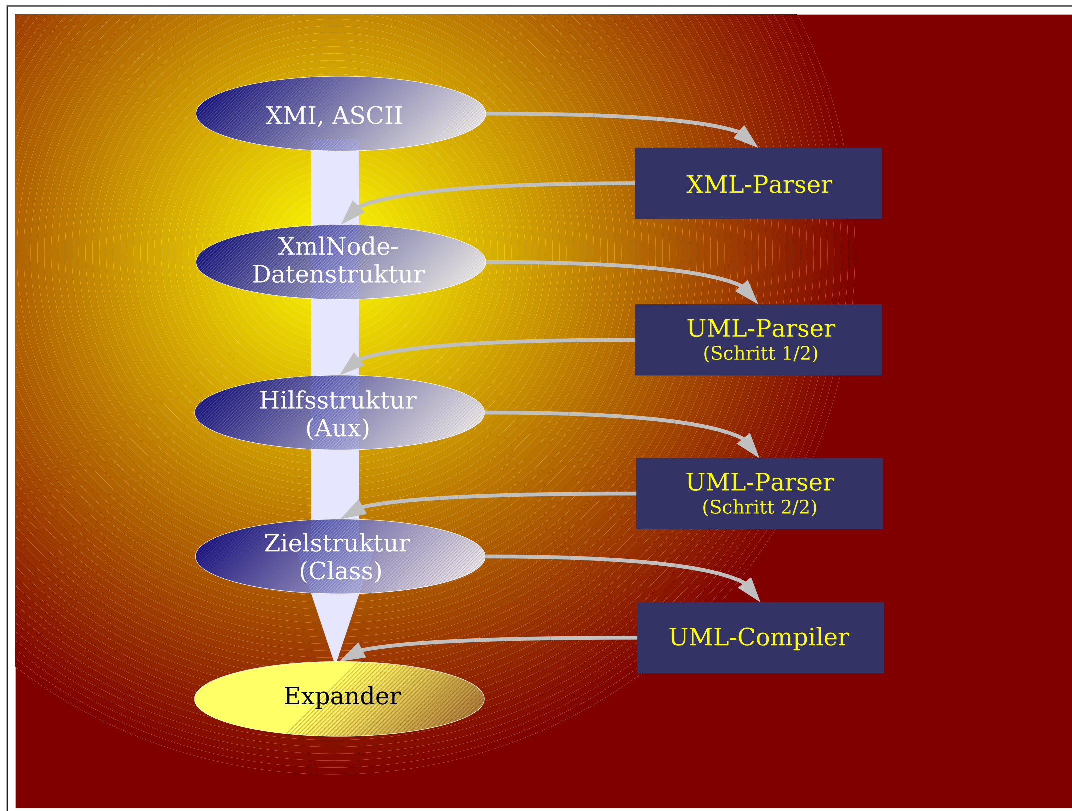


Abbildung 3.1: Parser- bzw. Compiler-Durchlauf

```
1 -- Definition der Hilfsstruktur
2 type Aux = (InterfaceTable, ResolveTable, AssocTable, ClassTable)
```

Im Folgenden soll nun auf die Umsetzung des UML-Parsers in Haskell eingegangen werden. Dabei wird auf alle zum Aufbau der Hilfsstruktur notwendigen UML-Konstrukte in XMI-Form eingegangen und erläutert, welche Konstrukte in welcher Form in die Hifsstruktur aufgenommen werden.

Der UML-Parser bekommt als Eingabe den vom XML-Parser eingelesenen XML-Knoten, welcher den Inhalt der gesamten XMI-Datei bis auf den Header der Form

```
1 <?xml version = '1.0' encoding = 'ISO8859_1' ?>
```

enthält. Da die zu betrachtenden, relevanten XML-Knoten von einigen irrelevanten Knoten umschlossen sind, wird die Funktion *getInnermostRelevantXmlNode* verwendet.

```
1 getInnermostRelevantXmlNode :: XmlNode -> XmlNode
2 getInnermostRelevantXmlNode xml = path xml ["XMI.content", "UML:Model"
    , "UML:Namespace.ownedElement"]
3
4 path :: XmlNode -> [String] -> XmlNode
5 path n [] = n
6 path n (x:xs) = path (getChild n x) xs
```

```
7
8 getChild :: XmlNode -> String -> XmlNode
9 getChild (Node _ _ childs) name = gc childs
10   where gc ((n@(Node na _ _)):rest) | na==name = n
11                                     | otherwise = gc rest
12         gc ((TextNode _ _ _):rest) = gc rest
```

Das Ergebnis eines Aufrufs von *getInnermostRelevantXmlNode* ist der XML-Knoten, welcher alle relevanten Knoten enthält. Die Hauptfunktion zum Erzeugen der Hilfsstruktur *createAux* benutzt die Funktion *getInnermostRelevantXmlNode*, um die Funktion *parseChilds* mit den notwendigen XML-Knoten aufzurufen. *parseChilds* nimmt eine Schlüsselrolle im UML-Parser ein, denn sie durchläuft alle ihr übergebenen auf einer Ebene liegenden XML-Knoten und ruft je nach Knotentyp die entsprechende Parse-Funktion auf, wodurch sukzessive die vier Tabellen der Hilfsstruktur gefüllt werden.

```
1 createAux :: XmlNode -> Aux
2 createAux xml = (parseChilds c ([],[],[],[]))
3     where (Node _ _ c) = (getInnermostRelevantXmlNode xml)
4
5 parseChilds :: [XmlNode] -> Aux -> Aux
6 parseChilds [] t = t
7 parseChilds (n@(Node name _ _):ns) t = parseChilds ns parse
8   where parse | (name == "UML:Class") = parseClass n t
9               | (name == "UML:Stereotype") = parseNameIdRelation n t
10              | (name == "UML:DataType") = parseNameIdRelation n t
11              | (name == "UML:Association") =
12                    parseAssociation (path n ["UML:Association.
                          connection"]) t
13              | (name == "UML:Interface") = parseInterface n t
14              | otherwise = t
```

Betrachten wir zunächst die Funktion *parseClass*. Diese Funktion hat die Aufgabe die Tabelle *ClassTable* zu füllen.

```
1 type ClassTable = [(String,String,[Element])]
2                     -- Id, Name, Elementliste
3
4 data Element
5   = Method String String [(String,String)] String
6   | Attribute String String -- Name, Typ
7   | Association String String Range
8   | Ancestor String -- Oberklassenname
9   deriving(Eq,Show,Read)
```

Eine Klasse besteht aus einer Id, einem Namen und einer Liste von Elementen. Ein Element kann dabei eine Methode, ein Attribut, eine Assoziation oder auch der Name der Oberklasse, von welcher geerbt wird, sein.

Eine Methode setzt sich aus dem Methodennamen, einem Stereotyp, einer Liste von Parametern und dem Rückgabetyp zusammen. Die Parameterliste enthält dabei Tupel, welche aus Parameternamen und dem Parametertyp gebildet werden.
Die Assoziation ist definiert über ihren Rollennamen, den Typ und Angaben zu ihren Kardinalitäten. Wir betrachten nun ein Beispiel, um die Arbeitsweise von *parseClass* einfacher nachvollziehen zu können. An diesem Beispiel sollen auch alle folgenden Funktionen des UML-Parsers erläutert werden.

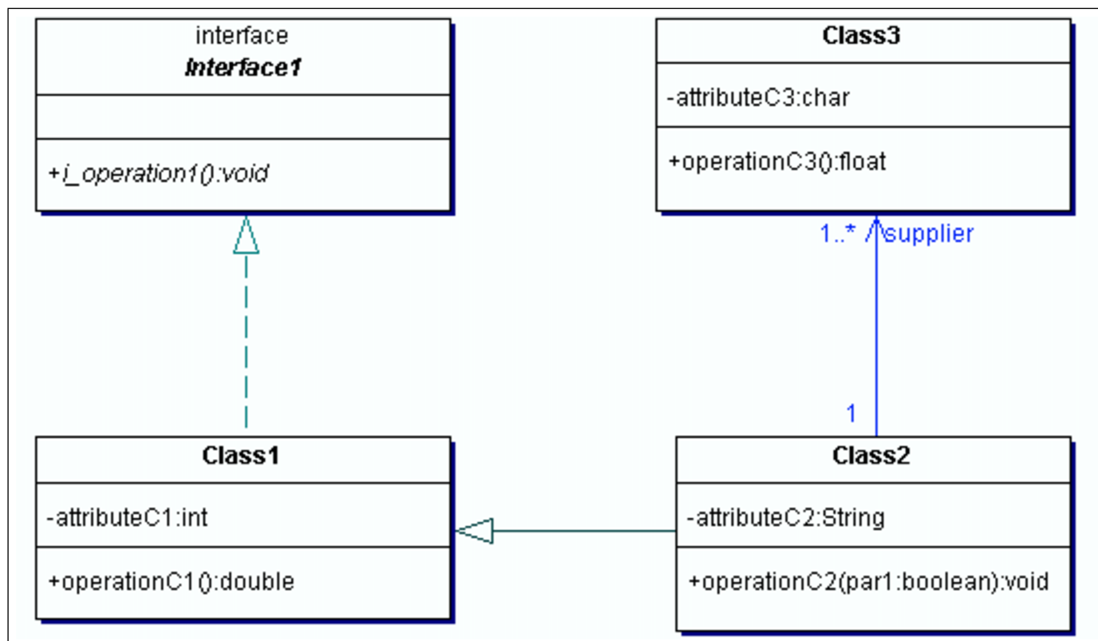


Abbildung 3.2: Beispielhaftes Klassendiagramm

Abbildung 3.2 zeigt ein Klassendiagramm, in welchem die Klasse *Class1* das Interface *interface1* implementiert und *Class2* von der Oberklasse *Class1* erbt. *Class2* ist weiterhin über eine gerichtete Assoziation mit der Klasse *Class3* verbunden.
Wir wollen nun schauen, auf welche Weise der zu diesem Klassendiagramm gehörige XMI-Output von der Funktion *parseClass* verarbeitet wird, um die Tabelle *ClassTable* der Hilfsstruktur zu füllen. Dazu schauen wir uns zunächst einmal den XML-Knoten an, welcher die Klasse *Class2* beschreibt. Dieser hat den folgenden Aufbau:

```
1    <UML:Class xmi.id = 'S.8'
2          name = 'Class2' visibility = 'public' isSpecification = '
              false'
3          isAbstract = 'false' isActive = 'false'>
4     <UML:Classifier.feature>
5        ...Informationen ueber Attribute und Methoden der Klasse...
6     </UML:Classifier.feature>
7     <UML:Namespace.ownedElement>
```

```
8         ...Informationen zur Oberklasse und zu Interfaces, welche die
              Klasse implementiert...
9      </UML:Namespace.ownedElement>
10    </UML:Class>
```

*parseClass* nutzt nun die Funktion *getAttrib*, um den Namen, sowie die Id der Klasse aus dem XML-Knoten auszulesen. Um aus den Kindknoten Informationen über die Attribute und Methoden, eine eventuelle Oberklasse, zu implementierende Interfaces und Assoziationen der Klasse zu gewinnen, wird die Funktion *parseElems* benötigt, welche die Elementliste der Klasse erstellt.

```
1  parseClass :: XmlNode -> Aux -> Aux
2  parseClass (Node _ attr chlist) (i,r,a,c) =
3                               (i,(xid,clname):r,a,cl:c)
4    where xid = getAttrib attr "xmi.id"
5          clname = getAttrib attr "name"
6          cl = (xid,clname,(parseElems chlist))
7
8  getAttrib :: [(String,String)] -> String -> String
9  getAttrib [] _ = ""
10 getAttrib ((f,v):cs) field | (f == field) = v
11                            | otherwise = getAttrib cs field
12
13 parseElems :: [XmlNode] -> [Element]
14 parseElems [] = []
15 parseElems ((Node na _ cl):nodes)
16     | (na=="UML:Classifier.feature") =
17         (parseOpAt cl)++(parseElems nodes)
18     | (na=="UML:Namespace.ownedElement") =
19         (parseAnc cl)++(parseElems nodes)
20
21 parseOpAt :: [XmlNode] -> [Element]
22 parseOpAt [] = []
23 parseOpAt (n@(Node na attr _):nodes)
24         | (na=="UML:Attribute") =
25       ((Attribute (getAttrib attr "name")
26                   (getAttrib adt "xmi.idref")):parseOpAt nodes)
27         | (na=="UML:Operation") =
28       ((Method  (opname)
29                 (getAttrib attr "stereotype")
30                 (parlistOmega) ret):parseOpAt nodes)
31   where (Node _ adt _) =
32                 path n ["UML:StructuralFeature.type",
33                         "UML:Classifier",
34                         "UML:Namespace.ownedElement",
35                         "UML:DataType"]
```

```
36          parlist = parseParams cl
37          parseParams [] = []
38          parseParams ((Node _ attr _):nodes) =
39            (((getAttrib attr "name"),
40              (getAttrib attr "type")):(parseParams nodes))
41          (Node _ _ cl) =
42                  (path n ["UML:BehavioralFeature.parameter"])
43          (parlistOmega,ret) =
44                  attrKill [] parlist (opname++".Return")
45          opname = getAttrib attr "name"
46          attrKill p [] _ = (p,"")
47          attrKill p ((f,v):cs) field
48            | (f == field) = (p++cs,v)
49            | otherwise = attrKill ((f,v):p) cs field
50
51  parseAnc :: [XmlNode] -> [Element]
52  parseAnc [] = []
53  parseAnc (n:nodes) =
54    ((Ancestor (getAttrib attr "xmi.idref")):(parseAnc nodes))
55    where (Node _ attr _) = path n ["UML:Generalization.parent",
56                                   "UML:GeneralizableElement"]
```

Die Funktion *parseOpAt*, welche von der Funktion *parseElems* benutzt wird, kümmert sich, wie der Name schon andeutet, um die Attribute und Methoden bzw. Operationen der Klasse. Weiterhin werden hier Verweise auf eventuell vorhandene Oberklassen erstellt, doch dazu später mehr. Attribute und Operationen befinden sich als Kindknoten im „UML:Classifier.feature“-Knoten. Ein Attribut wird erzeugt, sobald auf einen „UML:Attribute“-Knoten gestoßen wird. Dabei kann der Name des Attributes direkt aus der Attributliste des Attribut-Knotens extrahiert werden. Um an den Typ des Attributes heranzukommen, muss ein kleiner Umweg über die Id des Attribut-Typs genommen werden. Diese ist im Knoten „UML:DataType“ zu finden, welcher sich in den Kindknoten des „UML:Attribute“-Knotens befindet. Ein Attribut besteht in der Hilfsstrukur des UML-Parsers aus dem Namen und der Id, welche mit dem Typ des Attributes assoziiert ist. In der Zielstruktur wird diese Id über die *ResolveTable* aufgelößt.

```
1       <UML:Attribute xmi.id = 'S.15'
2             name = 'attributeC2' visibility = 'private'
                 isSpecification = 'false'
3             changeability = 'changeable' ownerScope = 'instance' type
                 = 'G.8'>
4         ...
5        <UML:StructuralFeature.type>
6         <UML:Classifier>
7          <UML:Namespace.ownedElement>
8           <UML:DataType xmi.idref = 'G.8'/>
9          </UML:Namespace.ownedElement>
```

```
10            </UML:Classifier>
11          </UML:StructuralFeature.type>
12        </UML:Attribute>
```

Eine Methode wird erzeugt, sobald auf einen „UML:Operation“-Knoten gestoßen wird. Der Name, sowie der Stereotyp der Methode können direkt aus der Attributliste des Operation-Knotens extrahiert werden. Die Parameter einer Methode sind im „UML:BehavioralFeature.parameter“-Knoten zu finden, welcher sich als Kindknoten im Operation-Knoten befindet. Die Unterfunktion *parseParams* extrahiert aus jedem Parameter-Knoten den Paramternamen, sowie die Parametertyp-Id und erstellt eine Liste von Parametern. Zu beachten ist hierbei, dass sich der Rückgabewert der Methode ebenfalls unter den Parameter-Knoten befindet. Dieser Parameterknoten hat immer den Namen "*Klassenname*.Return". Mittels der Unterfunktion *attrKill* wird nun der Parameter mit der Bezeichnung "*Klassenname*.Returnäus der durch *parseParams* generierten Liste herausgenommen und dessen Typ-Id als Rückgabetyp-Id der Methode zurückgegeben.

```
1         <UML:Operation xmi.id = 'S.14'
2               name = 'operationC2' visibility = 'public'
3               isSpecification = 'false'
4               isAbstract = 'false' ownerScope = 'instance'>
5           <UML:BehavioralFeature.parameter>
6             <UML:Parameter xmi.id = 'XX.8' name = 'par1' isSpecification =
                   'false' kind = 'inout' type = 'G.6'>
7               <UML:Parameter.type>
8                 <UML:Classifier xmi.idref = 'G.6'/>
9               </UML:Parameter.type>
10            </UML:Parameter>
11            <UML:Parameter xmi.id = 'XX.9' name = 'operationC2.Return'
                  isSpecification = 'false' kind = 'return' type = 'G.7'>
12              <UML:Parameter.type>
13                <UML:Classifier xmi.idref = 'G.7'/>
14              </UML:Parameter.type>
15            </UML:Parameter>
16          </UML:BehavioralFeature.parameter>
17        </UML:Operation>
```

Nachdem nun die Attribute und Methoden einer Klasse geparst sind, schauen wir auf die Funktion *parseElems* und stellen fest, dass diese neben der Funktion *parseOpAt* eine Funktion mit dem Namen *parseAnc* aufruft. Diese Funktion dient dazu, Ancestor-Elemente in die Elementeliste einer Klasse hinzuzufügen. Dabei ist ein Ancestor ein Verweis auf eine Klasse oder ein Interface, von welcher die betrachtete Klasse erbt bzw. welches die betrachtete Klasse implementiert. Besitzt eine Klasse eine Oberklasse oder implementiert Sie ein Interface, so existiert im „UML:Class“-Knoten ein Kindknoten Namens „UML:Namespace.ownedElement“. In dessen Unterknoten ist die Id zu finden, welche auf die entsprechende Oberklasse bzw. das entsprechende Interface verweist.

```
1       <UML:Namespace.ownedElement>
```

```
2       <UML:Generalization xmi.id = 'G.10'
3             name = '' visibility = 'public' isSpecification = 'false'
4              discriminator = ''>
5        <UML:Generalization.child>
6         <UML:GeneralizableElement xmi.idref = 'S.8'/>
7        </UML:Generalization.child>
8        <UML:Generalization.parent>
9         <UML:GeneralizableElement xmi.idref = 'S.7'/>
10       </UML:Generalization.parent>
11      </UML:Generalization>
12     </UML:Namespace.ownedElement>
```

Damit ist die Funktionsweise der Funktion *parseClass* beschrieben. Der Output des Parsers für die Klasse *Class2* unseres Beispiel hat die folgende Gestalt:

```
1 ("S.8","Class2",[Method "operationC2" "" [("par1","G.6")] "G.7",
2                  Attribute "attributeC2" "G.8",
3                  Attribute "lnkClass3" "G.9",
4                  Ancestor "S.7"])
```

Es fällt auf, dass in der Elementliste der Klasse ein Attribut mit dem Namen „lnkClass3“ vorhanden ist, obwohl dieses Attribut nicht als Attribut der Klasse deklariert worden ist. Der Grund dafür ist, dass die Klasse *Class2* eine Assoziation zur Klasse *Class3* besitzt. Borlands Together fügt in diesem Fall ein solches Attribut ein. Beim Generieren der Zielstruktur wird dieses Attribut jedoch nicht mehr beachtet.
Die Funktion *parseInterface* verfährt analog zu der erläuterten Funktion *parseClass*, und liefert für das Interface *interface1* folgenden Eintrag für die *InterfaceTable* der Hilfsstruktur:

```
1 ("S.9","Interface1",[Method "i_operation1" "" [] "G.7"])
```

Eine weitere Funktion die von der Schlüsselfunktion *parseChilds* aufgerufen wird ist die Funktion *parseNameIdRelation*. Diese Funktion hat die Aufgabe die *ResolveTable* der Hilfsstruktur zu erweitern. Durch die beiden Aufrufe der Funktion werden (Id, Name)-Tupel der folgenden beiden Arten einegfügt:

- Zuordnung einer Stereotyp-Id einer Methode zum Stereotyp-Namen
- Zuordnung von Datentyp-Ids zu Datentyp-Namen (beispielsweise muss die Typ-Id eines Attributes einer Klasse oder die Typ-Id eines Parameters einer Methode bei der Generierung der Zielstruktur aufgelöst werden)

Die letzte Funktion, welche zur Erstellung der Hilfsstruktur von *parseChilds* benötigt wird, ist die Funktion *parseAssociation*.

```
1 parseAssociation :: XmlNode -> Aux -> Aux
2 parseAssociation (Node _ _ (end1:end2:_)) (i,r,a,c) =
3   (i, r, assoc++a, c)
4   where (n1,r1,cid1,lr1,ur1) = getAssoc end1
```

```
5          (n2,r2,cid2,lr2,ur2) = getAssoc end2
6          assoc
7            | (n1 && not n2) = [(r1,cid2,cid1,
8                                  (lr2,ur2,lr1,ur1))]
9            | (not n1 && n2) = [(r2,cid1,cid2,
10                                 (lr1,ur1,lr2,ur2))]
11           | otherwise = [(r1,cid2,cid1,(lr2,ur2,lr1,ur1)),
12                            (r2,cid1,cid2,(lr1,ur1,lr2,ur2))]
13
14 getAssoc :: XmlNode -> (Bool, String, String, String, String)
15 getAssoc n@(Node _ attr _) = ( isNavig, r, clId, lr, ur )
16   where isNavig = (getAttrib attr "isNavigable")=="true"
17         r = getAttrib attr "name"
18         Node _ attr' _ =
19            path n ["UML:AssociationEnd.participant",
20                    "UML:Classifier"]
21         clId = getAttrib attr' "xmi.idref"
22         n'@(Node _ _ cl) =
23            path n [ "UML:AssociationEnd.multiplicity",
24                     "UML:Multiplicity" ]
25         (lr,ur) | cl==[] = ( "1", "1" )
26                 | otherwise = (getAttrib attr'' "lower",
27                                getAttrib attr'' "upper")
28         Node _ attr'' _ =
29            path n' [ "UML:Multiplicity.range",
30                      "UML:MultiplicityRange" ]
```

*parseAssociation* füllt die *AssocTable* der Hilfsstruktur. Diese ist folgendermaßen aufgebaut:

```
1 type AssocTable = [(String,String,String,Range)]
2 --(Rollenname, ClientId, SupplierId, Kardinalit"aten)
3
4 type Range = (String,String,String,String)
5 --(Client-Range from,
6 -- Client-Range to,
7 -- Supplier-Range from,
8 -- Supplier-Range to)
```

Die XMI-Darstellung eines UML-Klassendiagramms enthält für jede im Klassendiagramm enthaltene Assoziation einen „UML:Association“-Knoten, welcher immer zwei Kindknoten (für jedes Ende der Assoziation ein Knoten) mit der Bezeichnung „UML:AssociationEnd“ beherbergt. Für jeden dieser „UML:AssociationEnd“-Knoten ruft die Funktion *parseAssociation* die Funktion *getAssoc* auf. *getAssoc* liefert dann ein Tupel der Form (isNavigable, Name, ClientId, Multiplicity-from, Multiplicity-to). Der boolesche Wert *isNavigable* hat nur dann eine Bedeutung, wenn die betrachtete Assoziation gerichtet ist. Falls es sich um eine ungerichtete Assoziation handelt, so ist der Wert von

*isNavigable* für jedes Ende der Assoziation false. Handelt es sich dagegen, wie es in unserem Beispiel der Fall ist, um eine gerichtete Assoziation, so ist der Wert von *isNavigable* am Pfeilende der Assoziation true und am Pfeilanfang der Assoziation false. In unserem Beispiel heißt das, dass nur die Klasse *Class2* (am Pfeilanfang der Assoziation) auf Attribute und Methoden der Klasse *Class3* (am Pfeilende der Assoziation) zugreifen kann.
*getAssoc* extrahiert also aus der Attributliste eines „UML:AssociationEnd“-Knotens den Wert für *isNavigable* und den Rollennamen der Assoziation. Wurde kein Rollenname für ein Assoziationsende vergeben, so ist der von
*getAssoc* gelieferte String für den Rollennamen leer. Bei der Generierung der Zielstruktur wird an dieser Stelle einfach der Name der Klasse, welche sich an dem betrachteten Assoziationsende befindet, als Rollenname für das Assoziationsende verwendet.
Die ClientId, d.h. die Id der Klasse, welche an dem betrachteten Assoziationsende liegt, ebenso wie die Multiplizitäten, bekommt *getAssoc* aus den Kindknoten des „UML:Association“-Knotens. *parseAssociation* erstellt nun ausgehend von den Ausgaben der Funktion *getAssoc* einen Eintrag für die Tabelle *AssocTable*. *getAssoc* liefert in unserem Beispiel die folgenden Tupel:

- Für das Assoziationsende am Pfeilanfang (an Klasse *Class2*):
  - ('false', ", 'S.8', '1', '1') = getAssoc end1
- Für das Assoziationsende am Pfeilende (an Klasse *Class3*):
  - ('true', 'supplier', 'S.10', '1', '-1') = getAssoc end2

*parseAssociation* generiert nun aus den Tupeln den folgenden Eintrag für die *AssocTable* der Hilfsstruktur:

```
1 ("supplier", "S.8", "S.10", ("1","1","1","−1"))
```

Die Id SS.8ßteht dabei für die Klasse *Class2*, welche den Client darstellt und die Id SS.10ßteht für die Klasse *Class3*, welche als Supplier fungiert.

```
1     <UML:Association ...>
2      <UML:Association.connection>
3
4       <UML:AssociationEnd xmi.id = 'G.1' visibility = 'public'
            isSpecification = 'false'
5              isNavigable = 'false' aggregation = 'none'
6              targetScope = 'instance' changeability = 'changeable'>
7        <UML:AssociationEnd.qualifier>
8         <UML:Attribute xmi.idref = 'S.11'/>
9        </UML:AssociationEnd.qualifier>
10       <UML:AssociationEnd.multiplicity>
11        <UML:Multiplicity>
12         <UML:Multiplicity.range>
13          <UML:MultiplicityRange lower = '1' upper = '1'/>
```

```
14          </UML:Multiplicity.range>
15         </UML:Multiplicity>
16        </UML:AssociationEnd.multiplicity>
17        <UML:AssociationEnd.participant>
18         <UML:Classifier xmi.idref = 'S.8'/>
19        </UML:AssociationEnd.participant>
20       </UML:AssociationEnd>
21
22       <UML:AssociationEnd xmi.id = 'G.2'
23             name = 'supplier' visibility = 'public' isSpecification =
                 'false'
24             isNavigable = 'true' aggregation = 'none'
25             targetScope = 'instance' changeability = 'changeable'>
26        <UML:AssociationEnd.multiplicity>
27         <UML:Multiplicity>
28          <UML:Multiplicity.range>
29           <UML:MultiplicityRange lower = '1' upper = '-1'/>
30          </UML:Multiplicity.range>
31         </UML:Multiplicity>
32        </UML:AssociationEnd.multiplicity>
33        <UML:AssociationEnd.participant>
34         <UML:Classifier xmi.idref = 'S.10'/>
35        </UML:AssociationEnd.participant>
36       </UML:AssociationEnd>
37
38      </UML:Association.connection>
39     </UML:Association>
```

Damit sind alle Funktionen, welche an der Erstellung der Hilfsstruktur des UML-Parsers beteiligt sind, erläutert. Ein reeler Durchlauf des Parsers mit der zum Beispiel gehörigen XMI-Datei liefert das folgende Ergebnis für die Hilfsstruktur:

```
1 --InterfaceTable
2 [("S.9","Interface1",[Method "i_operation1" "" [] "G.7"])]
3
4 --ResolveTable
5 [("G.12","char"),        ("G.11","float"),
6  ("G.9","Class3"),       ("G.8","java.lang.String"),
7  ("G.7","void"),         ("G.6","boolean"),
8  ("G.4","int"),          ("G.3","double"),
9  ("S.9","Interface1"),  ("S.10","Class3"),
10 ("S.8","Class2"),       ("S.7","Class1")]
11
12 --AssociationTable
13 [("supplier","S.8","S.10",("1","1","1","-1"))]
14
```

```
15 --ClassNodes
16 [("S.10","Class3",[Method "operationC3" "" [] "G.11",
17                    Attribute "attributeC3" "G.12"]),
18  ("S.8","Class2", [Method "operationC2" ""
19                           [("par1","G.6")] "G.7",
20                    Attribute "attributeC2" "G.8",
21                    Attribute "lnkClass3" "G.9",
22                    Ancestor "S.7"]),
23  ("S.7","Class1", [Method "operationC1" "" [] "G.3",
24                    Attribute "attributeC1" "G.4",
25                    Ancestor "S.9"])]
```

Nachdem die gefüllte Hilfsstruktur nun zur Verfügung steht, kann sie in die Zielstruktur überführt werden. Die Zielstruktur steLlt eine Liste von Klassen der Datenstruktur *Class* dar. Eine Klasse stellt die für den UML-Compiler notwendigen Daten aufbereitet zur Verfügung und ist folgendermaßen definiert:

```
1 -- Definition der Datenstruktur Class
2 data Class = Class String [Element] deriving(Eq,Show,Read)
```

Das Füllen der Zielstruktur:

- *getClasses*
- *createClasses*
- *createClass*

Die Funktion *getClasses* sorgt dabei lediglich für einen richtig parametrisierten Aufruf der Funktion *createClasses*, welche für jede Klasse in der *ClassTable* der Hilfsstruktur die Funktion *createClass* aufruft. Wesentlich ist somit die Funktion *createClass*, welche folgende Aufgaben, bezogen auf die Umsetzung einer Klasse der *ClassTable* in eine Klasse der Datenstruktur *Class*, hat:

1. Übernahme aller Attribute der Hilfsstrukturklasse in die Zielklasse, wobei die Typ-Ids der Attribute mit Zuhilfenahme der *ResolveTable* zu Typ-Namen aufgelöst werden. Weiterhin wird hier das oben erwähnte Attribut "lnk*Klassenname*"(s.o.) übersprungen und nicht in die Elementeliste der Zielklasse übernommen.

2. Übernahme aller Methoden der Hilfsstrukturklasse in die Zielklasse, wobei die Stereotyp-Ids der Methoden zu Stereotyp-Namen und die Typ-Ids der Parameter der Methoden zu Typ-Namen aufgelöst werden. Dies geschieht ebenfalls mit Hilfe der *ResolveTable* der Hilfsstruktur.

3. Für alle Ancestor-Elemente in der Elementeliste der Hilfsstrukturklasse wird geprüft, ob die Ancestor-Id einer Id eines Interfaces in der *InterfaceTable* der Hilfsstruktur entspricht. Ist dies der Fall, so werden alle Elemente des zu implementierenden Interfaces in die Elementliste der Zielklasse übernommen. Erbt die betrachtete

Klasse von einer Oberklasse, d.h. existiert ein Ancestor-Element, dessen Id nicht der Id eines Interfaces in der *InterfaceTable* entspricht, so wird das Ancestor-Element in die Elementliste der Zielklasse übernommen, wobei die Ancestor-Id über die *ResolveTable* zum tatsächlichen Oberklassennamen aufgelöst wird.

4. Schließlich werden die Assoziationen der Hilfsstrukturklasse bearbeitet. Diese befinden sich in der *AssocTable* der Hilfsstruktur. Ein Assoziation-Element der *AssocTable* ist, wie bereits weiter oben erläutert, ein Tupel der Form (Rollenname, ClientId, SupplierId, Multiplizitäten), wobei der Rollenname ggf. leer sein kann. Umgesetzt wird dieses Tupel von der Funktion *createClass* in ein Tupel der Form (Assoziationsname, Assoziationstyp, Multiplizitäten). Falls der Rollenname nicht leer ist, entspricht der Assoziationsname dem Rollennamen, anderenfalls entspricht er dem Namen, welcher sich aus der Auflösung der SupplierId ergibt. Der Assoziationstyp entspricht der über die *ResolveTable* aufgelösten ClientId. Die Multiplizitäten werden ohne Änderungen übernommen.

```
1  createClass :: (String,String,[Element]) -> Aux -> Class
2  createClass (cid,cname,elems) sta@(i,r,a,c) = Class cname el
3    where el = (createElems elems)
4          createElems [] = createAssocs a
5          createElems ((Attribute an at):erest)
6           | (includesString an ("lnk"++(getAttrib r at)))
7             = (createElems erest)
8           | otherwise
9             = ((Attribute an (getAttrib r at))
10                                :(createElems erest))
11         createElems ((Method n st pl b):erest)
12            = ((Method n (getAttrib r st) (createParams n pl)
13                        (getAttrib r b)):(createElems erest))
14         createElems ((Ancestor ancn):ancrest)
15          | ((findAnc (i) ancn)==("","",[]))
16            = ((Ancestor (getAttrib r ancn))
17                         :(createElems ancrest))
18          | otherwise = (getElemList (createClass
19              (findAnc (i) ancn) sta))++(createElems ancrest)
20         getElemList (Class _ elemList) = elemList
21         findAnc [] ancn = ("","",[])
22         findAnc ((ancId,ancName,ancElements):findAncRest)
23                 ancnAnc
24          | (ancId == ancnAnc) = (ancId,ancName,ancElements)
25          | otherwise = findAnc findAncRest ancnAnc
26         createParams _ [] = []
27         createParams mn ((p,pt):prest)
28            = (p,(getAttrib r pt)):(createParams mn prest)
29         createAssocs [] = []
```

```
30          createAssocs ((assRn, assIdC, assIdS, assRange)
31                        :assRest)
32            | ((assIdC == cid) && (assRn == ""))
33              = ((Association (getAttrib r assIdS)
34                              (getAttrib r assIdS)
35                              assRange)
36                              :(createAssocs assRest))
37            | ((assIdC == cid) && (not (assRn == "")))
38              = ((Association assRn
39                              (getAttrib r assIdS)
40                              assRange)
41                              :(createAssocs assRest))
42            | otherwise = createAssocs assRest
```

Der Durchlauf der zum beispielhaften Klassendiagramm gehörigen XMI-Datei durch den UML-Parser liefert letztlich das folgende Ergebnis:

```
1  [Class "Class1" [Method "operationC1" "" [] "double",
2                   Attribute "attributeC1" "int",
3                   Method "i_operation1" "" [] "void"],
4   Class "Class2" [Method "operationC2" ""
5                          [("par1","boolean")] "void",
6                   Attribute "attributeC2" "java.lang.String",
7                   Ancestor "Class1",
8                   Association "supplier" "Class3"
9                               ("1","1","1","-1")],
10  Class "Class3" [Method "operationC3" "" [] "float",
11                  Attribute "attributeC3" "char"]]
```

Anzumerken ist hierbei, das der vorangegangen beschriebene UML-Parser das Ergebnis einer kompletten Neubearbeitung eines zu Anfang umgesetzten Parsers ist, in welchem die Trennung zwischen der Erzeugung der Hilfsstruktur und der darauf folgenden Generierung der Zielstruktur nicht vorgenommen worden war. Der Programmablauf war damit undurchsichtig geworden, so das beschlossen worden ist, den UML-Parser von Grund auf neu zu strukturieren. Die Arbeitsweise des hier beschriebenen UML-Parsers ist damit deutlich einfacher nachzuvollziehen. Ggf. notwendige Anpassungen an Weiterentwicklungen des XMI-Standards können somit leichter vollzogen werden.

## 3.3 UML-Compiler

Dieser Teil beschreibt den Aufbau und die Arbeitsweise des UML-Compilers. Zunächst werden die generellen Überlegungen zur Darstellung des UML-Teils von objektorientierten Programmen in der *Expander2* -Logik vorgestellt, danach der Compiler in seiner Funktionsweise erklärt.

### 3.3.1 Allgemeines Übersetzungsschema von UML

Um objektorientierte Programme in einer Logik darzustellen, die der *Expander2* verarbeiten kann, sind ein paar grundlegende Überlegungen notwendig. Zu Beginn der PG-Arbeit waren wir bemüht, den Aufbau von objektorientierten Programmen so genau wie möglich umzusetzen, z.B. Objekte als eine Variable in der *Expander2* -Logik darzustellen, auf deren Elemente nur mittels get- und set-Methoden, bzw. get- und set-Axiome zugegriffen werden kann. Objekte waren also *constructs*, die mittels eines *create*-Axioms erzeugt wurden. Das bedeutete, dass für ein Programmbeispiel sehr viele Axiome nötig waren. Um z.B. Methoden, die von einer Oberklasse geerbt wurden, auszuführen wurden Einbettungs-Axiome benötigt; diese Axiome überführten ein Objekt der Unterklasse in eines der Oberklasse und nach Anwendung der Methoden-Axiome wieder zurück in ein Objekt der Unterklasse.

Methoden einer Klasse wurden zunächst als Funktionen im *Expander2* dargestellt, die das Objekt und alle Parameter einzeln übergeben bekamen. Das Ergebnis der Methode wurde durch Auswerten eines großen Guards ermittelt. Dieser Guard musste alle Parameter, verwendete Objektattribute und die eigentliche Funktion auswerten. Erschwerend kam hinzu, dass die Objektattribute nur mittels get- und set-Axiomen zugegriffen wurde.

Im Verlauf der Beispielaxiomatisierung und des Testens der Umsetzung wurde jedoch schnell klar, dass eine derart nahe Transformation an der Darstellung von objektorientierten Programmen nicht zweckmäßig ist. Da die Anwendung von Axiomen mit großen Guards im *Expander2* teilweise einige Zeit dauert, kam es bei der Fülle und Größe der Axiome schnell zu Performance-Problemen. Ein weiteres Problem war, dass Guards in Axiomen immer vollständig auswertbar sein müssen, was erstens nicht immer nötig, teilweise später in der Beweisführung auch nicht möglich gewesen wäre.

Deswegen sind wir zu einer anderen Darstellung von objektorientierten Programmen gekommen, die in vielen Punkten eine Verbesserung gegenüber dieser modellnahen Darstellung aufweist. Objekte einer Klasse sind nun Listen, bestehend aus einer Klassen-Id, den Attributen einer Klasse, sowie eventueller Oberklassenattribute. Um auf die Elemente eines Objekts zuzugreifen können nun einfache Listen-Operationen verwendet werden, `getX(object)` und *updX(object,neuerWert)*. `X` ist hierbei eine Zahl aus $\mathcal{N}_0$, die als Index die betreffende Position in der Objektliste *object* referenziert. Hierzu einige einfache Beispiele:

- `get1([0,'attr1','attr2']) = 'attr1'`
- `upd2([0,'attr1','attr2'],'attr3') = [0,'attr1','attr3']`

### 3.3.2 Aufbau und Funktion des Compilers

Der UML-Compiler stellt aus der geparsten Zielstruktur des UML-Parsers nun den ersten Teil der *Expander2* -Logik her. Hierbei handelt es sich vorrangig um die Aufnahme aller Methodennamen in die Signatur und die Definition und Implementierung von createClass-Axiomen, isClass-Prädikaten und Axiomen zum Aufruf von Oberklassen-Methoden.

Für jede Klasse wird ein createClass-Axiom zur Herstellung einer Objektinstanz der Gestalt

```
(length(params) = laenge ==> createCLASS(params) = [ID,params])
```

erzeugt. Darin ist *CLASS* der Name der Klasse, *ID* eine systemweit eindeutige Kennziffer und *params* die Liste aller Attribute der Klasse. Ebenfalls wird jeweils ein isClass-Axiom der Gestalt

```
(isClass3(self) <=== (get0(self)=2))
```

erzeugt.

Einbettungs-Axiome werden nur für die Methoden erzeugt, die eine Klasse von einer Oberklasse erbt. Die Einbettung geschieht dabei durch Ändern der Klassen-Id auf die ID der Oberklasse und im Ergebnis auf die ID der Unterklasse. Das erzeugte Prädikat hat die Gestalt

```
(operation(self,params,self',result) <=== isCLASS(self) & self' =
                    upd0(self",ClassID) &
      operation(upd0(self,UpperClassID),params,self",result))
```

Die Zielstruktur des UML-Compilers ist dann ein *Expander2* -Tupel von sieben Listen:

**specs**
: enthält alle zu importierenden Module

**constructs**
: ist die Liste der Konstruktoren

**defuncts**
: enthält alle Funktionssymbole

**preds**
: ist die Liste der Prädikatensymbole

**fovars**
: enthält die First-Order-Variablen

**hovars**
: ist die Liste der Higher-Order-Variablen

**axioms**
: enthält die Axiome

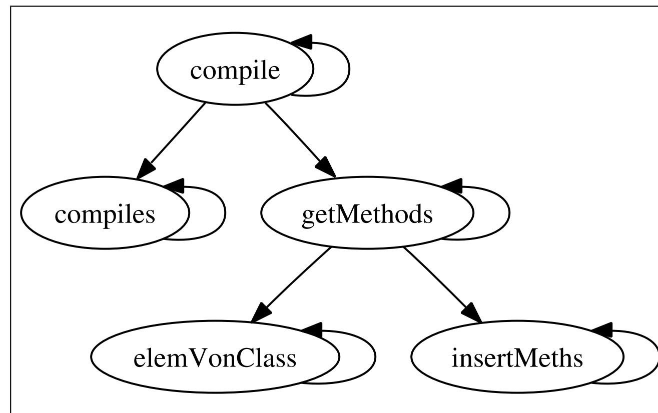


Abbildung 3.3: Schema: UML-Compiler

Der größte Teil der Übersetzung von der UML-Parser-Zielstruktur in die *Expander2* -Logik geschieht durch die Funktion *compile*. Dieser wird bei dem Aufruf eine *0* als erste KlassenID, das initiale *Expander2* -Tupel sowie zweimal die durch den Parser erstellte Klassenliste übergeben.

```
1 compile :: Int -> [Class] -> [Class] -> \expander -> \expander
2 compile _ _ [] e = e
3 compile cid cl ((Class cn elems):cs) (s,c,d,p,f,h,a) = (compile (cid
    +1) cl cs (compiles cid cl cn meths (s,c,(insert d ("create"++cn))
    ,(insert p ("is"++cn)),f,h,(insert (insert a isClassAx) create))))
4   where isClassAx = ("(is"++cn++"(self) <=== (get0(self)="++(show cid
      )++"))")
5         create = ("((length(params) = "++(show (length attrs))++") ==>
           create"++cn++"(params) = ("++(show cid)++":params))")
6         meths = (getMethods "unter" cl elems ([],"",[]))
7         attrs = (getInheritanceAttrList cl cn)
```

In der Compile-Funktion wird in jedem Durchlauf eine Klasse vom Beginn der Klassenliste bearbeitet und hierzu bereits die createCLASS-Funktion und das isCLASS-Prädikat erzeugt und in das *Expander2* -Tupel eingefügt. Für die create-Funktion werden alle Klassenattribute benötigt, welche durch die Funktion *getInheritanceAttrList* geholt werden.

```
1 getInheritanceAttrList :: UML -> String -> [Element]
2 getInheritanceAttrList uml name = concatMap (getAttrList uml) (
    getInheritancePath uml name)
3
4 getAttr :: UML -> String -> String -> (Int,String)
5 getAttr uml klasse attr = get e 1
6   where e = getInheritanceAttrList uml klasse
7         get ((a@(Attribute n t)):r) i | (n==attr) = (i,t)
8                                       | otherwise = get r (i+1)
```

```
9         get ((a@(Association n t _)):r) i | (n==attr) = (i,t)
10                                          | otherwise = get r (i+1)
11        get [] i = (-i, "" )
```

Diese beiden Funktionen, sind mit vielen anderen Funktionen, die sowohl vom UML- als auch vom OCL-Compiler verwendet werden in dem Modul CompTools zusammengefasst. So kann durch Übergeben einer Klassenliste und eines Klassennamens eine Liste aller Attribute erzeugt werden. Diese Liste enthält natürlich auch alle Assoziationen der Klasse.

Was nun noch fehlt ist das Aufnehmen der Methoden-Namen und etwaiger Einbettungs-Axiome in das *Expander2* -Tupel. Hierzu wird die Funktion *compiles* verwendet.

```
1 compiles :: Int -> [Class] -> String -> ([Element],String,[Element])
    -> \expander -> \expander
2 compiles _ _ _ ([],_,[]) e = e
3 compiles ucId cl cn (((Method mn _ _ _):ce),ancn,oberelems) (s,c,d,p,f
    ,h,a) = compiles ucId cl cn (ce,ancn,oberelems) (s,c,d,(insert p mn
    ),f,h,a)
4 compiles ucId cl cn ([],ancn,(Method mn _ params _):ce) (s,c,d,p,f,h,a
    ) = compiles ucId cl cn ([],ancn,ce) (s,c,d,p,f,h,(insert a ax))
5   where ocId = resolveId 0 cl ancn
6         resolveId cid ((Class cname celems):cls) anc | (cname==anc) =
             cid
7                                                      | otherwise =
                                                          resolveId (cid
                                                          +1) cls anc
8         ax = "("++mn++"(self,params,self',result)␣<===␣is"++cn++"(self
             )␣&␣self'␣=␣upd0(self'',"++show ucId++")␣&␣"++mn++"(upd0(
             self,"++show ocId++"),params,self'',result))"
```

Diese Funktion ist zweigeteilt. Sie bekommt von der compile-Funktion alle Methoden der Klasse in einem Tupel der Form *([Element],String,[Element])* übergeben, welche folgende Werte enthält (Klassenmethoden,'Name der vererbenden Klasse',Oberklassenmethoden). Dieses Tupel wird zuvor durch die Hilfsfunktion *getMethods* erzeugt. Der erste Teil von *compiles* fügt nun die Methodennamen der Klassenmethoden in die Prädikatenliste des *Expander2* -Tupels ein. Der zweite Teil erstellt zu jeder Oberklassenmethode ein Einbettungs-Axiom und fügt dieses ebenfalls in das *Expander2* -Tupel ein. Für das Beispiel, das bereits im UML-Parser-Teil gezeigt wurde, würde der UML-Compiler damit folgende Spezifikation liefern: Aus dem Ergebnis des Parsers:

```
1 [Class "Class1" [Method "operationC1" "" [] "double",
2                  Attribute "attributeC1" "int",
3                  Method "i_operation1" "" [] "void"],
4  Class "Class2" [Method "operationC2" ""
5                        [("par1","boolean")] "void",
6                  Attribute "attributeC2" "java.lang.String",
7                  Ancestor "Class1",
```

```
8                       Association "supplier" "Class3"
9                                   ("1","1","1","-1")],
10  Class "Class3" [Method "operationC3" "" [] "float",
11                      Attribute "attributeC3" "char"]]
```

entstehen folgende Elemente, die dann im *Expander2* -Tupel enthalten sind:

```
1  specs:
2    LIST LISTEVAL
3  defuncts:
4    createClass1 createClass2 createClass3
5  preds:
6    isClass1 isClass2 isClass3 i_operation1 operationC1 operationC2
        operationC3
7  fovars:
8    params result self self' self''
9  axioms:
10   ((length(params) = 1) ==> createClass3(params) = (2:params)) &
11
12  (isClass3(self) <=== (get0(self)=2)) &
13
14  (operationC1(self,params,self',result) <=== isClass2(self) & self' =
      upd0(self'',1) & operationC1(upd0(self,0),params,self'',result)) &
15
16  (i_operation1(self,params,self',result) <=== isClass2(self) & self' =
       upd0(self'',1) & i_operation1(upd0(self,0),params,self'',result))
       &
17
18  ((length(params) = 3) ==> createClass2(params) = (1:params)) &
19
20  (isClass2(self) <=== (get0(self)=1)) &
21
22  ((length(params) = 1) ==> createClass1(params) = (0:params)) &
23
24  (isClass1(self) <=== (get0(self)=0))
```

Nach Abarbeitung der gesammten Klassenliste in der compile-Funktion ist nun das *Expander2* -Tupel um alle relevanten Daten des UML-Klassendiagramms erweitert worden. Es ist deutlich zu sehen, dass nur ein geringer Teil der Daten aus der XML-Datei überhaupt ausgelesen und aus der Parser-Zielstruktur auch nur ein Teil der Daten seinen Weg in das *Expander2* -Tupel gefunden hat. Dies bedeutet allerdings nicht, dass unwichtige Daten im UML-Parser ausgelesen wurden, denn zusammen mit dem erzeugten *Expander2* -Tupel wird ebenfalls die Parser-Zielstruktur an den OCL-Compiler übergeben, der daraus weitere Daten verwendet.

# Kapitel 4

# OCL

## 4.1 Überblick

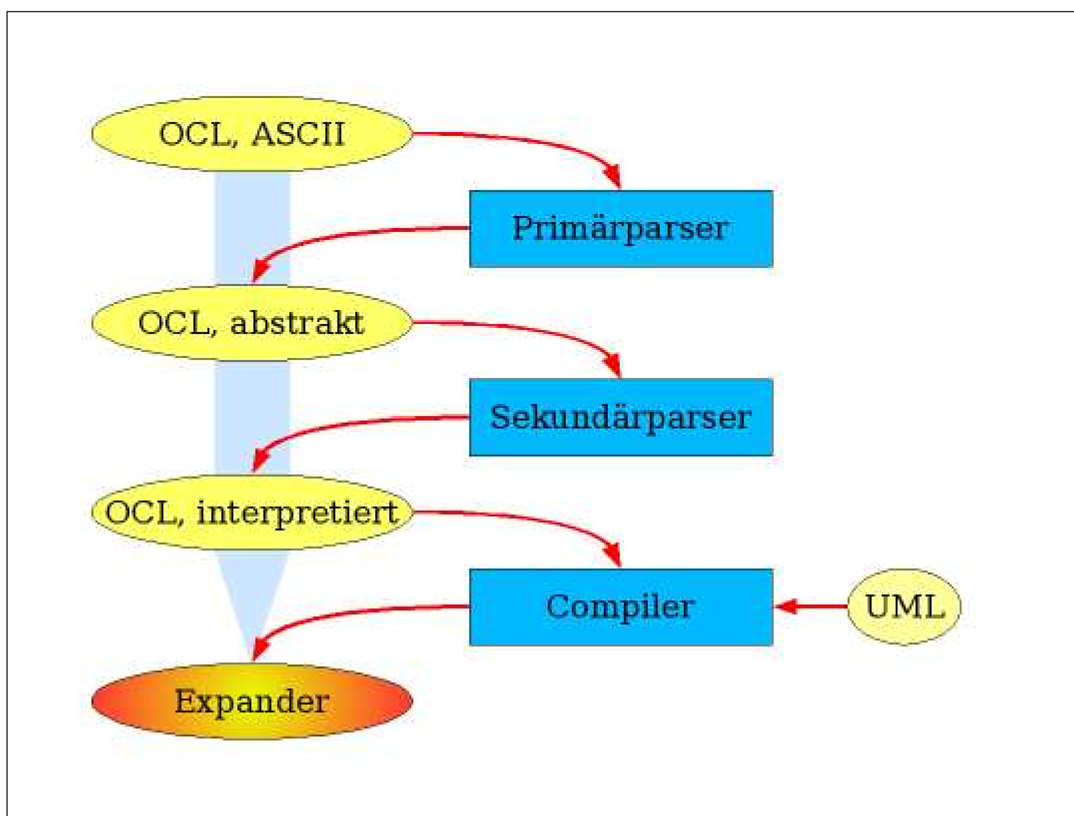


Abbildung 4.1: Aufbau des OCL-Compilers

Das Erzeugen der *Expander2* -Spezifikationen aus den OCL-Constraints erfolgt in mehreren Phasen, die in Abbildung 4.1 dargestellt sind. Zu Beginn werden die OCL-Constraints mittels des Primärparsers (siehe 4.4) eingelesen und in einen der Gramma-

tik entsprechenden Ableitungsbaum überführt.

Da diese Darstellung jedoch sehr komplex ist, viele redundante Informationen enthält und zum Transformieren einer Constraint nie der gesamte Ableitungsbaum erforderlich ist, wird der Ableitungsbaum mithilfe des Sekundärparsers (siehe 4.5) in eine Listenstruktur überführt. Gleichzeitig werden auch redundante Informationen herausgefiltert. Diese Überführung der Darstellung von einem linearen in einen zyklischen Aufbau ist nur möglich, da durch den Primärparser sicher gestellt wurde, dass die Syntax korrekt ist.

Nachdem nun die OCL-Spezifikation in einer kompakten und für den weiteren Verlauf "angenehmeren" Listendarstellung vorliegt, wird sie nun durch den Compiler (siehe 4.6) in eine *Expander2* - Spezifikation überführt. Der Compiler benutzt hierbei sowohl die bereits aus dem UML-Klassendiagramm erzeugte Spezifikation als auch die gesamten durch den UML-Parser (siehe 3.2) eingelesenen UML-Informationen. Diese werden zum Beispiel zum Auflösen von Rückgabetypen verwendet.

## 4.2 Grammatik

Die hier verwendeten OCL-Constraints beruhen auf einer leicht modifizierte Form der durch UML 1.5 [39] festgelegten Grammatik (siehe A). Die Modifikationen bestehen lediglich darin, dass der `|`-Operator entfernt wurde, indem für die verschiedenen Fälle mehrere aber dafür eindeutige Produktionen eingeführt wurden.

Die OCL-Grammatik ist jedoch unpräzise, d.h. sie beschreibt eine deutlich größere Menge von Sätzen als in der Sprache OCL gültig ist. Beispielsweise werden die Schlüsselwörter `self` und `result` oder auch die OCL-Collectionfunktionen als solche nicht explizit in der Grammatik aufgeführt. Ebenso wird die in OCL-Ausdrücken notwendige Typverträglichkeit nicht von der Grammatik erfasst. Das führt dazu, dass ein großer Teil der OCL-Syntax nicht beim Parsen geprüft werden kann.

## 4.3 interne Funktionen

Da OCL zahlreiche Funktionen zur Verfügung stellt, müssen diese auch im weiteren Verlauf in der Logik bzw. *Expander2* - Syntax spezifiziert werden, um den vollen Umfang von OCL nutzen zu können. Aus Effizienzgründen wird der Compiler jedoch die OCL-Constraints repräsentierenden Axiome so erzeugen, dass die im folgenden vorgestellten `defuncts` bereits im Vorfeld angewendet werden.

### 4.3.1 Operationen für alle Kollektionstypen

Im folgenden wird nun auf die für alle Kollektionstypen definierten Operationen eingegangen. Die konkreten Operationen für einzelne Kollektionstypen unterscheiden sich lediglich im Rückgabetyp und im Typ des übergebenen Parameters.

- `size()`: gibt die Zahl der Elemente zurück
- `count(object)`: Häufigkeit des Auftretens eines Objektes in der Kollektion
- `includes(object)`: wahr, wenn das Objekt ein Element der Kollektion ist
- `includesAll(collection)`: wahr, wenn alle Elemente der übergebenen Kollektion in der aktuellen Kollektion enthalten sind
- `excludes(object)`: wahr, wenn das Objekt kein Element der Kolletion ist
- `excludesAll(object)`: wahr, wenn keines der Objekte der übergebenen Kollektion Element der aktuellen Kollektion ist
- `isEmpty()`: wahr, wenn die Kollektion kein Element besitzt
- `notEmpty()`: wahr, wenn die Kollektion mindestens ein Element besitzt
- `iterate(expression)`: der übergebene Ausdruck wird für jedes Element der Kollektion ausgewertet; der Rückgabetyp ist vom Ausdruck abhängig
- `sum()`: die Addition aller Elemente in der Kollektion
- `exists(expression)`: wahr, wenn der Ausdruck für mindestens ein Element der Kollektion wahr ist
- `forAll(expression)`: wahr, wenn der Ausdruck für alle Elemente der Kollektion wahr ist
- `union(collection)`: vereinigt die übergebene Kollektion mit der aktuellen Kollektion
- `interesection(collection)`: bildet die Schnittmenge der übergebenen und der aktuellen Kollektion
- `including(object)`: fügt der Kollektion das übergebene Objekt hinzu
- `excluding(object)`: entfernt das übergebene Objekt aus der Kollektion
- `collect(expression)`: berechnet einen Wert für jedes Element der Kollektion und sammelt diese Werte in einer neuen Kollektion
- `select(expression)`: bildet die Teilmenge der Kollektion, deren Elemente die übergebene Bedingung erfüllen
- `reject(expression)`: bildet die Teilmenge der Kollektion, deren Elemente die übergebene Bedingung nicht erfüllen

**Iterierende Operationen**

Die Operationen `select`, `reject`, `collect`, `forAll`, `exists`, `iterate` iterieren über die einzelnen Elemente der Kollektion. Es gibt drei syntaktische Varianten für iterierende Operationen, die alle berücksichtigt werden müssen:

```
collection->operation( <expression> )
collection->operation( element | <expression> )
collection->operation( element:Type | <expression> )
```

In der ersten Variante wird die Iterator-Variable implizit deklariert. Der Ausdruck `<expression>` befindet sich im Scope der Klasse der Elemente, d.h. der Zugriff auf die Eigenschaften kann direkt erfolgen, z.B.:

```
context Geschoss inv:
  raeume -> forAll( hoehe=self.hoehe )
```

In der zweiten Variante wird eine explizite Referenz auf das Element definiert; das obige Beispiel sieht mit expliziter Referenz wie folgt aus:

```
context Geschoss inv:
  raeume -> forAll( r | r.hoehe=self.hoehe )
```

Bei der dritten Variante wird neben der expliziten Referenz auch der Typ angegeben; so wird sichergestellt, dass das Element der Kollektion auch vom erwarteten Typ ist.

```
context Geschoss inv:
  raeume -> forAll( r:Raum | r.hoehe=self.hoehe )
```

### 4.3.2 Zusätzliche Operationen für Sequenzen

Da Sequenzen geordnete Kollektionen sind, sind für sie zusätzliche Operationen für den expliziten Zugriff auf eine bestimmte Position definiert.

- `first()`: Zugriff auf das erste Element der Sequenz
- `last()`: Zugriff auf das letzte Element der Sequenz
- `at(pos)`: Zugriff auf das Element an Position *pos*
- `append(object)`: fügt ein Element an erster Stelle ein
- `prepend(object)`: fügt ein Element an letzter Stelle ein

### 4.3.3 *Expander2* - Umsetzung

In Tabelle 4.1 nun das Schema, nach dem der Aufruf von Collection-Funktionen in *Expander2* - Spezifikationen überführt wird. An dieser Stelle wird das Schema nicht detailliert beschrieben, sondern nur jeweils ein abstraktes Beispiel gegeben.

| Funktion | OCL | *Expander2* |
|---|---|---|
| *size* | `col->size()` | `length(col)` |
| *count* | `col->count(obj)` | `count(col,objg)` |
| *includes* | `col->includes(obj)` | `obj 'in' col` |
| *includesAll* | `col->includesAll(col')` | `col' <= col` |
| *excludes* | `col->excludes(obj)` | `obj 'not_in' col` |
| *excludesAll* | `col->excludesAll(col')` | `col' 'disjoint' col` |
| *isEmpty* | `col->isEmpty()` | `null(col)` |
| *notEmpty* | `col->notEmpty()` | `length(col)>0)` |
| *iterate* | `col->iterate(F,S)` | `foldl(S)(F)(col)` |
| *sum* | `col->sum()` | `sum(col)` |
| *exists* | `col->exists(P)` | `any(P)(col)` |
| *forAll* | `col->forAll(P)` | `all(P)(coL)` |
| *union* | `col->union(col')` | `col 'union' col'` |

Tabelle 4.1: Exemplarisches Transformationsschema für Collection-Funktionen

| Funktion | OCL | *Expander2* |
|---|---|---|
| *intersection* | `col->intersection(col')` | `col 'meet' col'` |
| *including* | `col->including(obj)` | `col 'meet' [obj]` |
| *excluding* | `col->excluding(obj)` | `col - [obj]` |
| *collect* | `col->collect(F)` | `map(F)(col)` |
| *select* | `col->select(P)` | `filter(P)(col)` |
| *first* | `col->first()` | `head(col)` |
| *last* | `col->last()` | `last(col)` |
| *at* | `col->at(i)` | `geti(col)` |
| *append* | `col->append(obj)` | `obj:col` |
| *prepend* | `col->prepend(obj)` | `col++[obj]` |

## 4.4 Primärparser

Zur Verarbeitung der OCL-Constraints wurde ein monadischer Parser gewählt. Der Parser basiert auf der monadischen Parserkombinatoren-Bibliothek *ParSec*. Diese stellt diverse Basisparser und v.a. eine Fehlerbehandlung zur Verfügung.

### 4.4.1 Initialisierung

```
1 import Text.ParserCombinators.Parsec
2 import qualified Text.ParserCombinators.Parsec.Token as P
3 import Text.ParserCombinators.Parsec.Language
```

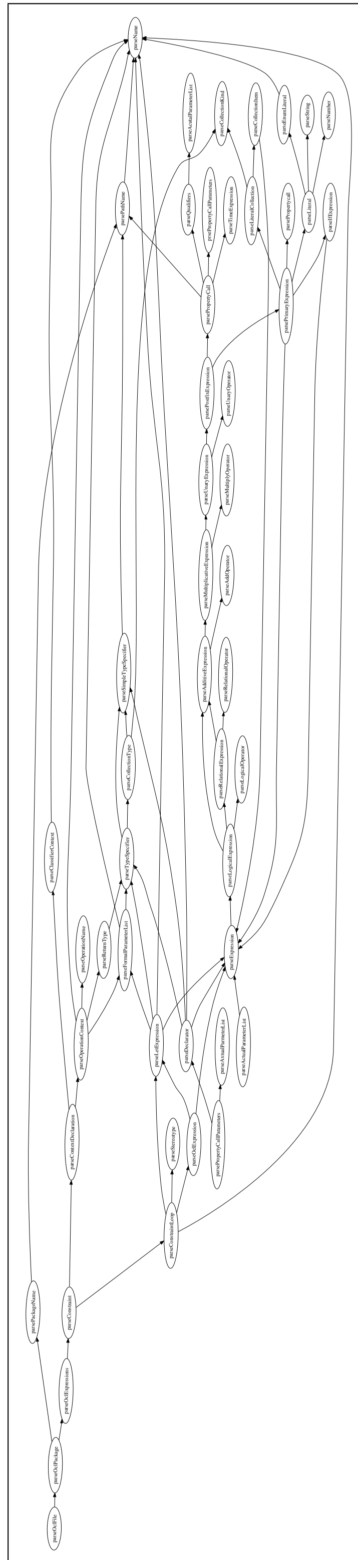

Abbildung 4.2: Schema: OCL-Pimärparser

```
4
5 data Optional a = None | Optional a deriving( Eq, Show, Read )
```

Durch die Zeilen 1-3 werden die nötigen Module der ParSec-Library eingebunden. Der in Zeile 5 definierte Datentyp `Optional` erleichtert im späteren Verlauf die Verwendung von optionalen Produktionssegmenten. Er entspricht abgesehen von der Bezeichnung dem `Maybe`-Typ.

### 4.4.2 abstrakte Syntax

Der durch den Parser erzeugte Parse-Tree soll durch Haskell-Datentypen repräsentiert werden; dies entspricht in etwa der abstrakten Syntax. Der Zusatz `deriving( Eq, Show, Read )` bei jedem Datentyp dient der leichteren Weiterverarbeitung, da dadurch Haskell für jeden Typ den Gleichheitstest bereitstellt und die Standardkonvertierung in und aus einen bzw. einem String ermöglicht.

```
5  data OclFile
     = OclFile [OclPackage] deriving( Eq, Show, Read )
   data OclPackage
     = OclPackage PackageName OclExpressions deriving( Eq, Show, Read )
9  data PackageName
     = PackageName PathName deriving( Eq, Show, Read )
   data OclExpressions
     = OclExpressions [Constraint] deriving( Eq, Show, Read )
   data Constraint
14   = Constraint ContextDeclaration [ConstraintLoop] deriving( Eq, Show,
       Read )
   data ConstraintLoop
     = ConstraintLoopDef (Optional Name) [LetExpression]
     | ConstraintLoopStereo Stereotype (Optional Name) OclExpression
      deriving( Eq, Show, Read )
   data ContextDeclaration
19   = ContextDeclOp OperationContext
     | ContextDeclClass ClassifierContext deriving( Eq, Show, Read )
   data ClassifierContext
     = ClassifierContext1 Name Name
     | ClassifierContext2 Name deriving( Eq, Show, Read )
24 data OperationContext
     = OperationContext Name OperationName FormalParameterList (Optional
      ReturnType) deriving( Eq, Show, Read )
   data Stereotype
     = StereotypePre
     | StereotypePost
29   | StereotypeInv deriving( Eq, Show, Read )
   data OperationName
     = OperationName Name
     | OperationNameEQ
```

```
     | OperationNamePlus
34   | OperationNameMinus
     | OperationNameLT
     | OperationNameLEQ
     | OperationNameGEQ
     | OperationNameGT
39   | OperationNameDiv
     | OperationNameMult
     | OperationNameNEQ
     | OperationNameImplies
     | OperationNameNot
44   | OperationNameOr
     | OperationNameXor
     | OperationNameAnd deriving( Eq, Show, Read )
   data FormalParameterList
     = FormalParameterList (Optional [FormalParameterList1]) deriving( Eq
        , Show, Read )
49 data FormalParameterList1
     = FormalParameterList1 Name TypeSpecifier deriving( Eq, Show, Read )
   data TypeSpecifier
     = TypeSpecifierSimple SimpleTypeSpecifier
     | TypeSpecifierCollection CollectionType deriving( Eq, Show, Read )
54 data CollectionType
     = CollectionType CollectionKind SimpleTypeSpecifier deriving( Eq,
        Show, Read )
   data OclExpression
     = OclExpression (Optional [LetExpression]) Expression deriving( Eq,
        Show, Read )
   data ReturnType
59   = ReturnType TypeSpecifier deriving( Eq, Show, Read )
   data Expression
     = Expression LogicalExpression deriving( Eq, Show, Read )
   data LetExpression
     = LetExpression Name (Optional FormalParameterList) (Optional
        TypeSpecifier) Expression deriving( Eq, Show, Read )
64 data IfExpression
     = IfExpression Expression Expression Expression deriving( Eq, Show,
        Read )
   data LogicalExpression
     = LogicalExpression RelationalExpression [LogicalExpression2]
        deriving( Eq, Show, Read )
   data LogicalExpression2
69   = LogicalExpression2 LogicalOperator AdditiveExpression deriving( Eq
        , Show, Read )
   data RelationalExpression
```

```
    = RelationalExpression AdditiveExpression (Optional
      RelationalExpression2) deriving( Eq, Show, Read )
  data RelationalExpression2
    = RelationalExpression2 RelationalOperator AdditiveExpression
      deriving( Eq, Show, Read )
74 data AdditiveExpression
    = AdditiveExpression MultiplicativeExpression [AdditiveExpression2]
      deriving( Eq, Show, Read )
  data AdditiveExpression2
    = AdditiveExpression2 AddOperator MultiplicativeExpression deriving(
       Eq, Show, Read )
  data MultiplicativeExpression
79  = MultiplicativeExpression UnaryExpression [
      MultiplicativeExpression2] deriving( Eq, Show, Read )
  data MultiplicativeExpression2
    = MultiplicativeExpression2 MultiplyOperator UnaryExpression
      deriving( Eq, Show, Read )
  data UnaryExpression
    = UnaryExpression (Optional UnaryOperator) PostfixExpression
      deriving( Eq, Show, Read )
84 data PostfixExpression
    = PostfixExpression PrimaryExpression [PrimaryExpressionNavig]
      deriving( Eq, Show, Read )
  data PrimaryExpression
    = PrimaryExprLitCol LiteralCollection
    | PrimaryExprLit Literal
89  | PrimaryExprProp PropertyCall
    | PrimaryExprExpr Expression
    | PrimaryExprIf IfExpression deriving( Eq, Show, Read )
  data PrimaryExpressionNavig
    = PrimaryExpressionNavigObject PropertyCall
94  | PrimaryExpressionNavigCol PropertyCall deriving( Eq, Show, Read )
  data PropertyCallParameters
    = PropertyCallParameters (Optional Declarator) (Optional
      ActualParameterList) deriving( Eq, Show, Read )
  data Literal
    = LiteralString String
99  | LiteralNumber Number
    | LiteralEnumLiteral EnumLiteral deriving( Eq, Show, Read )
  data EnumLiteral
    = EnumLiteral [Name] deriving( Eq, Show, Read )
  data SimpleTypeSpecifier
104 = SimpleTypeSpecifier PathName deriving( Eq, Show, Read )
  data LiteralCollection
```

```
  = LiteralCollection CollectionKind [CollectionItem] deriving( Eq,
      Show, Read )
data CollectionItem
  = CollectionItem Expression (Optional Expression) deriving( Eq, Show
      , Read )
data PropertyCall
  = PropertyCall PathName (Optional TimeExpression) (Optional
      Qualifiers) (Optional PropertyCallParameters) deriving( Eq, Show,
      Read )
data Qualifiers
  = Qualifiers ActualParameterList deriving( Eq, Show, Read )
data Declarator
  = Declarator [Name] (Optional SimpleTypeSpecifier) (Optional
      Declarator2) deriving( Eq, Show, Read )
data Declarator2
  = Declarator2 Name TypeSpecifier Expression deriving( Eq, Show, Read
      )
data PathName
  = PathName [Name] deriving( Eq, Show, Read )
data TimeExpression
  = TimeExpressionAtPre deriving( Eq, Show, Read )
data ActualParameterList
  = ActualParameterList [Expression] deriving( Eq, Show, Read )
data LogicalOperator
  = LogicalOperatorAnd
  | LogicalOperatorOr
  | LogicalOperatorXor
  | LogicalOperatorImplies deriving( Eq, Show, Read )
data CollectionKind
  = CollectionKindSet
  | CollectionKindBag
  | CollectionKindSequence
  | CollectionKindCollection deriving( Eq, Show, Read )
data RelationalOperator
  = RelationalOperatorEQ
  | RelationalOperatorGT
  | RelationalOperatorLT
  | RelationalOperatorGEQ
  | RelationalOperatorLEQ
  | RelationalOperatorNEQ deriving( Eq, Show, Read )
data AddOperator
  = AddOperatorPlus
  | AddOperatorMinus deriving( Eq, Show, Read )
data MultiplyOperator
  = MultiplyOperatorMult
```

```
  | MultiplyOperatorDiv deriving( Eq, Show, Read )
data UnaryOperator
  = UnaryOperatorMinus
  | UnaryOperatorNot deriving( Eq, Show, Read )
type Name = String
type Number = Double
```

### 4.4.3 Parserinitialisierung

```
oclDef :: P.LanguageDef st
oclDef = javaStyle {
  reservedNames = [],
  caseSensitive = True
}

lexer :: P.TokenParser st
lexer = P.makeTokenParser oclDef

whiteSpace = P.whiteSpace lexer
symbol s = do { whiteSpace; string s; whiteSpace}
float = P.float lexer
stringLiteral = P.stringLiteral lexer
```

In den Zeilen 1-5 wird eine Sprachdefinition aufgebaut, die im späteren Verlauf z.B. dem TokenParser dazu dient Kommentare zu überlesen. Es wird eine Sprachdefinition erzeugt, die auf dem Java-Style basiert, jedoch keine reservierten Bezeichner enthält.

In Zeile 8 wird nun ein TokenParser erzeugt, der als Sprachdefinition die vorher erzeugte *oclDef* verwendet. Anschließend werden Basisparser erzeugt, indem die durch *ParSec* bereitgestellten Basisparser an den vorher definierten TokenParser *gebunden* werden.

### 4.4.4 Hilfsparser

Im folgenden werden nun einige Hilfsparser bereit gestellt, die die spätere Parserentwicklung erleichtern.

```
parseOptional :: Parser a -> Parser (Optional a)
parseOptional p
  = do {
      x <- try(p);
      return (Optional x);
    }
  <|> return None;
```

Der Parser `parseOptional` dient dazu, optionale Produktionssegmente zu parsen. Er verwendet hierfür den vorher definierten Datentyp `Optional`. Zuerst wird versucht den

übergebenen Parser *p* auf den zu verarbeitenden Stream anzuwenden (4). Bei Erfolg wird das Ergebnis returniert, ansonsten der Konstruktor `None` (7), wodurch signalisiert wird, dass der optional anzuwendende Parser nicht verwendet wurde.

Hier wird auch gleich eine Besonderheit der ParSec-Bibliothek deutlich. Sollte ein Parser versagen, so wird aus Effizienzgründen grundsätzlich im Eingabestream nicht zurückgesprungen. Erst durch die Einbettung des Parsers in `try()` wird sichergestellt, dass bei Versagen des Parsers wieder zurückgesprungen wird. Da häufig das Versagen eines Parsers gleichbedeutend mit dem Scheitern des Gesamtparsers ist, ist dieses Prinzip gerechtfertigt, da es sowohl zu Ressourcen- als auch Geschwindigkeitsvorteilen führt.

```
1 parseList :: Parser a -> Parser [a]
2 parseList p
3   = do {
4     e <- try(p);
5     es <- parseList p;
6     return (e:es);
7   }
8   <|> return [];
```

Der Parser `parseList` dient dazu den übergebenen Parser so oft wie möglich anzuwenden und das Ergebnis als Liste zurückzugeben. Da auch die keinmalige Anwendung erlaubt ist, scheitert dieser Parser nie.

Zu Beginn wird versucht den Parser einmal anzuwenden (4). Scheitert dies, wird die leere Liste returniert (8). Bei Erfolg, wird durch Rekursion die Restliste gelesen (5) und anschließend das Ergebnis (6) zurück gegeben.

```
1 parseList1 :: Parser a -> Parser [a]
2 parseList1 p = do {
3   e <- p;
4   es <- parseList p;
5   return (e:es);
6 }
```

Der Parser `parseList1` wendet den übergebenen Parser so oft wie möglich, aber mindestens einmal an. Dies wird dadurch realisiert, dass der Parser einmal direkt angewendet wird (3) und dann an den vorher definierten Parser `parseList` übergeben wird (4).

```
1 parseSepList :: Parser a -> String -> Parser [a]
2 parseSepList p s = do {
3   x <- p;
4   xs <- parseList (do { symbol s; e<-p; return e});
5   return (x:xs);
6 }
```

Der Parser `parseSepList` dient dazu eine durch ein Symbol getrennte Liste einzulesen, wobei die Liste mindestens ein Element enthalten muss. Zu Beginn wird der Parser direkt angewendet (3). Anschließend wird an den Parser `parseList` ein neu generierter Parser

übergeben (4), der zuerst das Trennzeichen parst und anschließend den übergebenen Parser anwendet.

### 4.4.5 Hauptparser

Der OCL-Parser besteht aus mehreren monadischen Parsern, von denen jeder einzelne die Produktion eines Nichtterminals abdeckt. Da die eigentlichen Parserkombinatoren vom Prinzip identisch sind, soll an dieser Stelle nur exemplarisch auf eine Auswahl eingegangen werden.

```
1 parseLogicalOperator :: Parser LogicalOperator
2 parseLogicalOperator
3   = do { symbol "and"; return LogicalOperatorAnd; }
4   <|> do { symbol "or"; return LogicalOperatorOr; }
5   <|> do { symbol "xor"; return LogicalOperatorXor; }
6   <|> do { symbol "implies"; return LogicalOperatorImplies; }
```

Dieser Parser ist einer der trivialsten, da er nur eines von vier Schlüsselwörtern erlaubt. Bei Erfolg wird dem geparsten logischen Operator der ihn repräsentierende Konstruktor *zugewiesen*. In diesem Beispiel konnte auf die Einbettung mittels `try` verzichtet werden, da

- der Parser direkt bei dem ersten Zeichen scheitert und es somit zu keinem Eingabeverlust kommt
- kein Parser erfolgreich arbeiten kann, wenn ein Parser nach erfolgreichem Einlesen des ersten Zeichens scheitert

```
1 parseCollectionKind :: Parser CollectionKind
2 parseCollectionKind
3   = do { try( symbol "Set" ); return CollectionKindSet; }
4   <|> do { symbol "Sequence"; return CollectionKindSequence; }
5   <|> do { symbol "Bag"; return CollectionKindBag; }
6   <|> do { symbol "Collection"; return CollectionKindCollection; }
```

Bei diesem ebenfalls trivialen Parser konnte auf die Verwendung von `try` nicht verzichtet werden, da der Set-Parser (3) bei einem Einlesen des Strings *Sequence* z.B. erst auf dem dritten Zeichen scheitert. Würde die bisher verarbeitete Eingabe *Se* verworfen, so würde anschließend auch der Sequence-Parser (4) scheitern.

```
1 parseUnaryExpression :: Parser UnaryExpression
2 parseUnaryExpression
3   = do {
4     u <- parseOptional parseUnaryOperator;
5     p <- parsePostfixExpression;
6     return (UnaryExpression u p);
7   }
```

```
8
9 parsePathName :: Parser PathName
10 parsePathName
11   = do {
12     ret <- parseSepList parseName "::";
13     return (PathName ret);
14   }
```

Der Parser `parseUnaryExpression` soll als Beisiel für die Verwendung des Hilfsparsers `parseOptional` und der Parser `parsePathName` als Beispiel für die Verwendung des Hilfsparsers `parseSepList` dienen.

## 4.5 Sekundärparser

Der vorgestellte Parser wurde direkt auf der OCL-Grammatik aufgebaut und repräsentiert diese vollständig. Diese Struktur enthält jedoch sowohl redundante als auch für den Compiler überflüssige Informationen. Zu den redundanten Informationen gehört z.B., dass eine reelle Zahl nicht einfach als *LiteralNumber*, sondern als kompletter Produktionsweg dargestellt wird. So liefert der Parser zum Beispiel für 1.1

```
AdditiveExpression (MultiplicativeExpression (UnaryExpression None (
    PostfixExpression (PrimaryExprLit (LiteralNumber 1.1)) [])) [])
```

als "geparste" Darstellung. Der im Folgenden vorgestellte Sekundärparser (siehe Abbildung 4.3) arbeitet auf der Ausgabe des vorherigen Parsers und liefert für das selbe Literal

```
Lit (LiteralNumber 1.1)
```

als Ausgabe.

Wie man an dem Beispiel sehen kann, entsteht der meiste Overhead dadurch, dass alle xxxExpression-Produktionen optionale Segmente enthalten. Werden diese nicht gesetzt, so wird dennoch entweder ein *None* oder die leere Liste aufgenommen. Die wesentliche Reduktion besteht nun darin, dass in einem Fall, in dem der optionale Teil nicht gewählt wird, direkt in die nächst tiefere Expression-Instanz transformiert wird.

Eine weitere Reduktion der Ausgabe wird dadurch erreicht, dass nur noch die für den Compiler wichtigen Daten erhalten bleiben. So werden alle Informationen oberhalb *oclConstraint* verworfen, und für jedes **context**-Konstrukt ein Tupel mit den spezifizierten Constraints erzeugt.

### 4.5.1 Datenstruktur

```
1 data Context
2   = Method Name [Variable] [Definition] [PreCondition] [PostCondition]
3   | Class Name [Definition] [Invariant] deriving (Eq,Show,Read)
4
```

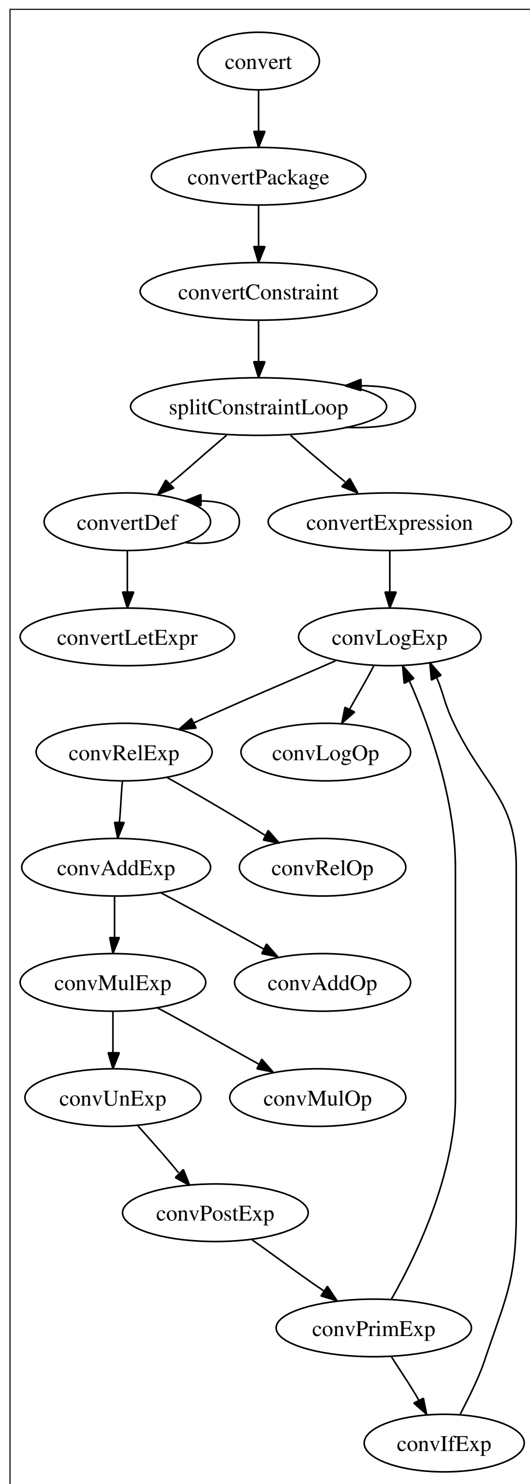


Abbildung 4.3: Schema: OCL-Sekundärparser

```
5 type Name = [String]
6 type Variable = (String,OCL.TypeSpecifier)
7 type PreCondition = Expression
8 type PostCondition = Expression
9 type Invariant = Expression
10
11 data Expression
12  = Expression OCL.OclExpression
13  | LogicalExpression Expression [(LogOp, Expression)]
14  | RelationalExpression Expression RelOp Expression
15  | AdditiveExpression Expression [(AddOp, Expression)]
16  | MultiplicativeExpression Expression [(MulOp, Expression)]
17  | UnaryExpression UnOp Expression
18  | PostfixExpression Expression [PrimaryExpressionNavig]
19  | If Expression Expression Expression
20  | Braces Expression
21  | Lit OCL.Literal
22  | LitCol OCL.LiteralCollection
23  | PropCall OCL.PropertyCall
24  | Tim deriving (Eq,Show,Read)
25
26 data PrimaryExpressionNavig
27  = PropObj OCL.PropertyCall
28  | PropCol OCL.PropertyCall deriving (Eq,Show,Read)
```

Um das vorher erwähnte "durchschleifen" der Typen zu ermöglichen, werden alle Expressions zu einem Datentyp vereinigt.

### 4.5.2 Arbeitsweise

Wie bereits beschrieben, arbeitet der Sekundärparser auf der Ausgabe des Primärparsers. Zu Beginn werden die einzelnen Context-Spezifikationen herausgefiltert und dann separat weiterverarbeitet, wobei zwischen Klasse- und Methodenconstraints unterschieden wird. Als Ergebnis wird eine Liste von vereinfachten Context-Spezifikationen geliefert. Es folgen nun diverse strukturelle Reorganisationen. Diese dienen lediglich dazu, die Daten einfacher zugänglich zu machen und um z.B. Invarianten, Pre- und Postconditions voneinander zu trennen. Auf diese Funktionen wird nicht weiter eingegangen. Die Hauptaufgabe wird nun in den `convertXXX`-Funktionen durchgeführt.

```
1 convLogExp :: Grammar.LogicalExpression -> OCLData.Expression
2 convLogExp (Grammar.LogicalExpression relExp optList)
3  | null optList = convRelExp relExp
4  | otherwise    = OCLData.LogicalExpression (convRelExp relExp) (map
     cv optList)
```

```
5   where cv (Grammar.LogicalExpression2 logOp addExp) = (convLogOp
       logOp,convAddExp addExp )
```

Diese Funktion steht stellvertretend für fast alle der verwendeten convert-Funktionen, da diese vom Aufbau alle identisch sind. Allen diesen Funktionen ist gemeinsam, dass das Ergebnis einen OCL-Ausdruck der Form

$$Expression\ Operator\ \ldots\ Operator\ Expression$$

repräsentieren soll. Ein Unterschied besteht lediglich in der Klasse der möglichen Operatoren und Ausdrücke.

Sollte es sich bei der übergebenen Expression nicht um eine Verknüpfung mehrererer Ausdrücke handeln, so wird der Ausdruck direkt an die nächst tiefere Instanz durchgereicht (3). Nur wenn eine Verknüpfung vorliegt (4), also der optionale Teil gewählt wurde, so wird als Ergebnis ein Ausdruck der gleichen Kategorie - in diesem Fall also eine LogicalExpression - geliefert. Dieser wird nun gebildet, indem jeder Ausdruck für sich übersetzt wird, ebenso wie die einzelnen Operatoren. Die hierbei verwendeten Funktionen zum Überführen der Operatoren vollziehen prinzipiell nur eine Umbenennung der verwendeten Konstruktoren.

Dem vorher beschriebenen Schema folgen nicht die Funktionen zum Überführen von lokalen Definitionen.

```
1  -- Grammer.ConstraintLoopDef -> OCLData.Definition
2  convertDef :: Grammar.Optional String -> [Grammar.LetExpression] ->
      Definition
3  convertDef None exprs = convertDef (Optional "") exprs
4  convertDef (Optional n) exprs = (n, (map convertLetExpr exprs) )
5
6  -- Grammar.LetExpression -> OCLData.LetExpression
7  convertLetExpr :: Grammar.LetExpression -> OCLData.LetExpr
8  convertLetExpr (Grammar.LetExpression n fp t (Grammar.Expression
      logExp)) = (n, fp', t', convLogExp logExp )
9    where t' = case (t) of None -> []; (Optional x) -> [x]
10         fp' = case (fp) of None -> []; (Optional x) -> getParams x
```

Der wesentliche Unterschied besteht jedoch nur darin, dass diese Definitionen benannte und auch parametrisierte Constraints bilden. Diese zusätzlichen Informationen werden entnommen und die Extraktionen der Constraints werden wieder an die vorher beschriebenen *convert*-Funktionen übergeben.

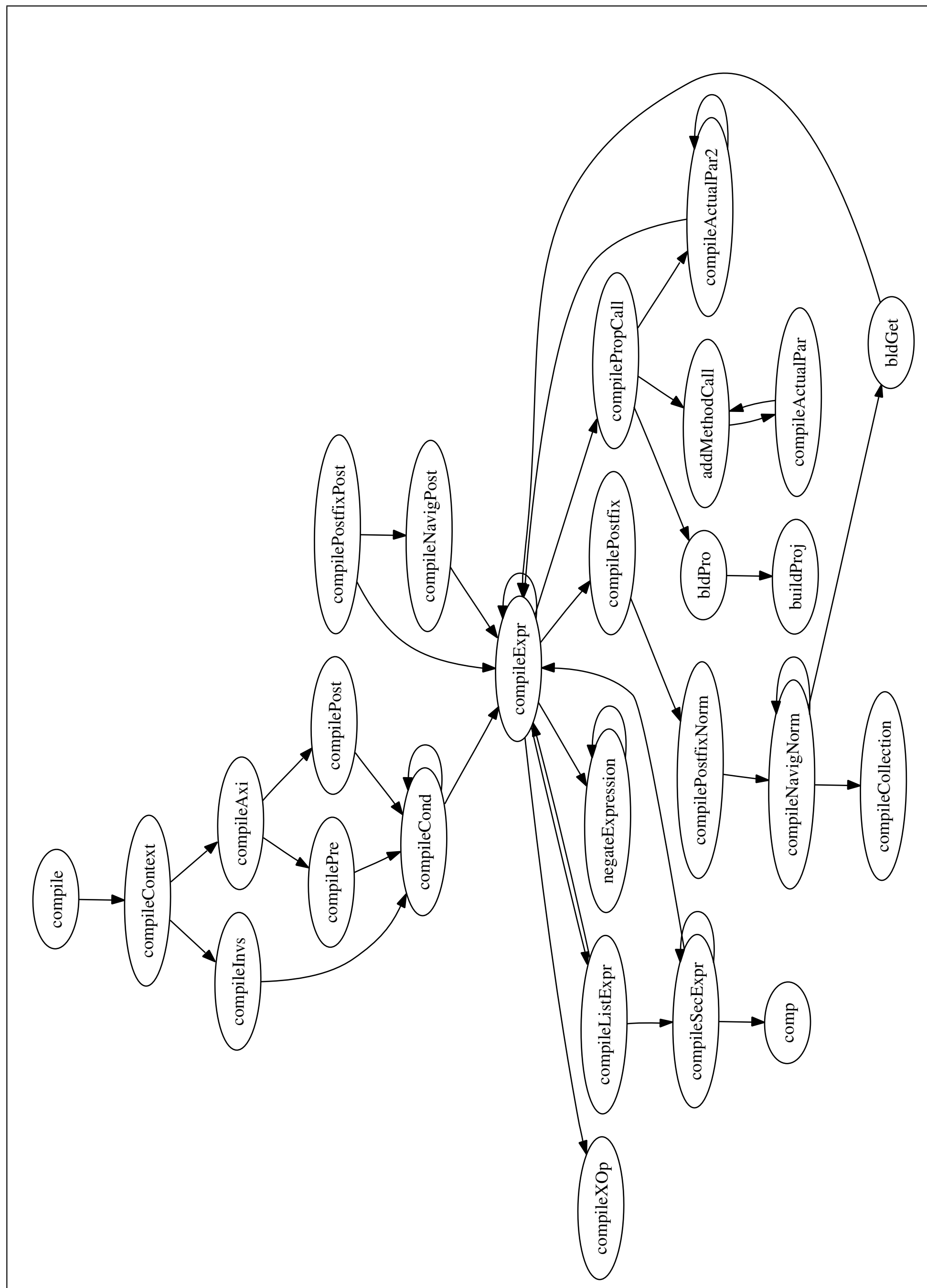


Abbildung 4.4: Schema: OCL-Compiler

# 4.6 Compiler

## 4.6.1 Grundlagen

Aufbauend auf der Ausgabe des Sekundärparsers soll der Compiler (siehe Abbildung 4.4) nun die nötigen *Expander2* -Spezifikationen erstellen; hierzu zählen die Deklaration von

*defuntcs*, *preds*, *fovars* und vor allem von *axioms*. Die einzelnen Context-Deklarationen werden separat bearbeitet.

Zur Bearbeitung gehören sowohl das eher triviale Ermitteln der notwendigen Deklarationen aber hauptsächlich das Aufstellen der notwendigen Axiome. Der Schwerpunkt bei der Aufstellung der Axiome besteht darin, die spezifizierten OCL-Methodenbezeichner den entsprechenden Objektinstanzen oder Übergabeparametern zuzuordnen (also den Kontext zu ermitteln) und auch die Behandlung der Collection-Methoden. Die prinzipielle Herangehensweise für Invarianten und Preconditions ist hierbei praktisch identisch. Der Hauptaugenmerk liegt bei der Axiomatisierung der Postconditions, da hier Rückgabewerte und Objektmanipulation einfließen.

Der Compiler liefert ein Tupel von String-Listen, die die einzelnen *Expander2* -Spezifikationstypen in der Reihenfolge *constructs*, *defuncts*, *preds*, *fovars*, *axioms* darstellen.

### 4.6.2 Deklaration

```
1 compileContext :: [CompileContext] -> O.Context -> Expander ->
    Expander
2 compileContext ctx (O.Class name defs inv) e@(s,c,d,p,f,h,a)
3   = ( s, c, d, inserts p [("inv"++(extractName name)),("is"++(
      extractName name))], f, h, insert a (evalState (compileInvs name
      inv) ((CtxClass (extractName name)):(ctxSetInst ctx "self"))) )
4 compileContext ctx (O.Method name vars defs pre post) e@(s,c,d,p,f,h,a
    ) = ( s, c, insert d (head name), p, f, h, insert a axi )
5   where axi = evalState (compileAxi name pre post) ((CtxParams vars):(
      CtxClass (extractName (tail name)):ctx))
```

Diese Funktion ist die Einstiegsfunktion für das Übersetzen eines Context-Elementes. Die hier verwendeten Hilfsfunktionen *insert* und *inserts* dienen dazu, in die Liste (Parameter 1) einen String bzw. eine Liste von Strings (Parameter 2) einzufügen, wobei keine doppelten Einträge aufgenommen werden.

Der erste Parameter ist das Ergebnis des UML-Parsers. Dies wird im späteren Verlauf benötigt, um z.B. Informationen über an Assoziationen beteiligten Klassen zu erhalten. Der zweite Parameter ist der zu übersetzende Kontext und der dritte Parameter stellt die bisher ermittelten *Expander2* -Spezifikationen dar.

Wie hier bereits durch die Verwendung von *evalState* zu erkennen ist, ist der zentrale Compiler mit Hilfe der Statusmonade implementiert. Dies entspricht letztlich dem zuerst verwendeten Durchschleifen des aktuellen Kontextes durch alle Methoden, erspart jedoch die explizite Spezifikation des Durchschleifens.

### 4.6.3 Hilfsfunktionen

#### Kontext & Statusmonade

Durch die rekursive Struktur des Compilers ist es zwingend erforderlich in jedem Schritt den aktuellen Kontext zur Verfügung zu haben. Dieser wird in Form einer Liste von

*CompileContext*-Elementen bereitgestellt.

```
data CompileContext
  = CtxUML UML
  | CtxParams [O.Variable]
  | CtxClass String
  | CtxConstraint ConstraintType
  | CtxInstance String
  | CtxAux Int
  | CtxModAttr String
  | CtxAttrPre
  deriving (Eq, Show, Read)

data ConstraintType = CTinv | CTpre | CTpost | CTnone deriving (Eq,
    Show, Read)
```

Die Listenstruktur hat den Vorteil, dass der Kontext während der Entwicklungszeit leicht um weitere Elemente erweitert werden kann. Des Weiteren wurden einfache Hilfsfunktionen breitgestellt, die z.B. den Klassennamen extrahieren. Auf diese wird nicht näher eingegangen, da sie alle nach dem gleichen Schema aufgebaut sind: die Liste wird so lange durchlaufen, bis ein Element des gesuchten Typs gefunden wurde, und anschließend wird dessen Wert zurückgegeben.

```
type CompileState = State [CompileContext] String
```

Dies ist ein Alias für die im weiteren Verlauf verwendete Statusmonade. Der Status der Monade ist also [*CompileContext*] und das Ergebnis der Monade ist vom Typ *String*.

### Projektionen

Wie bereits beschrieben, werden Objektinstanzen durch ein Tupel der Attribut-Belegung repräsentiert. Es ist daher erforderlich anhand eines Klassennamens und eines Attributnamens die entsprechende Position im Tupel zu ermitteln und den dann passenden Projektionsausdruck zu konstruieren.

Für die Konstruktion ist zunächst die eindeutige Position des Attributs innerhalb des Tupels notwendig. Zu diesem Zweck wird eine eindeutig nummerierte Liste aller Attribute der Klasse und ihrer Vorfahren erzeugt.

```
getInheritanceAttrList :: UML -> String -> [Element]
getInheritanceAttrList uml name = concatMap (getAttrList uml) (
    getInheritancePath uml name)

getAttrList :: UML -> String -> [Element]
getAttrList uml name = get e
  where (Class _ e) = getClass uml name
        get [] = []
        get ((a@(Attribute _ _)):rest) = a:(get rest)
```

```
9          get ((a@(Association _ _ _)):rest) = a:(get rest)
10         get (_:rest) = get rest
```

Die in (2) verwendete Funktion *getIneritancePath* ermittelt durch rekursives Aufsteigen die Vererbungshierarchie. Da Mehrfachvererbung ausgeschlossen ist, ist diese Reihenfolge eindeutig. Mittels *getAttrList* werden nun aus einer Klasse alle Attribut- und Assoziationselemente entnommen. Da diese Funktion immer wieder auf der gleichen Ausgangsstruktur angewendet wird, ist diese Reihenfolge eindeutig.

```
1  getAttr :: UML -> String -> String -> (Int,String)
2  getAttr uml klasse attr = get e 0
3    where e = getInheritanceAttrList uml klasse
4          get ((a@(Attribute n t)):r) i | (n==attr) = (i,t)
5                                        | otherwise = get r (i+1)
6          get ((a@(Association n t _)):r) i | (n==attr) = (i,t)
7                                            | otherwise = get r (i+1)
8          get [] i = (-i, "" )
9
10 attrIndex :: UML -> String -> String -> Int
11 attrIndex uml klasse attr = i
12   where (i,_) = getAttr uml klasse attr
```

Mittels *getAttr* wird nun die Position des gesuchten Attributes innerhalb der Liste ermittelt. Da diese Methode sowohl den Index als auch den Attributtyp zurückliefert, wird mit der Hilfsfunktion *attrIndex* lediglich der Index extrahiert.

Der eigentliche Projektionsausdruck wird nun durch folgene Funktionen konstruiert.

```
1  buildProj :: String -> String -> String -> CompileState -- Kontext,
      Klasse, Attribut, Instanz
2  buildProj _ "result" "self" = return "result"
3  buildProj klasse attr "self" = do
4    state <- get
5    if ( (isPost state) && (not (isPreAttr state) ) )
6            then do { put (addModAttr state attr); return attr }
7            else return ( buildProj' ( ctxParams state ) 1 (ctxUML state
               ) klasse attr "self" )
8  buildProj klasse attr obj = do
9    state <- get
10   return ( buildProj' [] 1 (ctxUML state) klasse attr obj )
11
12
13 buildProj' :: [O.Variable] -> Int -> UML -> String -> String -> String
       -> String
14 buildProj' ((name,typ):rest) i uml klasse attr obj
15   | name==attr = "get"++(show i)++"(params)"
16   | otherwise = buildProj' rest (i+1) uml klasse attr obj
17 buildProj' [] _ uml klasse attr obj = "get"++(show (attrIndex uml
      klasse attr))++"("++obj++")"
```

Die Art der Konstruktion des Projektionsausdrucks ist abhängig vom aktuellen Kontext. Die Hauptunterscheidung wird hier zwischen Preconditions und Postconditions gemacht. In Preconditions ist nur eine Instanz des Attributes verfügbar, so dass nur getestet werden muss, ob das Attribut ein Klassenattribut oder ein Methodenparameter ist. Anschliessend kann der Index ermittelt und so der Projektionsausdruck gebildet werden.

Innerhalb von Postconditions ist jedes Klassenattribut in zwei Instanzen verfügbar. Einmal der Wert nach Ablauf der Methode und einmal vor Aufruf der Methode. Diese werden unterschiedlich behandelt. Sollte es sich um einen @*pre*-Aufruf handeln - also der Wert vor Methodenaufruf - wird der Projektionsausdruck analog zur Precondition gebildet. Wird jedoch der Wert des Attributs nach Methodenausführung verwendet, so wird das Attribut als Variable in den Term aufgenommen und in dem Status vermerkt, dass an diesem Attribut eine Änderung vorgenommen wurde. Dies ist erforderlich, damit während der Konstruktion der Ergebnisinstanz die modifizierten Attribute übernommen werden können.

### Ergebnisinstanz

Bei der Konstruktion der Ergebnisinstanz fließen etwaige Instanzänderungen der Postconditions ein. In der Statusmonade wurde festgehalten, welche Attribute neue Werte erhalten haben. Diese Auflistung wird nun rekursiv abgearbeitet und mittels eines geeigneten *upd*-Aufrufs in die Rückgabeinstanz eingebettet.

```
1  buildSelf :: CompileState
2  buildSelf = do
3    state <- get;
4    attrs <- return (getModAttr state);
5    uml <- return (ctxUML state);
6    cl <- return (ctxClass state);
7    return (buildSelf' uml cl attrs "self" )
8
9  buildSelf' :: UML -> String -> [String] -> String -> String
10 buildSelf' uml _ [] i = i
11 buildSelf' uml cl (a:r) i = "upd" ++ show(ind) ++"(" ++ i ++", " ++ a
     ++ ")"
12   where ind = attrIndex uml cl a
```

Das Schema ist hierbei relativ simpel. Zuerst wird die Liste der geänderten Attribute ermittelt. Dann wird für jedes Attribut rekursiv mit Hilfe des Attributindex ein geeigneter *upd*-Aufruf gebildet, und so das Tupel der modifizierten Instanz gebildet.

### Hilfsspezifikationen

Durch den Compiler werden eine Reihe von Spezifikationen automatisch in jede erzeugte Spezifikation aufgenommen. Diese dienen sowohl dem einfacheren Compilieren als auch dem effizienteren Auswerten innerhalb des *Expander2* .

### 4.6.4 Elementarcompiler

**Hilfscompiler**

```
compileListExpr :: O.Expression -> ( a-> CompileState ) -> [(a,O.
    Expression)] -> CompileState
compileListExpr expr comp list = do
  e <- compileExpr expr;
  es <- compileSecExpr comp list;
  return (e++es)
```

Wie vorher beschrieben, verfügt ein Großteil der möglichen Ausdrücke über eine Liste aus Operator-Ausdruck-Tuppeln. Diese werden prinzipiell alle nach dem gleichen Schema übersetzt. Hierzu dient diese Hilfsfunktion. Sie erhält als Eingabe den immer vorhandenen Startausdruck, die Funktion, mit der der Operator übersetzt wird, und die Tuppelliste. Es wird nun rekursiv die Liste transformiert.

**Standardausdrücke**

```
compileExpr (O.RelationalExpression expr relOp expr') = do
  e <- compileExpr expr;
  o <- compileRelOp relOp;
  e' <- compileExpr expr';
  return ( e ++ o ++ e' )
compileExpr (O.AdditiveExpression expr list) = compileListExpr expr
    compileAddOp list
compileExpr (O.LogicalExpression expr list) = compileListExpr expr
    compileLogOp list
compileExpr (O.MultiplicativeExpression expr list) = compileListExpr
    expr compileMulOp list
```

Diese Ausdrücke sind in ihrem strukturellen Aufbau identisch, so dass die Übersetzung immer dem gleichen Schema folgt. Zuerst wird der erste Term ”manuell” übersetzt, und dann die Tupel-Liste zusammen mit der jeweiligen Operator-Funktion an die vorher beschriebene Hilfsfunktion übergeben. Die Operatoren werden direkt in die entsprechenden Schlüsselwörter übersetzt, so dass auf diese nicht weiter eingegangen wird.

**Negation**

Die Negation von Expressions wird an zwei Stellen benötigt. Einmal zur Umsetzung des durch OCL bereitgestellten Operators `not` und zur Darstellung des if-Statements.

```
negateExpression :: Expression -> Expression
negateExpression (Braces expr) = (Braces (negateExpression expr) )
negateExpression (UnaryExpression UnNot expr) = expr
negateExpression (If expr expr' expr'') = (If expr expr'' expr')
negateExpression (RelationalExpression expr op expr') = (
    RelationalExpression expr op' expr')
  where op' | op==RelEQ = RelNEQ
```

```
7                 | op==RelGT  = RelLEQ
8                 | op==RelLT  = RelGEQ
9                 | op==RelGEQ = RelLT
10                | op==RelLEQ = RelGT
11                | op==RelNEQ = RelEQ
12 negateExpression expr = expr
```

Wie dem Quellcode zu entnehmen ist, sind nur gewisse Expressionarten direkt von der Negation betroffen; so wird zum Beispiel eine Addition von einer Negation nicht beeinflusst, so dass diese unverändert zurückgegeben wird (12).

Derzeit wird die Negation von logisch verknüpften Listen von Ausdrücken und Methodenaufrufen (hierbei im speziellen jene mit Rückgabetyp *bool* noch nicht behandelt.

Um die Negation einer relationalen Expression zu erhalten, wird einfach direkt der Operator negiert (5-11); d.h. der Gleichheitsoperator wird z.B. durch den Ungleichheitsoperator ersetzt. Bei der Negation einer negierten Expression heben sich die Negationen gegenseitig auf (3) und bei einem geklammerten Ausdruck bleibt die Klammerung erhalten und der Ausdruck in der Klammer wird negiert (2).

```
1 compileExpr (O.UnaryExpression O.UnNot expr) = compileExpr (
    negateExpression expr)
```

Die direkte Negation wird wieder durch die Funktion `compileExpr` ausgelöst. Der Aufruf zur Negation innerhalb einer if-Expression wird im folgenden Abschnitt beschrieben.

### if-Expression

```
1 compileExpr (O.If expr expr' expr'') = do
2   e <- compileExpr expr;
3   ne <- compileExpr (negateExpression expr);
4   e' <- compileExpr expr';
5   e'' <- compileExpr expr'';
6   return ( "␣(␣(␣" ++ e ++ "␣&␣" ++ e' ++ "␣)␣|␣(␣" ++ ne ++ "␣&␣" ++ e
      '' ++ "␣)␣)" )
```

Der Übersetzung einer if-Expression liegt folgendes Schema zu Grunde. Der Ausdruck

`IF expr1 THEN expr2 ELSE expr3 ENDIF`

ist nur dann *wahr*, wenn

- `expr1` und `expr2` *wahr* sind, oder
- `expr1` *unwahr* und `expr3` *wahr* ist.

Dies lässt sich formal als die logische Formel

$$expr_1 \wedge expr_2 \vee \overline{expr_1} \wedge expr_3$$

darstellen. Die obige Funktion überführt nun eine if-Expression in eine diesen logichen Ausdruck repräsentierende *Expander2* -Darstellung.

### 4.6.5 Invarianten, Preconditions & Postconditions

Die initiale Compilierung sieht für die drei Conditionarten beinahe identisch aus. Der Kontext wird um ein Flag erweitert. Dies kennzeichnet, um welche Art von Condition es sich handelt. Anschließend wird die Compilierung der Conditions vorgenommen.

#### Initialcompilierung

Dies soll anhand von Invariantern veranschaulicht werden.

```
1 compileInvs :: [String] -> [O.Expression] -> CompileState
2 compileInvs _ [] = return ""
3 compileInvs name expr = do
4   state <- get;
5   name' <- return (extractName name);
6   put ((CtxConstraint CTinv):state);
7   n <- compileCond name' expr;
8   return ( "(␣inv" ++ name' ++ "(self)" ++ "␣<===␣(␣is" ++ name' ++ "(
      self)␣&␣" ++ n ++ "␣)␣)" )
```

Durch (4)-(6) wird der Status der Monade um das *CTinv*-Flag ergänzt, so dass der weitere Compileverlauf speziell für Invarianten erfolgt. In (7) werden nun die einzelnen Glieder der Invariante compilert und in (8) das Ergebnis - das fertige Invariantenaxiom - gebildet.

Die Initialcompiler für Preconditions und Postconditions sehen beinahe identisch aus, nur dass das $CTinf$-Flag durch $CTpre$ bzw. $CTpost$ ersetzt wird.

#### Condition

```
1 compileCond :: String -> [O.Expression] -> CompileState
2 compileCond name [] = return ""
3 compileCond name (expr:exprs) = do
4   e <- compileExpr expr;
5   e' <- return ( "(" ++ e ++ ")" );
6   e'' <- compileCond name exprs;
7   if (e''=="") then return e'
8                else return ( e' ++ "␣&␣" ++ e'' )
```

Diese Funktion dient dem Transformieren von Conditions. Hierzu wird prinzipiell jede Condition der Liste separat für sich transformiert, und die Ergebnisse anschließend konjunktiv miteinander verknüpft.

### 4.6.6 Propertycall

Alle Zugriffe auf Attribute oder Methoden erfolgen in OCL über den PropertyCall-Typ.

### Basis

Die Unterscheidung um welche Art von Propertycall es sich handelt, wird in der Funktion *compilePropCall* vorgenommen.

```
1 compilePropCall :: [String] -> G.Optional G.TimeExpression -> G.
    Optional G.Qualifiers -> G.Optional G.PropertyCallParameters ->
    CompileState
2 compilePropCall pn G.None G.None G.None = bldPro pn
3 compilePropCall pn G.None G.None (G.Optional (G.PropertyCallParameters
     G.None (G.Optional (G.ActualParameterList expr)))) =
    compileActualPar (head(pn)) expr
4 compilePropCall pn G.None (G.Optional (G.Qualifiers (G.
    ActualParameterList expr))) G.None = compileActualPar (head(pn))
    expr
5 compilePropCall pn G.None G.None (G.Optional (G.PropertyCallParameters
     G.None G.None )) = do
6   state <- get;
7   return ( head(pn)++"(" ++ (ctxInst state) ++ ")" )
8 compilePropCall pn (G.Optional t) G.None G.None = do
9   state <- get;
10  put (ctxAttrPre state);
11  n <- (bldPro pn);
12  return n
```

In (2) wird ein einfacher Attributaufruf ausgewertet. In Zeile (3) und (4) wird jeweils ein einfacher Methodenaufruf mit Parameterliste ausgewertet. Durch die OCL-Grammatik gibt es beide Möglichkeiten dies darzustellen, so dass hier zwei Fälle auf die gleiche Ergebnisfunktion abgebildet werden. In (5)-(7) wird direkt ein Methodenaufruf ohne Parameterliste transformiert. Ab (8) wird schließlich der Zugriff auf ein @*pre*-gekennzeichnetes Attribut innerhlab einer Preconditon ausgewertet. Hierzu wird das *AttrPre*-Flag in den Status aufgenommen, und anschließend die Termbildung wieder an *bldPro* weitergeleitet.

### Parameterliste

Die Transformation der Parameterliste erfolgt mittels *compileActualPar*.

```
1 compileActualPar :: String -> [G.Expression] -> CompileState
2 compileActualPar na [] = do
3   state <- get;
4   return ( na ++ "␣(␣" ++ (ctxInst state) ++ "␣)␣" )
5 compileActualPar na expr = do
6   expr' <- compileActualPar' (map OCL.convertExpression' expr);
7   state <- get;
8   return ( na ++ "␣(␣" ++ (ctxInst state) ++ ",␣" ++ expr' ++ "␣)␣" )
```

Für den Aufruf ohne Parameter wird nur die aktuelle Instanz übergeben (2-4). Eine nicht leere Parameterliste wird in (6-8) ausgewertet. Zunächst werden in einer Hilfsfunktion die einzelnen Parameter übersetzt (6) und anschließend der Aufruf wieder zusätzlich mit der aktuellen Instanz gebildet (8).

```
1 compileActualPar' :: [O.Expression] -> CompileState
2 compileActualPar' [] = return ""
3 compileActualPar' (e:[]) = compileExpr e
4 compileActualPar' (e:es) = do
5   e' <- compileExpr e;
6   es' <- compileActualPar' es;
7   return ( e' ++ ", " ++ es' )
```

Die Bearbeitung der einzelnen Parameter erfolgt rekursiv. Jeder Parameter wird separat übersetzt. Der Ergebnisterm wird direkt returniert. Sollten noch weitere Parameter in der Liste vorhanden sein, wird das Ergebnis zusätzlich um , und das Ergebnis der Transformation der restlichen Parameter erweitert.

### 4.6.7 Postfixexpressions

Die Behandlung von Postfixexpressions hängt davon ab, ob es sich um eine Postcondition oder um eine Precondition bzw. Invariante handelt. Bei Postfixexpressions handelt es sich i.A. um Aufrufe der Form *object-methode(...).attribute.methode(...)....*; z.B.

- `oeffnen()`
- `tuer.oeffnen()`
- `haus.zimmer(2).tuer.oeffnen()`

Der initiale Aufruf erfolgt mittels der Methode *compilePostfixNorm*.

```
1 compilePostfixNorm :: O.Expression -> [O.PrimaryExpressionNavig] ->
      CompileState
2 compilePostfixNorm e navig = do
3   e' <- bldGet e;
4   state <- get;
5   put (ctxSetInst state e');
6   n <- compileNavigNorm navig;
7   return (n)
```

Da der Basisaufruf (3) immer ein Attribut des aktuellen Objektes oder ein Parameter ist, kann dieser wie üblich transformiert werden. Für den weiteren Aufruf der Postfixexpression ist nicht mehr das Objekt die aufrufende Instanz, sondern der zuvor transformierte Basisaufruf. Dieser wird nun als aufrufende Instanz markiert (4+5) und anschließend wird der Rest der Postfixexpression rekursiv abgearbeitet (6).

Diese Bearbeitung ist nun davon abhängig, ob es sich um einen Objekt- oder einen Collectionaufruf handelt. Wir betrachten zunächst den Fall, dass es sich um einen Objektaufruf handelt.

```
1 compileNavigNorm :: [O.PrimaryExpressionNavig] -> CompileState
2 compileNavigNorm ((O.PropObj p):r) = do
3   inst <- compileExpr (O.PropCall p);
4   state <- get;
5   put (ctxSetInst state inst);
6   e <- if null r then return (inst)
7                  else compileNavigNorm r;
8   return (e);
```

Dieser Fall erfolgt praktisch analog zum vorher beschriebenen Basisaufruf. Die anstehende Postfixexpression wird normal übersetzt, zur aktuellen Instanz gesetzt und - falls vorhanden - die restlichen Aufrufe rekursiv abgearbeitet.

Handelt es sich bei der aktuellen Postfixexpression um einen Collectionaufruf, so wird prinzipiell identisch verfahren. Es gibt nur zwei Besonderheiten, die aus den Eigenschaften von Collection-Funktionen resultieren:

1. Es werden keine Methodenaufrufe durch Prädikate erzeugt, sondern direkt die konkrete Axiomatisierung der Collectionfunktion eingesetzt. Dies ist problemlos möglich, da das Verhalten von Collectionfunktionen durch OCL festgelegt ist und daher immer identisch ist.

2. Je nach Art der Funktion sind 0 bis 2 Aufrufparameter möglich. Auch diese werden gesondert behandelt, da diese eine spezielle Form haben und entsprechend direkt weiterverarbeitet werden.

### 4.6.8 Methodenaufrufe

Erfolgt innerhalb der OCL-Constraints ein Methodenaufruf, so wird die Erzeugung der entsprechenden Spezifikation an die Funktion *addMethodCall* delegiert.

```
1 addMethodCall :: String -> [O.Expression]-> CompileState
2 addMethodCall name params = do
3   state'' <- get;
4   res <- getAuxVar;
5   pars <- compileActualPar' params;
6   correct state'';
7   state <- return ((CtxInstTyp (getRetType (ctxUML state'') (ctxClass
       state'') name (ctxClass state''))):state'');
8   inst <- if (isPost state) then getAuxVar else (return (ctxInst state
       ));
9   state' <- get;
10  put (ctxSetInst (ctxAddTmpAxi state' (name ++ "(" ++ (ctxInst state)
        ++ ",["++pars++"],"++ inst ++","++res++")") ) inst);
```

```
11  res' <- if (isVoid (ctxUML state) (ctxClass state) name) then (
        return inst) else (return res);
12  return (res');
```

Das allgemeine Vorgehen ist dabei wie folgt: zunächst werden etwaige Aufrufparameter (4+5) transformiert, dann wird der Aufruf für die die Methode repräsentierenden Prädikate erzeugt (10), wobei zwei Hilfsvariablen (jeweils für Rückgabeergebnis und Rückgabeinstanz) definiert werden.

Dieser erzeugte Prädikataufruf wird nun zwischengespeichert. Anstelle des Methodenaufrufs wird in die Spezifikation eine das Ergebnis repräsentierende Variable aufgenommen (9). Mittels der vorhandenen UML-Struktur wird ermittelt, ob die Methode ein Ergebnis liefert oder vom Typ *void* ist. Im ersten Fall wird die Variable für die Rückgabeinstanz gesetzt, im zweiten Fall die Variable für das Ergebnis.

## 4.7 Bedienung

Die Benutzung des Compilers ist denkbar einfach, da es nur eine Möglichkeit gibt ihn aufzurufen:

```
umlocl2expander uml ocl
```

wobei *uml* der Name der Datei ist, in der sich die XMI-Darstellung des UML-Klassendiagramms befindet, und *ocl* der Name der Datei, in der die OCL-Consraints gespeichert sind. Die erzeugte *Expander2* -Spezifikation wird direkt in der Standardausgabe ausgegeben. Um also z.B. unter Linux eine Spezifikationsdatei zu erhalten, bedient man sich der linuxeigenen Möglichkeiten

```
umlocl2expander uml ocl > spec
```

wobei *spec* nun der Name der Datei ist, in der die Spezifikation gespeichert werden soll.

# Teil III

# Beispiele

# Kapitel 5

# Autorennspiel

## 5.1 Einleitung

Zu Beginn der PG war es gefragt ein Beispiel zu finden, das einerseits nicht zu trivial in der Umsetzung als in den Beweismöglichkeiten sein durfte. Die Wahl fiel dabei neben dem Bank-Beispiel (siehe 6) auf ein Autorennspiel.
Das Spiel beruhte auf simplen Regeln, so dass das Rennauto relativ schnell modelliert werden konnte. Das Auto durfte pro Zug entweder um eins Beschleunigen oder Abbremsen oder, sofern die Geschwindigkeit bei eins lag, um 90° nach links oder rechts drehen. Ein Fahren entgegen der Fahrtrichtung (Start -> Ziel) war nicht erlaubt, genauso wenig durfte der Wagen über den Fahrbahnrand fahren noch mit etwaigen anderen Wagen kollidieren.
Im Ansatz war gedacht, mit diesen Beispiel noch die Vererbung zu zeigen, da ein zweiter Wagen über die Strecke fahren sollte, der alle Eigeneschaften vom "normalen" Auto erben sollte, bis auf den Unterschied, dass der neue Wagen jeweils um zwei Felder beschleunigen oder abbremsen durfte pro Zug.
Da das Autorennspiel mit zu den ersten Beispielen zählt, die bearbeitet wurden, hat es im Laufe der PG eine, nicht gerade kleine, Evolution erlebt. Diese Entwicklung ergibt sich einerseits aus der Erfahrung, die während der Arbeit mit dem *Expander2* gesammelt wurde, als auch, dass das Beispiel ein bisschen vereinfacht werden mußte, um zu einem lauffähigen Beweis zu gelangen.
Diese Entwicklung wird im folgenden an der Streckenmodellierung verdeutlicht werden.

## 5.2 Die Rennstrecke Version 1

### 5.2.1 Umsetzung

Die erste Umsetzung der Modellierung aus UML und OCL-Contraints wurde noch mit den Zwischenschritten über eine Beschreibung in Logik und Haskell-Code gemacht, da zu diesem Zeitpunkt noch keine Erfahrung mit der Zielstruktur gegeben war. Ziel war es,

die Verständlichkeit der Übertragung zu erleichtern.
In Abbildung 5.1 ist zu sehen, wie das erste UML-Diagramm des Autorennspiels aussah.

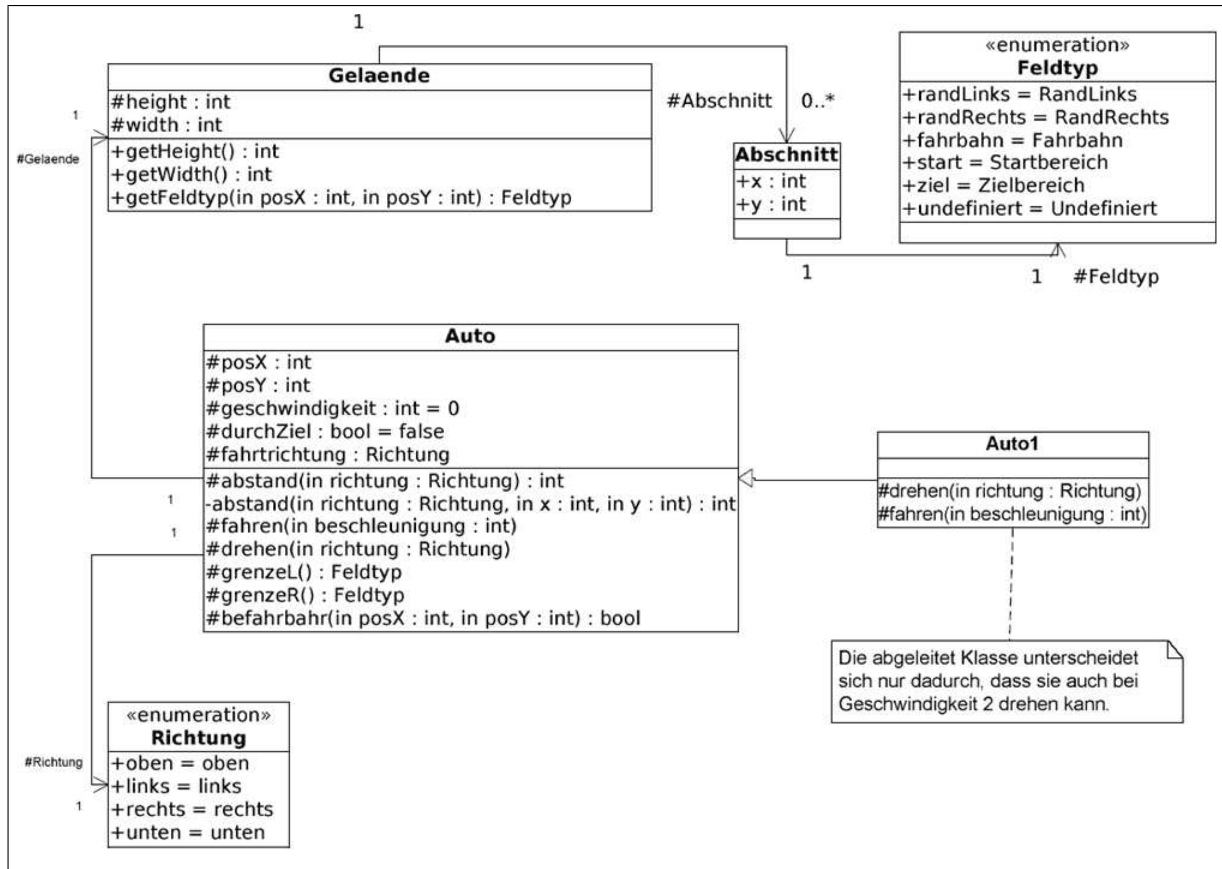


Abbildung 5.1: UML Diagramm, der ersten ”Auto-Version”

### 5.2.2 *Expander2* -Code

Im folgenden werden die einzelnen Passagen der Geländeaxiomatisierung beschrieben.
Ein Gelände wurde zum Beispiel in der Form

```
Gelaende = (10,10,[
Abschnitt(1, 1, RandLinks),
Abschnitt(2, 1, Start),
Abschnitt(3, 1, RandRechts),
Abschnitt(1, 2, RandLinks),
Abschnitt(2, 2, Fahrbahn),
Abschnitt(3, 2, RandRechts),
Abschnitt(1, 3, RandLinks),
Abschnitt(2, 3, Ziel),
```

`Abschnitt(3, 3, RandRechts)])`
initialisiert. Wobei die ersten beiden Dezimalwerte die Dimensionen des Geländes angeben, in X- und Y-Richtung. Der Rest besteht aus einer Liste von `Abschnitt`en, die die Form `Abschnitt(X,Y,Geländetyp)` haben. Wobei Geländetyp einem dieser Werte entspricht:

- Start
- Ziel
- RandLinks
- RandRechts
- Fahrbahn

Ein gültiges Gelände wird nun durch diese Invariante beschrieben. Das in `self` übergebene Gelände wird nun mit Hilfe der durch `aux1` - `aux7` beschriebenen Prämissen auf Gültigkeit geprüft.

```
1  (inv_Gelaende(self) <=== (
2              (all (aux1)(get2(self))) &
3              (all (aux2 (self))(get2(self))) &
4              (all (aux3)(get2(self))) &
5              (all (aux4 (self))(get2(self))) &
6              (0 < get0(self)) &
7              (0 < get1(self)) &
8              (any (aux5)(get2(self))) &
9              (any (aux6)(get2(self))) &
10             (all (aux7 (self))(get2(self)))
11            )) &
```

Die Invariante wird beschrieben durch folgende Elemente:

1. das `all` in Zeile `2` wird `True` zurück, wenn für alle Elemente der Abschnittsliste (`get2(self)`) `aux1` erfüllt ist. `aux1` wird durch dieses Axiom `aux1(abschnitt) <=== 0 < get0(abschnitt)` beschrieben, in dem überprüft wird, ob die X-Koordinate des `Abschnitt`s größer 0 ist

2. wie bei dem ersten Axiom wird für alle Abschnitte durch `aux2` geprüft, ob die X-Koordinate eines Abschnittes <= der Dimension auf der X-Achse ist (`aux2 (self)(abschnitt) <=== get0(abschnitt) <= get0(self)`)

3. `aux3` und `aux4` verlaufen analog `aux1` und `aux2`, nur wird hier die Gültigkeit auf Y-Achse geprüft

4. Zeile 6 und 7 checkt, ob die Dimensionen des Geländes größer 0 sind

5. `aux5` ist erfüllt, wenn im Gelände ein "Start" gegeben ist (`aux5(abschnitt) <=== get2(abschnitt) = Start`)

6. `aux6` analog dazu, wenn ein "Ziel" gefunden wird

7. `aux7` - gegeben durch

```
1 aux7 (self)(abschnitt) <=== ((abschnitt =/= Fahrbahn)
2          | ((getFeldtyp_Gelaende(self,get0(abschnitt)+1,get1(
              abschnitt))
3            `in` [Ziel, Start, Fahrbahn, RandLinks, RandRechts])
              &
4            (getFeldtyp_Gelaende(self,get0(abschnitt)-1,get1(
              abschnitt))
5            `in` [Ziel, Start, Fahrbahn, RandLinks, RandRechts])
              &
6            (getFeldtyp_Gelaende(self,get0(abschnitt),get1(
              abschnitt)+1)
7            `in` [Ziel, Start, Fahrbahn, RandLinks, RandRechts])
              &
8            (getFeldtyp_Gelaende(self,get0(abschnitt),get1(
              abschnitt)-1)
9            `in` [Ziel, Start, Fahrbahn, RandLinks, RandRechts]))
              )
```

- überprüft für jedes Feld, ob die es umrandenden Felder gültige Elemente der Liste `[Ziel, Start, Fahrbahn, RandLinks, RandRechts]` sind

Sind nun alle Prämissen erfüllt, so wird die gesamte Invariante `True`.

## 5.3 Entwicklung

Während der gesamten Zeit, in der an dem Beispiel gearbeitet wurde, hat es sich kontinuierlich weiterentwickelt, da die gewonnenen Erfahrungen im Umgang mit dem *Expander2* und der Entwicklung der Parser / Compiler mit eingeflossen sind.
Durch die erste Modellierung der Rennstrecke war es unmöglich einen Beweis zu verfolgen, der versucht, die Gültigkeit des Autorennspiels für alle gültigen Rennstrecken zu beweisen. Es war kein Problem zu zeigen, dass das Auto über ein gegebenes Gelände fahren konnte, doch sobald versucht wurde, dies für alle möglichen Gelände zu zeigen, war es durch den gegebenen Ansatz unmöglich.

Nachdem einige Versuche erfolglos verliefen, brachte eine neue Betrachtung des Spieles und der Strecke den entscheidenden Fortschritt. Neben einigen Vereinfachungen und

Optimierungen in der Automodellierung[1] wurde die Rennstrecke ausschlaggebend abstrahiert. Anstelle einer "realistisch" abgebildeten Strecke, wurde die gesamte Fahrbahn auf eine ein Feld breite Bahn minimiert, an deren Anfang der Start und Ende das Ziel ist. Die Idee dahinter ist es, die Fahrbahn induktiv beschreiben zu können, da man immer nur zeigen muss, dass das nächste Feld gültig ist, denn alle Felder davor erfüllen dies bereits. Wenn nun alle Felder miteinander verbunden sind, so kann das Rennauto vom Start zum Ziel fahren.

## 5.4 Die Rennstrecke Version 2

### 5.4.1 Umsetzung

Im Gegensatz zu den vorhergegangen Implementierungen wurde diese Version vom Compiler aus dem erzeugt, was die Parser aus dem UML und OCL generiert hatten. Der so enstandene *Expander2* -Code wurde noch ein bisschen angepaßt, weil zu dem Zeitpunkt an einigen Stellen nicht konkrete Werte sondern Variablen eingesetzt wurden, was einen Beweis unnötig verkomplizierte[2].

### 5.4.2 *Expander2* -Code

Die Klasse `Point2D` bildet die Teile aus der später die Strecke bestehen wird. Ein Punkt besteht jeweils aus der KlassenID und der x- und y-Koordinate.

```
1  -- Point2D := [ 0, x, y]
2    ( isPoint2D(self) <=== get0(self) = 0 & length(self) >= 3 )
3  & ( invPoint2D([0,x,y]) <=== Nat(x) & Nat(y) )
4  & ( Point2D__create([x,y]) = [ 0, x, y ] )
```

Die Invariante des Geländes ist nun erfüllt, wenn `verbundGueltig` ein `True` zurückliefert und wenn "start"- und "ziel"-Punkt ein Nachbarfeld von "fahrbahn" sind (`isNachbar( self, [start, fahrbahn ], self, true )`. Wie das abläuft sieht man im folgenden.

```
1  -- Gelaende := [ 1, breite , hoehe, fahrbahn, start , ziel ]
2  & ( Gelaende__create([breite,hoehe,fahrbahn,start,ziel]) = [1,breite
       ,hoehe,fahrbahn,start,ziel] )
3  & ( isGelaende(self) <=== get0(self)=1 & length(self) >= 6 )
4  & ( invGelaende([1,breite,hoehe,fahrbahn,start,ziel])
5       <=== self=[1,breite,hoehe,fahrbahn,start,ziel]
```

[1] es wurde unter anderem auf eine Vererbung wie `auto1` in der erste Version verzichtet

[2] es wurde zum Beispiel `params` anstelle von `[breite,hoehe,fahrbahn,start,ziel]` eingesetzt

```
6           & verbundGueltig( self, [fahrbahn], self, true )
7         & isNachbar( self, [start, fahrbahn ], self, true )
8         & isNachbar( self, [ziel, fahrbahn ], self, true ) )
```

`verbundGueltig` ist sofort `true`, wenn nur noch ein `Point2D` (`[0,x,y]`) in der Fahrbahnliste über ist, wenn mehr Elemente in der Liste sind, so wird das erste Element und die Restliste an `isNachbar` weitergereicht und die Restliste wiederum an `verbundGueltig`. Der untere Teil behandelt die Fälle, in denen `verbundGueltig` `false` liefert.

```
1   -- verbundGueltig( [Point2D] )
2   & ( verbundGueltig( self, [[0,x,y]], self, true ) <=== isGelaende(
      self) )
3   & ( verbundGueltig( self, x:xs, self, true ) <=== isGelaende(self)
4     & isNachbar( self, [x,xs], self, true )
5     & verbundGueltig( self, xs, self, true )
6     )
7   & ( verbundGueltig( self, x:xs, self, false ) <=== isGelaende(self)
8     &   ( isNachbar( self, [x,xs], self, false )
9       |  verbundGueltig( self, xs, self, false ) )
10    )
```

In `isNachbar` werden alle Punkte erzeugt, die den ersten `Point2D`, der übergeben wurde, in x- und y-Richtung umgeben. Es wird `true` zurückgegeben, wenn diese Punkte in der Restliste enthalten sind. Ebenso wird `true` ausgegeben, wenn die Restliste keine Elemente mehr enthält.

```
1   & ( isNachbar( self, [[0,x,y],l], self, result ) <=== isGelaende(
      self)
2     &   ( result = eqF(0)(length(l)) )
3       | ( result = [0,x,y+1] `elem` l )
4       | ( result = [0,x,y-1] `elem` l )
5       | ( result = [0,x+1,y] `elem` l )
6       | ( result = [0,x-1,y] `elem` l )
7     )
```

Auf diese Weise wird die gesamte Fahrbahnliste abgearbeitet und überprüft, ob alle Felder der Liste miteinander verbunden sind. Wenn nun ebenfalls die ”start”- und ”ziele”-Punkte Elemente der Liste sind, so ist die Invariante `true`.

Mit dieser Art der ”Wegbeschreibung” war es nun möglich die Strecke rekursiv zu modellieren, damit ein Beweis per Fixpunktinduktion möglich ist.

## 5.5 Beweis

Ein Beweis, den man auf diesem Modell führen kann und der geführt wurde, ist der Beweis der Aussage "auf jedem gültigem Gelände gibt es einen Weg vom Start zum Ziel, der von einem Auto abgefahren werden kann". Dieser Beweis wurde aufgrund der Grösse nicht in einem geführt, sondern in die folgenden drei Teilbeweise aufgeteilt:

1. jeder Pfad kann von einem Auto befahren werden
2. jeder Start-Ziel-Pfad ist ein Pfad
3. jedes gültige Gelände verfügt über mindestens einen Start-Ziel-Pfad

Durch die Spezifizierung sind die Beweise 2 und 3 sehr leicht zu führen. Beweis 1 ist nicht viel komplexer aber durch die Grösse erheblich aufwändiger. Diese Aussage wird formal dargestellt durch

```
1     isPfad([1,breite,hoehe,fahrbahn,start,ziel],weg)
2 ==> faehrtPfad( [ 2, head(weg), [1, breite, hoehe, fahrbahn, start,
    ziel], 1, 0, false], weg)
```

Hierbei steht `[1,breite,hoehe,fahrbahn,start,ziel]` für jedes beliebige Gelände. Der erste Parameter bei *faehrtPfad* entspricht einem beliebigen Auto, das sich auf dem vorher festgelegtem beliebigen Gelände befindet, und o.B.d.A. mit Geschwindigkeit 1 Richtung Norden fährt. Dies ist keine Einschränkung an den Beweis, da es sich um einen Existenzbeweis handelt und es so mit genügt zu zeigen, dass ein Auto existiert, dass den Pfad entlang fahren kann.

Auf diesen Baum wird nun eine Fixpunktinduktion mittels der folgenden Axiome angewandt.

```
1   isPfad(self0,x0:[])
2 & (      isPfad(self1,x1:(y0:s0))
3    <===    isNachbar(self1,[x1,[y0]],self1,true)
4          & isPfad(self1,y0:s0))
```

Ab diesem Punkt ist nur noch eine geeignete Axiomanwendung o.ä. notwendig um die Aussage zu beweisen. Zwischenzeitlich werden sehr große Teilbäume generiert, die sich aber einfach auflösen lassen. Da gleich zu Beginn die Anwendung des Axioms

```
1      isNachbar(self,[[0,x,y],l],self,result)
2 <===      isGelaende(self)
3         & result = eqF(0)(length(l))
4       | result = [0,x,y+1]`elem`l
5       | result = [0,x,y-1]`elem`l
6       | result = [0,x+1,y]`elem`l
7       | result = [0,x-1,y]`elem`l
```

notwendig ist, entstehen vier Fälle, die sich nur in den relativen Koordinaten unterscheiden. Diese vier Fälle können nahezu identisch abgearbeitet werden.

An dieser Stelle nur noch die letzten Schritte

```
1  All x1 y0 s0 breite1 hoehe1 fahrbahn1 start1 ziel1:
2  (       Any x2 y1:
3          (   [0,x2,y1-1] = y0
4           & x1 = [0,x2,y1])
5       & faehrtPfad([2,y0,[1,breite1,hoehe1,fahrbahn1,start1,ziel1],1,0,
           false],y0:s0)
6   ==> Any x6 y5:
7       (   y0 = [0,x6,y5-1]
8        & x1 = [0,x6,y5]))
9
10 Simplifying (2 steps) the preceding tree leads to the formula
11
12 True
```

# Kapitel 6

# Bank

## 6.1 Einleitung

Zeitgleich zum Autorennspiel wurden weitere Beispiele gesucht. Das Bank- bzw. Geldautomatenbeispiel wurde ausgewählt, da es uns zu dem Zeitpunkt als geeignet erschien. Beschrieben wird der Ablauf einer Geldabhebung an einem Automaten, dazu gehört die Überprüfung, ob die Buchung zulässig ist, sprich genug Geld auf dem, zur Karte gehörenden, Konto in ausreichender Menge zur Verfügung steht.

## 6.2 Umsetzung und Verwurf

Die Idee wurde zunächst mit einem UML-Diagramm beschrieben.

Das Besondere daran war allerdings, dass es sich recht bald während der Umsetzung zeigte, dass es leider nicht für unsere Zwecke geeignet war. Das lag daran, dass die Abarbeitung der einzelnen Schritte bei einer Abhebung linear war, also eine Funktion der anderen folgte und es keine Entscheidungen gab, die eine andere Möglichkeit als den nächsten Schritt zuließen. Aus diesem Grund sind mit dem Beispiel, so wie es modelliert wurde, leider nur Invariantenbeweise möglich, die uns allerdings nicht in die Richtung weiterbrachten, in die wir wollten.
Daher wurde das Beispiel verworfen, bevor es eine Umsetzung für den *Expander2* überhaupt begann.

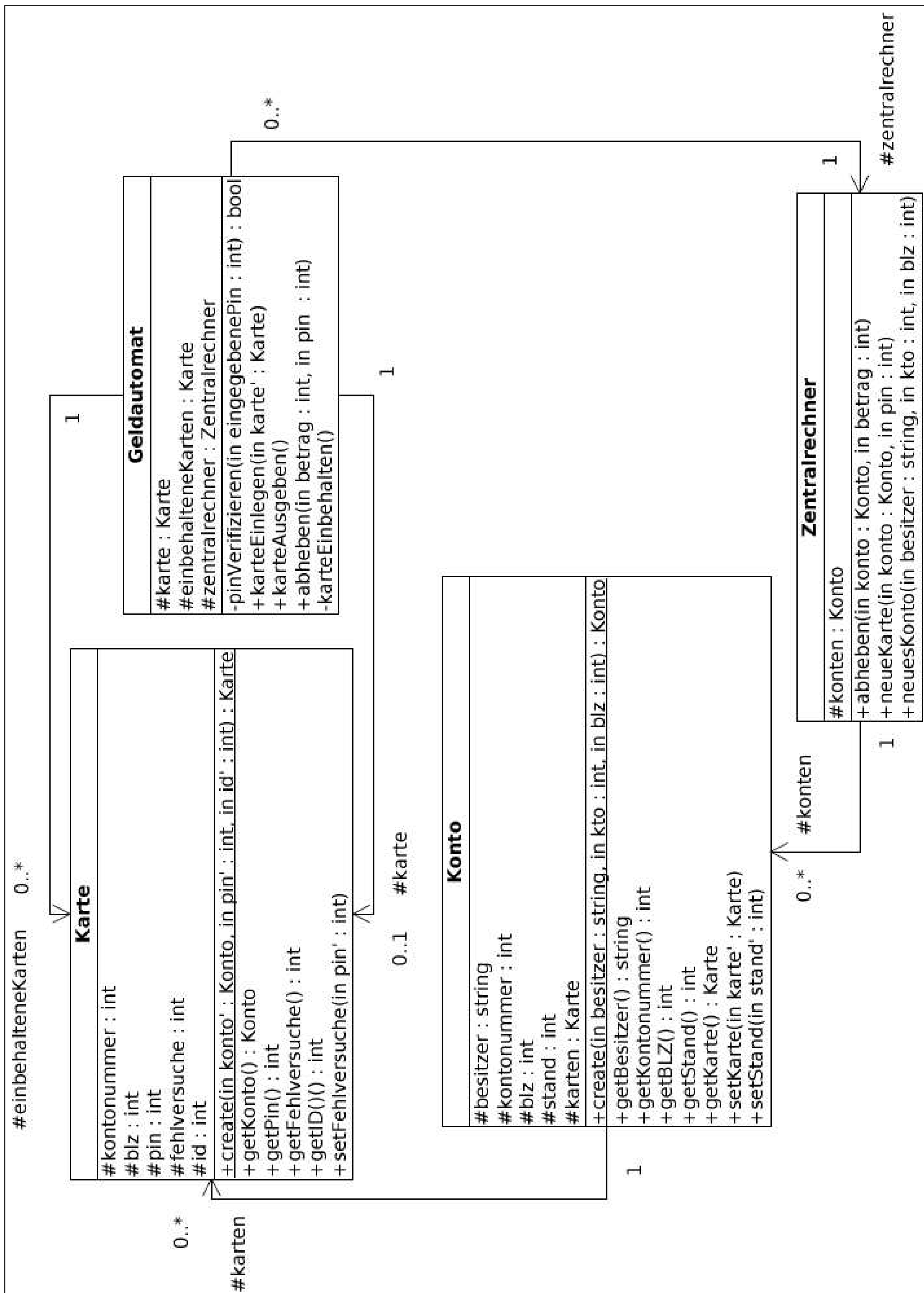


Abbildung 6.1: UML Diagramm des Bankbeispiels

# Kapitel 7

# Mobile Phone System

## 7.1 Einleitung

Nachdem sich leider das Bankbeispiel als nicht geeignet erwies, mußte wieder ein neues Beispiel gesucht werden.
Es wurde eine Umsetzung eines Mobile Phone System in Object-Z [40] gefunden. Das Beispiel war insofern interessant, als dass nicht nur die Modellierung beschrieben war, sondern auch gleich Beweismöglichkeiten beschrieben wurden.
Das Problem war nun, aus der Sprache Object-Z, die komplexe mathematische Konstrukte erlaubt und eine Umsetzung in OCL nicht gerade erleichtert. Nachdem eine erste OCL-Übersetzung gemacht hatte, wurde ein UML-Diagramm mit Together erstellt. Aufgrund von Problemen bei der Umsetzung, wurde das OCL später noch einmal neu umgesetzt.

## 7.2 Beschreibung

Bei dem `Mobile Phone System` handelt es sich um eine Beschreibung eines Systems, in dem sich eine Menge Zellen und eine Menge Handys befinden. Es stehen in diesem System Funktionen zur Verfügung mit denen ein Handy angerufen werden kann oder von sich aus einen Anruf tätigen kann. Wenn nun ein Handy angerunfen wird oder anruft, wird es an eine Zelle gebunden, indem beide, Handy und Zelle, eine Frequenz benutzen, die aus der Menge der zur Verfügung stehenden Frequenzen der Zelle, herausgenommen wird.

Die Beweisidee ist, zu zeigen, dass ein Handy nicht mit mehr als einer Zelle gleichzeitig verbunden sein kann.

Die Umsetzung der einzelnen Klassen in *Expander2* -Code wurde in Gruppen bearbeitet. Wie man dem UML-Diagramm entnehmen kann, ist dieses Beispiel alles andere als trivial in der Umsetzung, hier exemplarisch gezeigt an zwei Klassen:
`mobileState` und `freqAllocator`

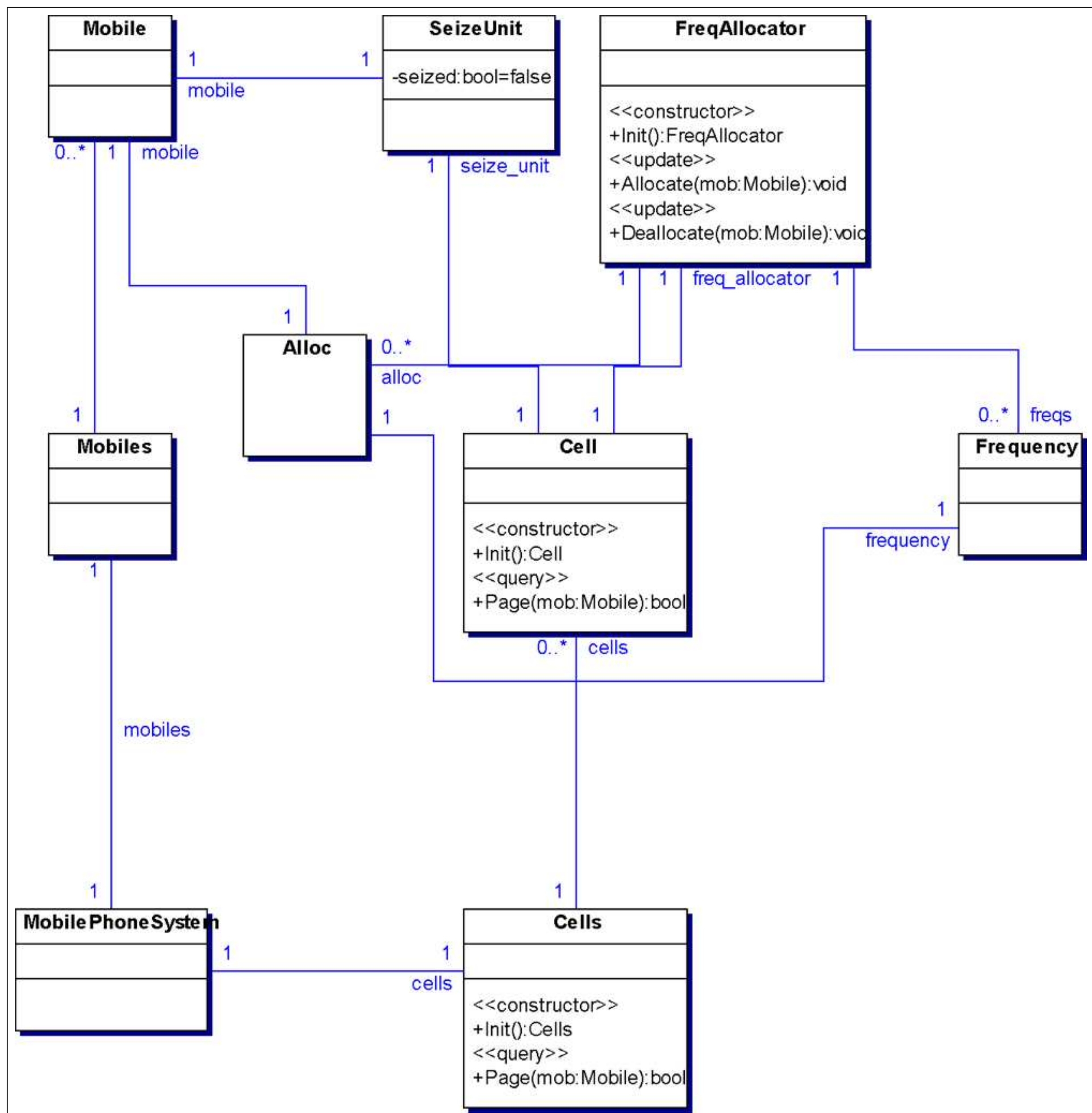


Abbildung 7.1: UML Diagramm - Mobile Phone System (leider nicht aktuell)

### 7.2.1 mobileState

Diese Klasse stellt das Grundgerüst dar, von dem die Klasse `mobile` erbt. Hier werden Zustände definiert, die ein Handy einnehmen kann. Welche das sind, hängt davon ab, welche Funktion aufgerufen wurde.
Wird das Handy angerufen, so wird es zuerst gepaged, dadurch ändert sich der Zustand von 0 (`idle`) auf 1 (`paged`)

```
1 ((self = 0) & (self' = 1) ==> bePaged(self) = (self')) &
```

Initiiert das Handy jedoch selbst eine Verbindung so wechselt der Zustand von 0 (`idle`) auf 5 (`request`)

```
1 ((self = 0) & (self' = 5) ==> request(self) = (self')) &
```

### 7.2.2 freqAllocator

Während nun `mobileState` sehr schnell und einfach umzusetzen war, kam es in anderen Klassen schon eher zu relativ abenteuerlichen Konstrukten, wie im folgenden zu sehen ist.
Die Klasse `freqAllocator` kümmert sich im System darum, die Frequenzen einer Zelle zu verwalten, in dem sie passende Funktionen zur Verfügung stellt:

```
1 ((get1(get0(get1(self))) ‘not_in‘ get0(get0(self))) &
2 (self' = (((get1(get0(get1(self))):get0(get0(self))),
3 (get1(get0(self))−get1(get0(get1(self))))),
4 get1(get0(get1(self))))) ==> freqAllocate(self) = (self')) &
```

```
1 ((get1(get0(get1(self))) ‘in‘ get0(get0(self))) &
2 (self' = (((get0(get0(self))−get1(get0(get1(self))))
3 ,get1(get0(get1(self))):get1(get0(self)))))
4  ==> freqDeAllocate(self) = (self'))
```

## 7.3 Probleme

Wie man an der Art der Umsetzung sieht, stammt sie noch aus einer frühen Zeit, in es noch auf eine lauffähige Version ankam, jedoch noch nicht auf eine effiziente Weise den Beweis durchzuführen. Es wurde zum einen noch mit Guards gearbeitet, als auch viel zu exzessiv mit `get`.

War das Bankbeispiel zu trivial, so scheiterte die Umsetzung des `Mobile Phone Systems` daran, dass es einerseits sehr schwer aus dem Object-Z zu übersetzen war und andererseits durch einen hohen Grad an Parallelität sehr komplex ist. Was an dem Diagramm 7.2 der Aufruffolge bei einem Paging eines Handys im System zu sehen ist.

## 7.4 Fazit

Was während der Umsetzung noch nicht klar war, war, dass so wie es in Object-Z modelliert war, eher dazu diente die Modellierungssprache Object-Z zu präsentieren, als eine

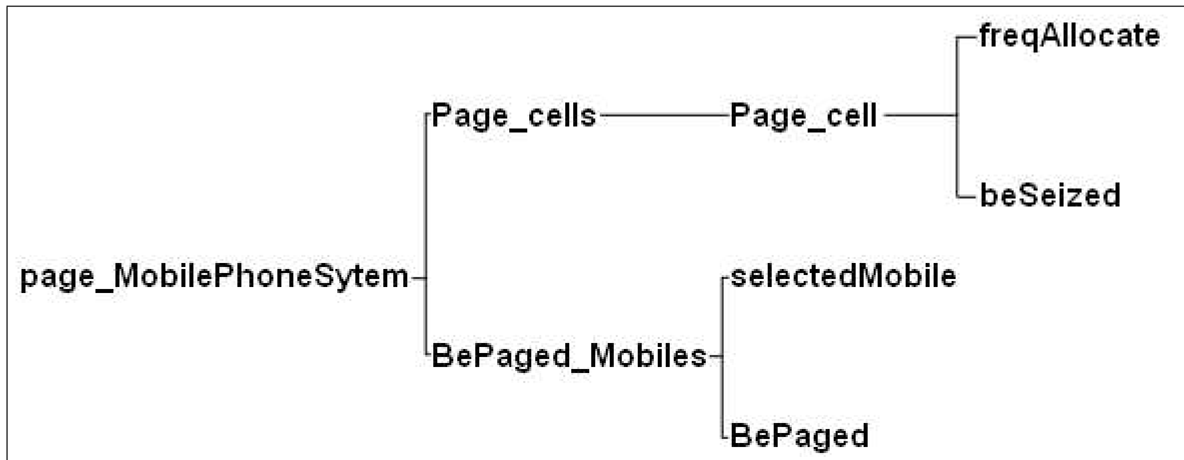


Abbildung 7.2: Parallelität im Mobile Phone System

vollständige Umsetzung des `Mobile Phone System`s zu zeigen. Das führte dazu, dass es bei einigen Funktionen zu Problemen kam, auf Daten zuzugreifen, die in der Object-Z-Modellierung nicht spezifiziert waren. Dieses Problem wurde leider erst gegen Ende der Umsetzung klar, als das Zusammenspiel der Klassen eingearbeitet wurde. Damit eine Beweisführung möglich gewesen wäre, hätte das Beispiel erweitert werden müssen, was aufgrund der Komplexität zeitlich nicht möglich war. Das sorgte letztendlich dafür, dass es nur möglich war, Invariantenbeweise zu führen. Deren Umsetzung machte dann aber keine Schwierigkeiten.
Allerdings konnten anhand dieses Beispieles einige Optimierungen im Umgang mit dem *Expander2* gefunden werden. Der erste Schritt im *Expander2* daurte in den ersten Versionen knappe 24 Minuten, dies konnte durch Optimierungen in mehreren Schritten letztendlich auf knappe 30 Sekunden gedrückt werden.
Mit dem jetzigen Wissensstand wäre es sicherlich möglich, das gesamte System effizienter und beweistauglicher umzusetzen, dies setzt jedoch eine Neumodellierung voraus.

# Kapitel 8

# H-Bahn

## 8.1 Einleitung

Kurz vor Ende der Projektgruppenarbeit mußte im Rahmen des Abschlußvortrags ein Beispiel gefunden werden, dass nicht zu komplex ist, dessen Beweis der Gültigkeit, aber nicht zu trivial ist, jedoch in einem Rahmen bleibt, der sich innerhalb von 15 Minuten vorführen läßt. Aus diesem Grund fiel die Wahl auf die Modellierung der H-Bahn, so war es nicht nur ein kurzes und knappen Beispiel, sondern hatte auch gleich Bezug zur Uni und war für die Zuhörer des Vortrags vielleicht fassbarer, als ein kurzer Beweis eines künstlich geschaffenen Theoretikums.
Aufgrund der Kürze dieses Beispiel, wird es an dieser Stelle komplett erläutert.

## 8.2 Beschreibung

### 8.2.1 Das Bahnsystem

Das H-Bahn-System besteht aus drei H-Bahnen, von der jede eine Streckenliste hat. Die Nummern in der Liste, entsprechen den realen Station an der Universität Dortmund, also:

1. Eichlinghofen
2. Campus-Süd
3. Campus-Nord
4. S-Universität
5. Technologiepark

### 8.2.2 bahn und hbahn

Somit sehen die H-Bahnen und das H-Bahn-System im *Expander2* -Code folgendermaßen aus:

```
(bahn1 = (1:2:4:5:4:[2])) &
(bahn2 = (5:4:2:1:2:[4])) &
(bahn3 = (3:2:3:2:3:[2])) &

(hbahn(bahn1,bahn2,bahn3)) &
(hbahn(1:2:4:5:4:2:l1,5:4:2:1:2:4:l2,3:2:3:2:3:2:l3) <=== hbahn(l1,l2,
    l3) )
```

### 8.2.3 faehrt

Das Abfahren der Strecke funktioniert auf die Weise, dass das erste Element der Liste abgeschnitten wird, wenn die Bahn eine Station weitergefahren ist. Da sich auf einem Abschnitt zwischen zwei Station nicht zwei Bahnen entgegen kommen dürfen, muss vorher überprüft werden, ob eine Kombination vorliegt, die nicht erlaubt ist. Wie das Funktioniert wird in den folgenden Codeabschnitten gezeigt:

```
((wegfrei([a:al,b:bl,c:cl]) = (0,0,0)) ==> faehrt([a:al,b:bl,c:cl]) =
    [al,bl,cl]) &
((wegfrei([a:al,b:bl,c:cl]) = (1,0,0)) ==> faehrt([a:al,b:bl,c:cl]) =
    [al,faehrtnach(b:bl),cl] ) &
((wegfrei([a:al,b:bl,c:cl]) = (1,0,0)) ==> faehrt([a:al,b:bl,c:cl]) =
    [faehrtnach(a:al),bl,cl] ) &
...
...
```

Wie man sieht, je nachdem welches Ergebnis ”wegfrei” liefert, ergibt sich, welche Bahn fahren darf und welche der Bahnen halten muss, da ihr eine andere entgegenkommt. Das Warten wird durch die funktion ”faehrtnach” realisiert, was nichts anderes macht, als das erste Element der Liste abzuschneiden, also die Bahn, wie der Name schon sagt, hinterherfahren läßt. Dies ist nötig, da die Listen für den Beweis eine identische Länge haben müssen.

### 8.2.4 wegfrei, isFrei, frei und check

Die Prämisse ”wegfrei” ruft selbst ”isFrei” auf, dies geschieht mit jeder Kombination der Bahnen.

```
(wegfrei([al,bl,cl]) = (isFrei(al,bl),isFrei(bl,cl),isFrei(al,cl))) &
```

”isFrei” selbst nimmt den Wert an, der aus der Überprüfung der Bahnstrecken ermittelt wird, 0 oder 1.

```
1 (frei(al,bl,result) ==> isFrei(al,bl) = result) &
```

In ”frei” wird nun überprüft, ob es mit den beiden Bahnstrecken - den übergebenen Listen - zu Konflikten kommen kann. Dies kann nur der Fall sein, wenn beide Listen noch mindestens zwei Elemente in der Liste haben, sollte dies nicht der Fall sein, so kann es zu keinen Problemen kommen, da mindestens eine der Bahnen schon an der letzten Station angekommen ist und somit nicht mehr weiterfährt.

```
1 (frei(al,bl,result) <=== (al=(a1:a2:as)) & (bl=(b1:b2:bs)) & result=(
     check(a1,a2,b1,b2))) &
2 (frei(al,b,result)  <=== (al=(a1:[]))    & (bl=(b1:b2:bs)) & result=0)
     &
3 (frei(al,bl,result) <=== (al=(a1:a2:as)) & (bl=(b1:[]))    & result=0)
     &
4 (frei(al,bl,result) <=== (al=(a1:[]))    & (bl=(b1:[]))    & result=0)
     &
```

Sind nun mindestens noch zwei Elemente in jeder der Listen, so werden jeweils die ersten beiden Elemente an ”check” übergeben, wo überprüft wird, ob die vier Elemente eine illegale Kombination bilden. Diese Kombinationen sind in ”signal” gespeichert. Die Überprüfung findet durch ein einfaches ”‘in‘” und ”‘not_in‘” statt.

```
1 ((a1,a2,b1,b2) ‘in‘     signal ==> check(a1,a2,b1,b2) = 1) &
2 ((a1,a2,b1,b2) ‘not_in‘ signal ==> check(a1,a2,b1,b2) = 0) &
3
4 (signal = [(1,2,2,1),(2,1,1,2),(2,3,3,2),(2,4,4,2),(3,2,2,3),
5            (4,3,3,4),(4,2,2,4),(5,4,4,5),(4,5,5,4),(2,3,4,2),
6            (2,4,3,2),(3,2,2,4),(4,2,2,3)]) &
```

### 8.2.5 fahrschleife

Dafür, dass Bewegung ins Spiel kommt, sorgt die ”fahrschleife”, sie läßt die drei H-Bahnen immer wieder fahren, bis die Listen leer sind.

```
1 (fahrschleife([[],[],[]])) &
2 (fahrschleife(bahnen) <=== fahrschleife(faehrt(bahnen))) &
```

## 8.3 Beweis

Der Beweis, der in diesem Beispiel möglich ist, zeigt, dass die H-Bahnen ihre Streckenliste beliebig oft abfahren können, wenn dieses einmal funktioniert. Der Beweis wird im *Expander2* per Fixpunkt-Induktion gelöst. Der Beweisansatz sieht wie folgt aus:

```
1 All l1 l2 l3: (hbahn(l1,l2,l3) ==> fahrschleife([l1,l2,l3]))
```

Es lässt sich so beweisen, dass die Aussage gültig ist, also die Bahnen beliebig oft ihre Strecken befahren können, ohne dass sie sich auf einem Abschnitt entgegenkommen.

Eine bebilderte Durchführung des Beweises kann man den Vortragsfolien entnehmen.

# Teil IV

# Schlusswort

# Kapitel 9

# Zukünftige Arbeit

## 9.1 Fehlende Features

Durch den Compiler und *Expander2* in der jetzigen Form können nicht alle ausgewähltenb Merkmale korrekt transformiert oder verarbeitet werden. Hier nun eine Auflistung der fehlenden bzw. nicht voll funktionsfähigen Features:

- wird auf eine Objektinstanz mehrfach referenziert, so wird dies derzeit leider so umgesetzt, dass jede Referenz einem seperaten Objekt entspricht. Ändert man also den Wert eines Objektes so ist diese Veränderung nur lokal in der aufrufenden Referenz vorhanden

- wir haben nicht herausgefunden, wie man innerhalb der OCL-Constraints ein neues Objekt erzeugt; dies schränkte unsere Beispiele ziemlich ein oder erforderten ein Manipulieren von Hand

- Collection-Funktionen, die einen komplexeren nicht eindeutigen Aufruf besitzen (z.B. `iterate`) werden nicht immer korrekt transformiert

## 9.2 Ausblick

Die PG-Arbeit stellt (hoffentlich) eine starke Basis für weiter Projekte zur Verfügung. Primär sollten v.a. die vorher erwähnten Schwächen behoben werden, um eine komplette Repräsentation zu erhalten. Außerdem hat sich die PG auf ein UML-Element, Klassendiagramme, beschränkt. UML bietet jedoch noch einige andere Diagramme, wie z.B. Sequenzdiagramme oder Kollaborationsdiagramme, die vielleicht eine gute Ergänzung oder auch Alternative zu dem bisher gewählten OCL-Constraints ist.

Desweiteren hat sich die PG komplett auf bereits existierende Beweistechniken gestützt. Da diese jedoch eher für funktionale und logische Programme konzipiert wurden, werden selbst einfache Beweise im objektorientierten Bereich sehr groß und daher nicht sehr effizient. Es wäre also naheliegend nicht nur die Art der Spezifikation zu verändern

oder zu erweitern, sondern auch speziell objektorientierte Beweistechniken zu entwickeln.

# Kapitel 10

# Erfahrungen während der PG-Zeit

Im Laufe der Projektgruppe haben wir uns mit der Arbeitsweise des *Expander2* auseinandergesetzt. Es sind immer wieder Fragen und Probleme aufgetaucht, die wir mit Hilfe der Betreuer oder auch alleine geklärt haben. Gerade zu Beginn des Semesters, als kaum jemand die Arbeitsweise des Expanders wirklich kannte, gab es immer wieder Probleme. Es kostete sehr viel Zeit unsere Beispiele zu axiomatisieren und sie so umzusetzten, dass sie vom *Expander2* akzeptiert wurden. Mit dem Programm Together wurden UML Diagramme erstellt. Diese Diagramme dienten einmal der visuellen Darstellung der Beispiele aber hauptsächlich als Grundlage für die Compilereingabe, da Together den Export von UML-XMI-Darstellungen erlaubt.

Zu Beginn der Projektgruppe haben alle an den Beispielen mitgearbeitet. Ab dem zweiten Semester erfolgte nun eine Einteilung in UML-Parser und Compiler (Cordes, Daniluk, Eisenberg), OCL-Parser und Compiler (Rädisch), Beispielumsetzung (Elmrabti, Kacrmarczyk, Porsche, Samet, Sghouri) und Beweisführung (Steinfort). Da es innerhalb der Beispielgruppe zu Problemen sowohl thematischer als auch organisatorischer[1] Art kam, waren auch die für die Compiler Verantwortlichen immer wieder an Beispielen und Beweisen tätig.

Innerhalb der PG gab es auch einige teils schwerwiegende Probleme. So hatten alle Teilnehmer zu Beginn einen vollkommen unterschiedlichen Wissensstand und nur wenige hatten gleich von Beginn an den kompletten Überblick. Als problematischer erwies sich, dass bei einigen Teilnehmern nicht erkennbar war, dass sie dazu bereit waren das fehlende Wissen und Verständnis (z.B. für Prädikatenlogik, *Expander2* , Haskell oder Compilertechniken) aufzuarbeiten. Erschwerend kam hinzu, dass diese dann auch noch durch häufiges Fehlen bzw. Mini-Besuche auffielen. Da diese also zunehmend weniger zur PG-Arbeit beitrugen, kam es immer mehr zu Konflikten zwischen der PG und einigen Teilnehmern, die zum Ende der PG stetig zunahmen.

Interessanterweise muss man also sagen, dass nicht das Thema der Projektgruppe für die größten Probleme innerhalb der PG verantwortlich war, sondern die Einstellung einiger Teilnehmer.

[1] z.B. häufiges Fehlen oder mangelnde Vorbereitung

# Teil V

# Anhang

# Anhang A

# OCL-Grammatik

Im folgenden nun die konkrete Syntax für OCL, wobei dies die Darstellung von *Nichtterminalen* und dies die Darstellung von **Terminalen** ist.

| | | | |
|---|---|---|---|
| 1 | *oclFile* | → | (**package** *packageName* *oclExpressions* **endpackage**)+ |
| 2 | *packageName* | → | *pathName* |
| 3 | *oclExpressions* | → | (*constraint*)* |
| 4 | *constraint* | → | *contextDeclaration* (*constraintLoop*)+ |
| 5 | *constraintLoop* | → | **def** (name)? **:** (*letExpression*)* |
| 6 | | → | *stereotype* (name)? **:** *oclExpression* |
| 7 | *contextDeclaration* | → | **context** *operationContext* |
| 8 | | → | **context** *classifierContext* |
| 9 | *classifierContext* | → | *name* **:** *name* |
| 10 | | → | *name* |
| 11 | *operationContext* | → | *name* **::** *operationName* **(** *formalParameterList* **)** (**:** *returnType*)? |
| 12 | *stereotype* | → | **pre** |
| 13 | | → | **post** |
| 14 | | → | **inv** |
| 15 | *operationName* | → | *name* |
| 16 | | → | **=** |
| 17 | | → | **+** |
| 18 | | → | **-** |
| 19 | | → | **<** |
| 20 | | → | **<=** |
| 21 | | → | **>=** |
| 22 | | → | **>** |
| 23 | | → | **/** |
| 24 | | → | ** |
| 25 | | → | **<>** |

| | | | |
|---|---|---|---|
| 26 | | → | **implies** |
| 27 | | → | **not** |
| 28 | | → | **or** |
| 29 | | → | **xor** |
| 30 | | → | **and** |
| 31 | *formalParameterList* | → | *name* **:** *typeSpecifier* (**,** *name* **:** *typeSpecifier*)* |
| 32 | *typeSpecifier* | → | *simpleTypeSpecifier* |
| 33 | | → | *collectionType* |
| 34 | *collectionType* | → | *collectionKind* **(** *simpleTypeSpecifier* **=** |
| 35 | *oclExpression* | → | ((*letExpression*)* **in**)? *expression* |
| 36 | *returnType* | → | *typeSpecifier* |
| 37 | *expression* | → | *logicalExpression* |
| 38 | *letExpression* | → | **let** *name* ((**(** *formalParameterList* **)**)?<br>(**:** *typeSpecifier*)? **=** *expression* |
| 39 | *ifExpression* | → | **if** *expression* **then** *expression*<br>**else** *expression* **endif** |
| 40 | *logicalExpression* | → | *relationalExpression*<br>(*logicalOperator additiveExpression*)* |
| 41 | *relationalExpression* | → | *additiveExpression*<br>(*relationalOperator additiveExpression*)? |
| 42 | *additiveExpression* | → | *multiplicativeExpression*<br>(*addOperator multiplicativeExpression*)* |
| 43 | *multiplicativeExpression* | → | *unaryExpression*<br>(*multiplyOperator unaryExpression*)* |
| 44 | *unaryExpression* | → | (*unaryOperator*)? *postfixExpression* |
| 45 | *postfixExpression* | → | *primaryExpression* (*primaryExpressionNavig*)* |
| 46 | *primaryExpressionNavig* | → | **.** *propertyCall* |
| 47 | | → | **->** *propertyCall* |
| 48 | *primaryExpression* | → | *literalCollection* |
| 49 | | → | *literal* |
| 50 | | → | *propertyCall* |
| 51 | | → | **(***expression* **)** |
| 52 | | → | *ifExpression* |
| 53 | *propertyCallParameters* | → | **(** (*declarator*)? (*actualParameterList*)? **)** |
| 54 | *literal* | → | *string* |
| 55 | | → | *number* |
| 56 | | → | *enumLiteral* |
| 57 | *enumLiteral* | → | *name* (**::** *name*)+ |
| 58 | *simpleTypeSpecifier* | → | *pathName* |
| 59 | *literalCollection* | → | *collectionKind*<br>**{** ( *collectionItem* ( **,** *collectionItem*)*)? **}** |

| | | | |
|---|---|---|---|
| 60 | *collectionItem* | → | *expression* (.. *expression*)? |
| 61 | *propertyCall* | → | *pathName* (*timeExpression*)? (*qualifiers*)? (*propertyCallParameters*)? |
| 62 | *qualifiers* | → | [ *actualParameterList* ] |
| 63 | *declarator* | → | *name* (, *name*)* (: *simpleTypeSpecifier*)? (; *name* : *typeSpecifier* = *expression*)? \| |
| 64 | *pathname* | → | *name* (:: *name*)* |
| 65 | *timeExpression* | → | **@pre** |
| 66 | *actualParameterList* | → | *expression* (, *expression*)* |
| 67 | *logicalOperator* | → | **and** |
| 68 | | → | **or** |
| 69 | | → | **xor** |
| 70 | | → | **implies** |
| 71 | *collectionKind* | → | **Set** |
| 72 | | → | **Bag** |
| 73 | | → | **Sequence** |
| 74 | | → | **Collection** |
| 75 | *relationalOperator* | → | = |
| 76 | | → | > |
| 77 | | → | < |
| 78 | | → | >= |
| 79 | | → | <= |
| 80 | | → | <> |
| 81 | *addOperator* | → | + |
| 82 | | → | - |
| 83 | *multiplyOperator* | → | * |
| 84 | | → | / |
| 85 | *unaryOperator* | → | - |
| 86 | | → | **not** |
| 87 | *name* | → | [, A-Z, _] ([a-z, A-Z, 0-9, _])* |
| 88 | *number* | → | ([0-9])+ (. ([0-9])+)? (exponent)? |
| 89 | *exponent* | → | (e\|E) ((+\|-))? ([0-9])+ |
| 90 | *string* | → | ’ ()* ’ |

Tabelle A.1: OCL-Grammatik

# Tabellenverzeichnis

4.1 Exemplarisches Transformationsschema für Collection-Funktionen . . . . . 52
A.1 OCL-Grammatik . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

# Abbildungsverzeichnis


2.1 Oberfläche des *Expander2* . . . . . . . . . . 11
2.2 Übersicht über Proof-Carrying Code (PCC) . . . . . . . . . . 15
2.3 Die Untermenge der DEC Alpha Assemblersprache . . . . . . . . . . 17
2.4 Die abstrakte Maschine . . . . . . . . . . 17
2.5 Der *Verification Condition* Generator . . . . . . . . . . 21
2.6 DEC Alpha Assemblerprogramm für Speicherzugriffe . . . . . . . . . . 22
2.7 Teil eines formalen *Safety Proof* von $SP_r$ . . . . . . . . . . 23

3.1 Parser- bzw. Compiler-Durchlauf . . . . . . . . . . 30
3.2 Beispielhaftes Klassendiagramm . . . . . . . . . . 32
3.3 Schema: UML-Compiler . . . . . . . . . . 45

4.1 Aufbau des OCL-Compilers . . . . . . . . . . 48
4.2 Schema: OCL-Pimärparser . . . . . . . . . . 53
4.3 Schema: OCL-Sekundärparser . . . . . . . . . . 62
4.4 Schema: OCL-Compiler . . . . . . . . . . 65

5.1 UML Diagramm, der ersten ”Auto-Version” . . . . . . . . . . 79

6.1 UML Diagramm des Bankbeispiels . . . . . . . . . . 87

7.1 UML Diagramm - Mobile Phone System (leider nicht aktuell) . . . . . . . . . . 89
7.2 Parallelität im Mobile Phone System . . . . . . . . . . 91

# Literaturverzeichnis


[1] *A Formal Methods Presenter and Animator*; Padawitz, Peter; ls5-www.cs.uni-dortmund. de/~peter/Expander2/Expander2.html

[2] *A Gentle Introduction to Haskell 98*; Hudak, P.; Peterson, J.; www.haskell.org/tutorial

[3] *Algebraische Spezifikation und Verifikation; Seminarausarbeitung*; Daniluk, Damian; 2003; http://ls5-www.cs.uni-dortmund.de/~peter/PG437/DamianVortrag.pdf

[4] *A Survey of O'Haskell*; Jones, M.P.; Nordlander, J.; Sydow, B. v.; Carlson; www.cs.chalmers.se/~nordland/ohaskell/survey.html

[5] *A Tutorial on (Co)Algebras and (Co)Induction*; B. Jacobs, J. Rutten; EATCS Bulletin 62 (1997) 222-259

[6] *Expander2: Beweisfeatures*; Rädisch, Tim; Steinford, Sebastian; 2003; http://ls5-www.cs.uni-dortmund.de/~peter/PG437/Raedisch.pdf

[7] *Expander2 - Manual*; Padawitz, Peter; http://ls5-www.cs.uni-dortmund.de/~peter/Expander2/Manual.html

[8] *Formale Methoden des Systementwurfs*; Padawitz, Peter; ls5-www.cs.uni-dortmund.de/~peter/TdP96.ps.gz

[9] *Haskell-Einführung an der Uni Freiburg*; http://www.informatik.unifreiburg.de/~kelle/kurzeinf.html

[10] *Introduction to Functional Programming using Haskell*; R. Bird; Prentice Hall 1998

[11] *Logic in Computer Science: Modeling and Reasoning about Systems*; Huth, Ryan Kapitel 4

[12] *Mathematischstrukturelle Grundlagen der Informatik*; Ehrig, Mahr, Cornelius, Große, Rhode, Zeitz; Springer 1999; 3-540-41923-3

[13] *Needed Narrowing; Berechnungsmodell von Curry*; Frank, Sephan; http://uebb.cs.tu-berlin.de/csem00/vortraege/sfrank.ps

[14] *Object Modelling with the OCL*; Clark, T.; Warmer, J.B.; Springer LNCS 2263, 2002

[15] *Object oriented specification case studies*; Lano, K.; 1994; Prentice Hall; 0-13-097015-8

[16] *Objektorientierte Softwareentwicklung*; Östereicher, Bernd; Oldenbourg 2003

[17] *OMG-XML Metadata Interchange (XMI) Specificatio, v1.2*; OMG; http://www.omg.org/cgi-bin/doc?formal/2002-01-01

[18] *Proof-Carrying Code*; Necula, G.C.; Lee, P.; Report CMU-CS-96-165, Carnegie Mellon University 1996

[19] *Proof-Carrying Code; Seminarausarbeitung*; Cordes, Sven; Eisenberg, Dominik; 2003; http://ls5-www.cs.uni-dortmund.de/~peter/PG437/PCCVortrag.pdf

[20] *Proof for Functional Programming*; Thompson, Simon; in: ”Research Directions in Parallel Functional Programming”, Springer 1999, Kapitel 2 u. Kapitel 4

[21] *Proofs and Types*; Girard, J.-Y.; Lafont, Y; Taylor, P; Cambridge University Press 1990

[22] *Proofs are Programs: 19th Century Logic and 21st Century Computing*; Wadler, Ph.; Report, Avaya Labs 2000

[23] *Das verbesserte System der Illuminaten mit allen seinen Einrichtungen und Graden*; Weishaupt, Adam; Gotha 1787

[24] *Recursion and Induction*; Dunne, Peter; http://osiris.sunderland.ac.uk/~cs0pdu/pub/mat118/haskell/ ↦ recurind/recurind.html

[25] *Reduktionssysteme*; Avenhaus, Jürgen; Springer 1995; 3-540-58599-1

[26] *Sitzungsprotokolle*; PG 437

[27] *Structured Swinging Types*; Padawitz, Peter; http://ls5-www.cs.uni-dortmund.de/~peter/SST.ps.gz

[28] *Swinging Types*; Padawitz, Peter; Theoretical Computer Science 243 (2000) 93-165; ls5-www.cs.unidortmund.de/~peter/Swinging.html

[29] *Swinging Types at Work*; Padawitz, Peter; ls5-www.cs.uni-dortmund.de/~peter/BehExa.ps.gz

[30] *Theorie der Programmierung*; Steffen, Bernhard; http://ls5-www.cs.uni-dortmund.de/teaching/ss2003/skript-tdp-ss03.pdf.gz

[31] *The Object Constraint Language*; Warmer, J.B.; Kleppe, A.G.; Addison-Wesley 1999

[32] *The next 700 theorem proovers*; Odifreddi, P; in "Logic and Computer Science", Academic Press (1990) 361-386

[33] *The Unified Modeling Language Reference Manual*; Rumbaugh, J; Jacobson, I; Booch, G; Addison-Wesley 1999

[34] *Towards a Coalgebraic Semantics of UML: Class Diagrams and Use Cases*; Meng; Aichernig, B.K.

[35] *Towards a Workbench for Interactive Formal Reasoning*; Padawitz, Peter; ls5-www.cs.uni-dortmund.de/~peter/Expander2/Chiemsee.ps.gz

[36] *UML@work: Von der Analyse zur Realisierung*; Hitz; Kappel; dpunkt-Verlag 2003, 477-563

[37] *UML konzentriert*; Fowler, Martin; Scott, Kendall

[38] *UML und OCL; Seminarausarbeitung*; Elmrabti, Ashour; 2003; http://ls5-www.cs.uni-dortmund.de/~peter/PG437/UMLVortrag.pdf

[39] *Unified Modeling Language, v1.5*; OMG; http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf

[40] *An object-Z specification of a mobile phone system*; Gordon Rose, Roger Duke; Object-oriented specification case studies (1994) 110-129; ISBN:0130970158