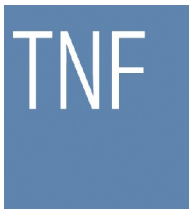
JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU



Technisch-Naturwissenschaftliche
Fakultät

OCL-Parser und -Editor für dynamische Metamodelle

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Philipp Mitterer, 0855300

Angefertigt am:
Institut für Systems Engineering and Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed M. Sc.

Linz, Februar 2012

Kurzfassung

Diese Bachelorarbeit beschäftigt sich mit der Umsetzung eines OCL-Parsers mit dazugehörigem Editor für dynamische Metamodelle. Dies bedeutet, dass sich Typen, Operationen und Attribute des Metamodells zur Laufzeit verändern können und nicht statisch festgelegt sein müssen.

Die Anbindung des Metamodells an den Parser erfolgt über eine einfache Schnittstelle, wodurch beliebige Metamodelle schnell für den Parser adaptiert werden können. Der Parser selbst unterstützt einen Großteil der offiziellen OCL-Spezifikation Version 2.2. Zusätzlich wurde der Parser auf Korrektheit, Performanz und Skalierbarkeit validiert. Dabei stellte sich heraus, dass der Parser sehr performant und skalierbar ist.

Der erstellte Editor unterstützt den Entwickler mit Syntax-Highlighting und Code-Completion. Das Syntax-Highlighting formatiert unterschiedliche Symbole im Quellcode unterschiedlich und bietet dem Entwickler damit mehr Überblick. Die Code-Completion hilft dem Entwickler bei der Eingabe von Objektoperationen und -attributen, indem sie alle erlaubten Operationen und Attribute als eine Liste von Vorschlägen präsentiert.

Abstract

you can even say that the meta model may even change over time

why majority?

besser: provides an alternative implementation

This bachelor thesis deals with the implementation of a OCL parser and corresponding editor for dynamic meta-models. In a dynamic meta-model types operations and attributes might change during runtime and are not specified statically.

The connection between parser and meta-modell is established via a simple interface. Therefore different meta-models can be adapted for the parser easily. The parser supports the majority of features stated in the official OCL specification version 2.2. Furthermore the parser was evaluated for correctness, performance and scalability. The evaluation showed that the parser has a high performance and scalability.

The editor suppports developers with syntax highlighting and code completion. Syntax highlighting changes the appearance of different symbols in source codes and therefore provides a better overview. Code completion aids developers with the insertion of object operations and attributes by providing a list of all valid operations and attributes for this object.

perhaps defined first what a meta model is and then why why meta model are dynamic and what dynamism in meta models means

Inhaltsverzeichnis

Kapitel 1

Motivation und Problemstellung

Modellgetriebene Softwareentwicklung (Model-Driven Engineering) ist ein moderner Ansatz des Software Engineering. Dabei werden Modelle, der zu erstellenden Applikation, anstatt von Programmcode entwickelt [1, S. 138 f.]. Um Fehler zu vermeiden werden diese Modelle von Tools (z.B. *Model/Analyzer* [2]) mittels definierter Regeln auf Konsistenz überprüft. Zum Definieren derartiger Regeln ist OCL (Object Constraint Language) sehr gut geeignet und wird deshalb von diesen Tools auch häufig unterstützt.

Was passiert nun, wenn sich das Metamodell ändert? Mit dieser Fragestellung haben sich Demuth et al. [3] auseinandergesetzt und XLM zur Co-Evolution von Modell und Metamodell entwickelt. Ein Problem ist jedoch, dass der Parser, der die OCL-Constraints in überprüfbare Konsistenzregeln umwandelt, auf kein statisch definiertes Metamodell zurückgreifen. Ein geeigneter Parser müsste mit einem sich dynamisch zur Laufzeit verändernden Metamodell umgehen können.

besser: Wilms [???] referenz auf seine thesis to appear (frag in nach einem titel?)

Ein ähnliches Problem hat Sebastian Wilms bei seiner Arbeit. Seine Konsistenzregeln beziehen sich auf kein explizites Modell, sondern auf Java-Klassen zur Laufzeit. Da Java zur Laufzeit Klassen nachladen kann, ist auch hier das Metamodell nicht fix vorgegeben, sondern ändert sich dynamisch. Ein OCL-Parser für diese Konsistenzregeln müsste ein derartiges dynamisches Metamodell handhaben können.

Ein fehlendes Feature in OCL ist die Möglichkeit, sich selbst Funktionen zu definieren und diese in anderen OCL-Constraints oder -Funktionen zu referenzieren bzw. aufzurufen. Ein derartiges Feature würde es ermöglichen, OCL-Constraints zu modularisieren und nach dem DRY-Prinzip Redundanz zu vermeiden.

Geht man einen Schritt weiter, könnte man so ermöglichen, rekursive Funktionen zu definieren. Zwar ist nach der Church-Turin-These eine Umwandlung einer rekursiven in eine iterative Funktion immer möglich, jedoch

ist eine rekursiv geschriebene Funktion oft leichter zu definieren und einfacher zu verstehen. Darüber hinaus ist es speziell in OCL schwierig, solch eine Umwandlung durchzuführen, da dafür meist der Zustand (der bei rekursiven Funktionen implizit gespeichert wird) explizit gespeichert werden muss.

Die Definition neuer Funktionen führt zu einer Änderung des OCL-Metamodells zur Laufzeit. Ein OCL-Parser, der dieses Feature unterstützt, muss in der Lage sein, sich auf ein dynamisches Metamodell einzustellen.

Kapitel 2

Ziel und Anforderungen

Das Hauptziel dieser Arbeit ist die Umsetzung eines OCL-Parsers. Darüber hinaus soll ein Editor für OCL geschaffen werden, der den Entwickler mit Syntax-Highlighting und Code-Completion für OCL unterstützt. Deshalb werden die meisten Kapitel dieser Arbeit in die drei Teile "Parser", "Code-Completion" und "Syntax-Highlighting" aufgeteilt.

2.1 Parser

Der Parser ist der zentrale und größte Teil dieser Arbeit. Der Parser soll in der Lage sein OCL-Ausdrücke in einen abstrakten Syntaxbaum (Abstract Syntax Tree - AST) der ARL (Abstract Rule Language) zu übersetzen, auf welchem Konsistenzüberprüfungen durchgeführt werden können. Dabei soll der Parser fehlerhafte Ausdrücke abweisen und den Fehler dem System melden. Fehlerhafte Ausdrücke sind jene, die entweder keine syntaktisch korrekten OCL-Ausdrücke sind oder semantisch nicht erlaubt sind, da sie beispielsweise Typverletzungen enthalten.

Der Parser soll über eine möglichst einfache und vor allem schmale API verschiedenste Metamodelle unterstützen, damit das Anbinden neuer Metamodelle einen geringen Aufwand bereitet. Die Metamodelle können sich zur Laufzeit ändern und der Parser muss sich dieser Änderung anpassen. Das bedeutet vor allem, dass sich Typen und ihre Operationen und Properties ändern können. Dadurch sind beim Definieren oder Ändern von OCL-Constraints neue Operationen möglich und manche bestehenden Operationen nicht mehr erlaubt.

Bei der Umsetzung des Parser soll darauf geachtet werden, dass sich der Parser möglichst an die offizielle OCL-Spezifikation [4] hält. Dabei kann allerdings nicht die komplette Spezifikation umgesetzt werden, da sich diese auf UML bezieht und nicht in allen Metamodellen umgesetzt werden kann. Deshalb wird nur eine Teilmenge der Features implementiert. Bei dieser Arbeit wurde die Version 2.2 der OCL-Spezifikation verwendet.

ich nehme an, dies ist die antwort auf meine frage im abstract. wäre gut wenn du dies auch im abstract so rüberbringst

2.2 Code-Completion

Code-Completion ist ein Feature, dass von vielen Entwicklungsumgebungen bereits unterstützt wird. Es unterstützt den Entwickler dahingehend, dass es ihm während der Eingabe des Quellcodes Vorschläge unterbreitet, welche Wörter er einsetzen könnte. Dabei werden vor allem die Operationen und Properties vorgeschlagen, die auf ein Objekt angewandt werden können bzw. die ein Objekt besitzt. Das erhöht zum einen die Entwicklungsgeschwindigkeit, da nicht mehr das komplette Wort eingegeben werden muss. Zum anderen hilft es dem Entwickler, da dieser sich nicht merken oder erst anderenorts nachsehen muss, welche Operationen auf einem Objekt erlaubt sind.

Der OCL-Editor soll diese Art von Code-Completion unterstützen. Dabei ist es auch hier wieder essenziell, dass die Code-Completion mit einem dynamischen Metamodell arbeiten kann, weil sich auch hier während der Laufzeit die Vorschläge ändern können.

2.3 Syntax-Highlighting

Das Syntax-Highlighting ist ein weiteres Feature, welches dem Entwickler die Arbeit erleichtert. Dabei werden unterschiedliche Wörter und Zeichen je nach ihrer syntaktischen Bedeutung im Editor unterschiedlich (z.B. andere Farbe, fett oder kursiv) dargestellt. Somit ist es dem Entwickler leichter möglich, den Quellcode zu lesen und zu analysieren. Der Editor soll Syntax-Highlighting für OCL unterstützen.

Kapitel 3

Theoretische Grundlagen

Dieses Kapitel beschreibt ein paar theoretische Grundlagen, die für das Verständnis dieser Arbeit relevant sind.

3.1 Object Constraint Language - OCL

Die Object Constraint Language (OCL) ist eine reine Spezifikationssprache. Sie dient der Spezifikation von Regeln, den sogenannten Constraints, auf Objekte und ist dabei frei von Seiteneffekten. [4, S. 5]
OCL wurde ursprünglich für UML entwickelt. Eine Teilmenge der Spezifikation unterstützt mittlerweile auch MOF (MetaObject Facility), jedoch kann der gesamte Umfang von OCL nur mit UML genutzt werden. [4, S. 1]

Die Punkte dieses Kapitels beziehen sich auf Elemente von OCL, wie sie in der Spezifikation von OCL [4] geschildert sind. Es soll dabei keine komplette Auflistung der Möglichkeiten von OCL, sondern vielmehr ein Überblick über die wichtigsten Merkmale bereitet werden.

3.1.1 Constraints

Constraints sind die oben bereits beschriebenen Regeln auf Objekte von Modellen. Es gibt diverse Arten von Constraints (u.A. Invarianten, Preconditions, Postconditions). Allerdings werden in dieser Arbeit nur Invarianten behandelt. Jede Constraint besitzt einen Context in welchem die Constraint validiert wird. Der Context eines Constraints ist der Typ auf den die Regel angewandt werden soll. Bei der Validierung wird eine Constraint mit allen Instanzen des Constexts überprüft.

Auf Operationen oder Properties eines Objekts kann wie bei C# oder Java mittels Punkt-Operator zugegriffen werden:

```
person.name = "John"
person.isAdult()
```

Das Context-Objekt kann in einer Constraint mit dem Schlüsselwort *self* angesprochen werden. Ist der Context eindeutig, kann *self* auch weggelassen werden:

```
self.name = "John"
name = "John"
```

Wie in vielen anderen Sprachen gibt es auch in OCL eine If-Anweiung. Die Angabe eines Else-Pfades ist bei OCL zwingend erforderlich.

```
if self.age >= 18 then
  self.isAdult()
else
  not self.isAdult()
endif
```

3.1.2 Datentypen

Neben den Typen des Modells besitzt OCL eine Reihe von eigenen Typen, die für Constraints verwendet werden können.

Basisdatentypen

OCL definiert drei spezielle Basisdatentypen:

- **OclAny** ist der Supertyp aller anderen Typen (auch der Modeltypen). D. h. Jedes Objekt kann wie ein OclAny-Objekt verwendet werden und erbt dementsprechend alle OclAny-Operationen (u.A. `oclAsType()`, `oclIsKindOf()`, `<>`, `oclIsNew()`)
- **OclVoid** ist der Datentyp des Literals `null`. Ein Objekt dieses Datentyps kann jedem anderen Datentyp außer *OclInvalid* zugewiesen werden.
- **OclInvalid** ist der Datentyp des Literals `invalid`. Ein Objekt dieses Datentyps kann jedem anderen Datentyp außer *OclVoid* zugewiesen werden.

Primitive Datentypen

Die primitiven Datentypen von OCL sind spezielle Datentypen, deren Typen mittels Literale erzeugt werden können. Neben einer Reihe von unären und binären Operationen (`+`, `-`, `or`, etc.), die in Infix-Notation geschrieben werden können, können wie bei allen anderen Datentypen Operationen auf den Objekten ausgeführt werden (z.B: `5.abs()`). Die folgenden vier primitiven Datentypen definiert OCL:

Integer ist der Ganzzahlen-Datentyp von OCL. Literale werden einfach als Zahlen mit optionalem Vorzeichen geschrieben (z.B. `-538`). Auf diesen Datentyp sind zahlreiche arithmetische Operationen möglich.

Real ist der Datentyp der reellen Zahlen. Literale werden mit einem Punkt als Kommazeichen und optionaler Potenz-Angabe geschrieben (z.B. `4.5`, `5.3e-3`).

Boolean ist der Boolesche Datentyp von OCL. Für diesen Datentyp gibt es nur die beiden Literale `true` und `false`. Auf `Boolean`-Objekte sind zahlreiche logische Operationen möglich (`and`, `or`, `xor`, etc.).

String ist der Zeichenketten-Datentyp. String-Literale werden zwischen zwei Anführungszeichen oder Apostrophen gesetzt (`"Hallo"`, `'Welt'`)

Collections

Zur Abbildung von Mengen und Listen unterstützt OCL fünf verschiedene Collection-Typen. **Collection** ist der Supertyp aller Collectiontypen. **Set** repräsentiert eine mathematische Menge, d.h. es sind keine Duplikate in der Collection. Im Gegensatz dazu können in einer **Bag** Duplikate vorkommen. **OrderedSet** und **Sequence** bringen zusätzlich eine Reihenfolge in die Elemente. Ansonsten entsprechen sie Set bzw. Bag.

Collection-Literale werden mit dem Namen des Collection-Typs und den initial enthaltenen Elementen geschrieben.

```
Sequence{2, 3, 5, 7, 11}
```

Neben den normalen OclAny-Operationen, die auf jedes Objekt ausgeführt werden kann, kann auf Collections eine Reihe von Collection-Operationen und Iterator-Operationen ausgeführt werden. Diese werden im Gegensatz zu normalen Operationen mit einem Pfeil-Operator aufgerufen.

```
Bag{4, 5}->includes(5)
```

Operationen auf Collections unterteilen sich in die normalen Collection-Operationen (*size()*, *isEmpty()*, *includes()*) und den Iterator-Operationen. Iterator-Operationen sind Operationen, die über die alle Elemente der Collection iterieren. Dafür kann optional bei Iterator-Operationen eine Iterator-Variable definiert werden, die das aktuelle Element repräsentiert.

```
-- alle Kinder müssen jünger sein als das Elternteil
self.childs->forall(child : Person | child.age < self.age)
-- keine zwei Kinder einer Person dürfen den gleichen Namen haben
self.childs->isUnique(name)
```

Die Iterate-Operation ist eine spezielle sehr mächtige Iterator-Operation. Zusätzlich zu einer Iterator-Variable besitzt sie auch eine Akkumulator-Variable, in der nach jeder Iteration das Ergebnis des Operationskörpers gespeichert wird. Das Gesamtergebnis der Iterate-Operation ist der Wert der Akkumulator-Variable nach allen Iterationen. Mit dieser Operation lassen sich alle anderen Iterator-Operationen nachbilden.

```
-- alle Kinder müssen jünger sein als das Elternteil
self.childs->iterate(child : Person ; result : Boolean = true |
  result and (child.age < self.age))
```

Tuples

Tuples erlauben es einem, in OCL neue Datentypen zu definieren. Ein Tuple besteht aus einer Reihe von benannten Elementen, auf die wie Properties zugegriffen wird.

```
Tuple{ x : Integer = 5, y : String}.x = 5
```

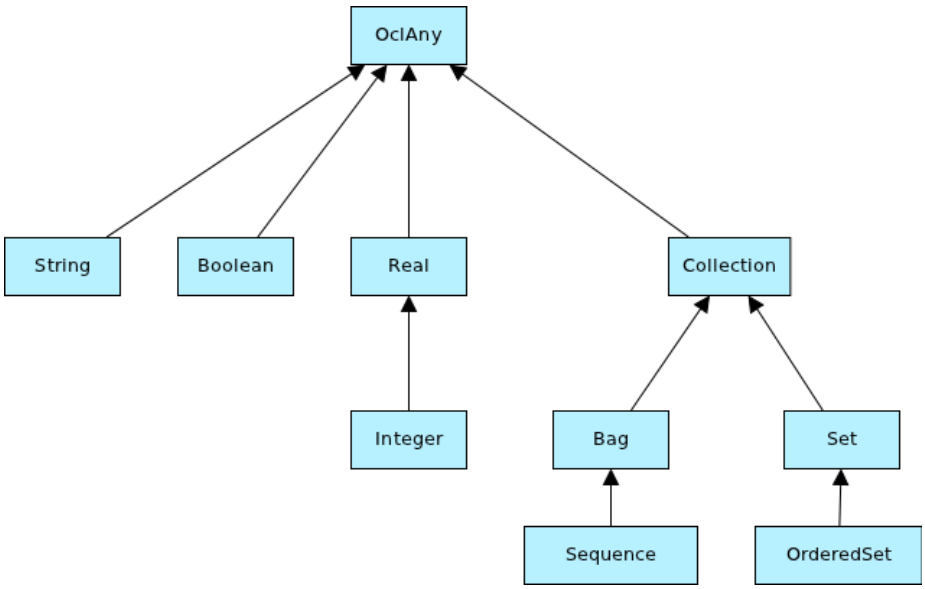


Abbildung 3.1: Zuweisungkompatibilität der OCL-Typen

Zuweisungkompatibilität

Die Zuweisungskompatibilität der OCL-Datentypen ist in Abbildung 3.1 dargestellt. Zusätzlich gelten folgende Regeln:

- Eine Collection ist zu einer anderen Collection zuweisungskompatibel, wenn der Collection-Typ und der Element-Typ zuweisungskompatibel ist.
- *OclVoid* und *OclInvalid* sind zu allen Typen zuweisungskompatibel, außer zu sich gegenseitig.

3.2 Abstract Rule Language

3.3 Compiler

Ein Compiler übersetzt den Programmcode in eine für den Rechner verständliche und ausführbare Form, meist Maschinencode. Dies passiert in mehreren Stufen, die bei heutigen Compilern meist verzahnt ausgeführt werden. [5, S. 6]

1. lexikalische Analyse
2. syntaktische Analyse
3. Typprüfung
4. Code-Erzeugung

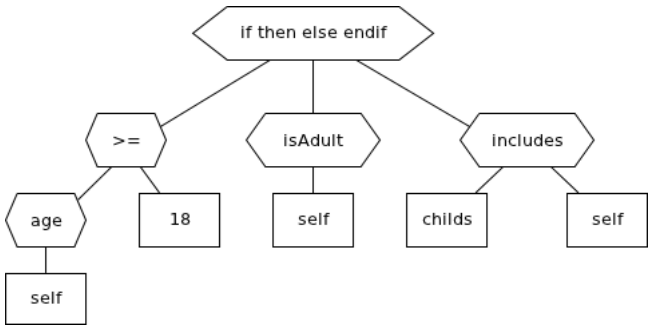


Abbildung 3.2: AST des Ausdrucks `if self.age >= 18 then self.isAdult() else childs.inludes(self) endif`

3.3.1 Lexikalische Analyse

Ein lexikalischer Scanner (oder auch Lexer genannt) spaltet im Zuge der lexikalischen Analyse die Zeichen und Wörter des Inputstroms in Symbole (den sogenannten Tokens) auf, die dem Alphabet der Sprache entsprechen. Daraus entsteht eine Sequenz von Tokens, welche direkt als Input für die syntaktische Analyse dient. Dabei landen für das Programm unbedeutende Symbol wie Whitespaces und Kommentare nicht im Tokenstrom, sondern werden überlesen. [6, S. 84 f.]

3.3.2 Syntaktische Analyse

Ein sogenannter Parser übersetzt den Tokenstrom in eine Form, die der syntaktischen Struktur des Programms entspricht [5, S. 6]. Eine derartige Form ist der Abstract Syntax Tree. Während des Übersetzens überprüft der Parser außerdem die syntaktische Korrektheit des Programms, also ob die Struktur des Programms der Grammatik der Sprache entspricht [6, S. 160].

Parser lassen sich in 2 mögliche Kategorien einteilen: Top-Down-Parser und Bottom-Up-Parser [5, S. 18-24].

Abstract Syntax Tree

Der Abstract Syntax Tree (AST) ist eine Datenstruktur, die Programmstrukturen als Baum darstellt. Ein Operator wird als Knoten dargestellt und die Kinder dieses Knotens entsprechen den Parametern der Operation. Ein Beispiel für einen AST ist in Abbildung 3.2 dargestellt.

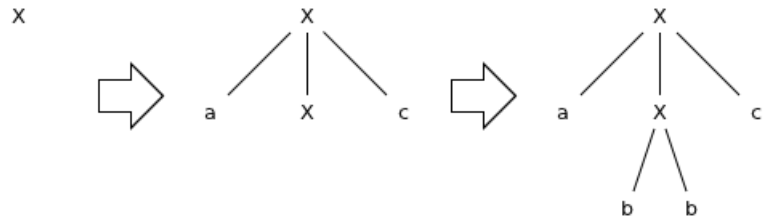


Abbildung 3.3: Beispiel für Top-Down-Parsing

Top-Down-Parser

Bei Top-Down-Parsern wird der AST von oben nach unten aufgebaut. Stehen mehrere Produktionen zur Auswahl muss deshalb im Vorhinein entschieden werden, welche Produktion die hier richtige ist. Dieses Prinzip wird im folgenden Beispiel erklärt.

```
X : a X c
  | b b
  ;
```

Für den Eingabestrom `abbc` wird am Anfang entschieden, ob `a X c` oder `b b` die richtige Produktion ist, wobei die Wahl offensichtlich auf die erste Variante fallen wird. Nach dem weiterlesen wird die Entscheidung erneut für `bbc` getroffen, wobei in diesem Fall die zweite Variante die richtige ist. Dieses Beispiel ist in Abbildung 3.3 veranschaulicht.

Bottom-Up-Parser

Das Bottom-Up-Parsing ist das Gegenteil vom Top-Down-Parsing. Hierbei entsteht der AST von unten nach oben. Bei diesen Parsern liegt die Entscheidung darin, ob sie die aktuellen Symbole zu einer Produktion reduzieren, oder noch weitergehen. Bottom-Up-Parsing am bereits bekannten Beispiel:

```
X : a X c
  | b b
  ;
```

Für den Eingabestrom `abbc` wird zuerst `a` in den AST aufgenommen. Da es keine Produktion für `a` gibt, wird auch noch `b` und ein weiteres `b` in den AST aufgenommen. Für `b b` gibt es eine Produktion und deshalb wird `b b` zu `X` reduziert. Dieses Beispiel ist in Abbildung 3.4 veranschaulicht.

3.3.3 Typprüfung

Bei typisierten Sprachen wird nach dem Parsen zusätzlich eine *Typprüfung* vollzogen. Diese stellt sicher, dass Werte Variablen oder Operationsparametern nur zugewiesen werden, sofern deren Typen zuweisungskompatibel sind.

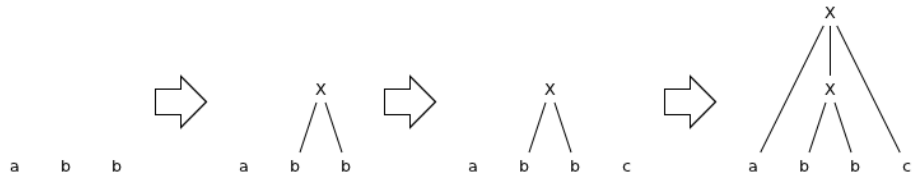


Abbildung 3.4: Beispiel für Bottom-Up-Parsing

3.3.4 Code-Erzeugung

In der Code-Erzeugungsphase wird der AST in ausführbaren Maschinencode übersetzt. Darauf wird hier jedoch nicht näher eingegangen, da es in dieser Arbeit zu keiner Code-Erzeugung kommt, sondern der AST bereits das Endprodukt ist. Dies ist auch der Grund, weshalb der vorgestellte OCL-Parser kein vollwertiger Compiler ist.

Es sei hier angemerkt, dass in dieser Arbeit der Begriff *Parser* für die Ausführung der lexikalische Analyse, der syntaktische Analyse und der Typprüfung steht, sofern dies nicht anders angegeben ist.

Kapitel 4

Related Work

In diesem Kapitel werden andere Projekte und Arbeiten analysiert, die ähnliche Problemstellungen und Anforderungen besitzen.

4.1 OCL-Parser

Es existieren derzeit einige OCL-Parser. Am bedeundensten sind wohl Eclipse OCL [7] und Dresden OCL [8]. Eclipse OCL unterstützt lediglich Modelle, die auf dem ebenfalls von Eclipse entwickelten EMF Ecore-Metamodell basieren. Dresden OCL unterstützt ebenfalls EMF Ecore-basierte Modelle. Darüber hinaus werden auch Modelle aus Java-Klassen und Modell aus XML-Dateien unterstützt. Allerdings handelt es sich bei all diesen Metamodellen um statisch definierte Modelle, die sich zur Laufzeit nicht ändern.

4.2 Code-Completion

Da dieses Thema in der Literatur bis jetzt nur sehr spärlich behandelt wurde, kann hier nur begrenzt auf diese zurückgegriffen werden. Dennoch unterstützen die meisten IDEs heutzutage den Entwickler mit Code-Completion.

Dabei macht es einen großen Unterschied, ob die behandelte Sprache typisiert ist oder nicht. Bei untypisierten Sprachen ist der Typ einer Variable oder Methode unbekannt und die Code-Completion kann deshalb nur alle möglichen Methodennamen anbieten. Auf diese Weise arbeitet auch Squeak [9]. RoelTyper [10] bietet hier eine mögliche Verbesserung an, indem mittels Typinferenz Typkandidaten für ein Objekt zu bestimmt werden und dadurch die Kandidaten für Code-Completion eingeschränkt werden können.

Bei typisierten Sprachen kann der Typ eines Objekts jedoch festgestellt werden. Dadurch ist die Code-Completion in der Lage, passende Vorschläge anzubieten. Das Problem hierbei ist herauszufinden, welchen Typ ein Code-Fragment besitzt. Eclipse [11] verfügt deshalb zu jeder Zeit über ein Modell des Programms [12]. Die Schwierigkeit dieses Ansatzes liegt darin, dass jede Codeänderung durch Benutzereingaben das Modell verändert und dieses dann aktualisiert werden muss.

Kapitel 5

Illustrative Beispiele

Zum leichteren Verständnis der Umsetzung der Arbeit und der Probleme, die bei dieser Umsetzung aufgetreten sind, werden in diesem Kapitel einige Beispiel-OCL-Ausdrücke beschrieben. Diese Beispiele dienen der Veranschaulichung der Umsetzung (siehe Kapitel 6).

Das Klassendiagramm des Modells unserer Beispiele wird in Abbildung 5.1 gezeigt. Das Beispiel modelliert ein kleines System, wie es an einer Universität verwendet werden könnte. Ein Student absolviert beliebig viele Kurse. Jeder Kurs hat eine bestimmte Anzahl an Credits und der Student erhält durch das Absolvieren von Übungen und Tests Punkte für diesen Kurs. Zusätzlich gibt es Übungskurse, die Punkte für Übungen extra auflisten. Darüber hinaus kann jeder Kurs eine Reihe von anderen Kursen als Voraussetzung haben, die der Student zuvor absolviert haben muss.

Die Datentypen, welche in dem Beispiel verwendet werden, heißen `ModelInteger` und `ModelString`, um sie von den OCL-Typen `Integer` und `String` zu unterscheiden.
Aus diesem Szenario lassen sich nun eine Reihe von OCL-Ausdrücken modellieren. Alle Ausdrücke sind im Kontext eines Studenten deklariert.

Das erste Beispiel könnte eine Voraussetzung für ein Stipendium zeigen. Die Summe der Credits aller Kurse, die ein Student absolviert hat, muss mindestens 30 sein.

```
self.courses->collect(c : Course | c.credits)->sum() >= 30
```

Diese Bedingung kann mittels `iterate`-Operation auch anders formuliert werden:

```
self.courses->iterate(c : Course; sum : Integer = 0 |
    sum+c.credits) >= 30
```

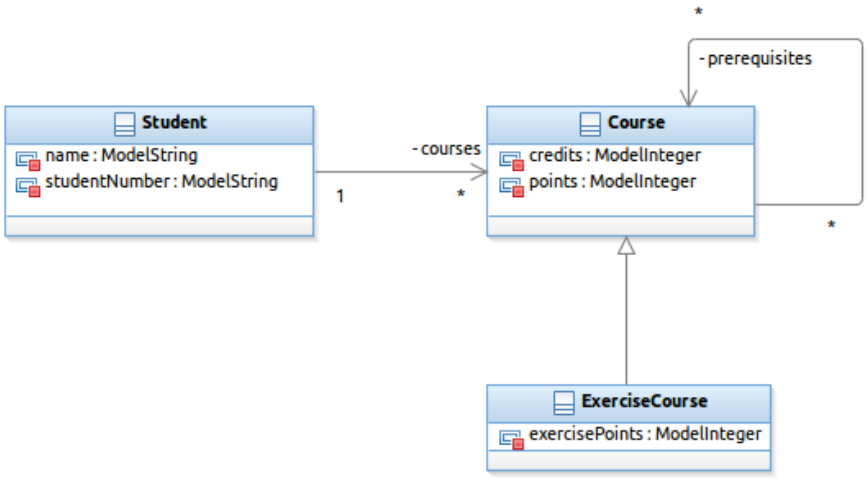


Abbildung 5.1: Das Modell der Beispiele

Das zweite Beispiel ist bereits etwas komplexer und zeigt, wie universell einsetzbar die `iterate`-Operation ist. Es werden die Übungspunkte aller Übungskurse in einem `Bag` gesammelt.

```
self.courses->iterate(c : Course; points : Bag(Integer) = Bag{} |
    if c.oclIsKindOf(ExerciseCourse) then
        points->including(c.oclAsType(ExerciseCourse).exercisePoints)
    else
        points
    endif)
```

Das letzte Beispiel ist eine Bedingung, dass Kurse sich nicht gegenseitig als Voraussetzung haben dürfen. Dabei wird der `forall`-Iterator mit 2 Iterator-Variablen verwendet.

```
self.courses->forall(c1 : Course, c2 : Course |
    c1.prerequisites->includes(c2) implies c2.prerequisites->excludes(c1))
```

Kapitel 6

Umsetzung

Dieses Kapitel beschreibt die tatsächliche Umsetzung dieser Arbeit und erläutert die aufgetretenen Probleme und deren Lösungen.

Der OCL-Parser und der OCL-Editor (mit Code-Completion und Syntax-Highlighting) wurden als Eclipse-basierte Plug-ins für den IBM Rational Software Architect in Java entwickelt.

6.1 Nicht unterstützte OCL-Features

Aufgrund der Generizität des OCL-Parsers, die es erlaubt, mit beliebigen Metamodellen zu arbeiten, ist es nicht möglich, alle Features von OCL umzusetzen. Welche Feature aus diesem oder anderen Gründen nicht implementiert wurden, ist in diesem Abschnitt geschildert.

OCL-Messages

Mittels Messages ist es in OCL möglich abzufragen, ob während der Ausführung einer Operation eine andere Operation aufgerufen wurde [4, S. 29 f.] Da nicht alle Metamodelle in der Lage sind, dies abzufragen, werden Messages im OCL-Parser nicht unterstützt.

Association-Classes

Association-Classes bieten in UML die Möglichkeit, zu einer Association zusätzliche Informationen in einer eigenen Klasse zu speichern. Somit könnte man, wie in Abbildung 6.1 gezeigt wird, unser Universitätsbeispiel modellieren, sodass nicht für jeden Kursbesuch eine Kurs-Instanz erzeugt wird. Auch dies wird nicht von allen Metamodellen und somit auch nicht vom OCL-Parser unterstützt.

States

In UML ist es möglich, Zustandsdiagramme zu modellieren und somit Klassen Zustände (States) zuzuweisen. Auf diese States kann in OCL mit der Funktion `oclIsInState` zugegriffen werden. Da nicht alle Metamodelle Zustände speichern, werden States vom Parser nicht unterstützt.

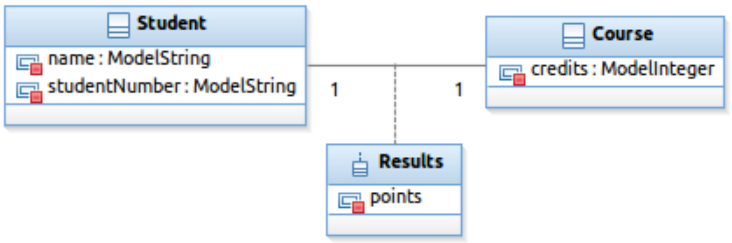


Abbildung 6.1: Association-Classes

@pre

In Postconditions kann mittels `@pre` auf Werte vor Ausführung der Operation zugegriffen werden. Dies wird derzeit von ARL und somit vom OCL-Parser nicht unterstützt.

Collection-Range

Es wäre in OCL möglich, mittels Collection-Rang ers leichter zu generieren, ist im Parser jedoch nicht u 5) würde somit `Sequence(1,2,3,4,5)` entsprechen.

dies hat aber wohl nichts mit uml zu tun? ich dachte ausserdem, dass es hier auch schon eine lösung gibt. zumindest hatten wir dies diskutiert.

Kurze Schreibweise von Collect

Anstatt `courses->collect(credits)` kann in OCL ise `courses.credits` verwendet werden, wird derzeit gs nicht unterstützt.

6.2 OCL-Grammatik

Die verwendete Grammatik für OCL entspricht überwiegend der in der OCL Spezifikation beschriebenen Grammatik [4, S. 61 ff.], unterscheidet sich aber in einigen Details. Diese Unterschiede sind großteils nötig, um die Grammatik zu einer LR(1)-Grammatik zu machen, da beim Parsen ein LALR-Parser verwendet wird. Die komplette Grammatik ist im Anhang A zu finden. Die wichtigsten Produktionen der Grammatik sind im Folgenden beschrieben.

6.2.1 OclExpression

`OclExpression` ist das Startsymbol des Parse-Vorgangs. Es repräsentiert einen beliebigen OCL-Ausdruck. Eine `OclExpression` kann eine `LiteralExp`, textttLetExp, `OclMessageExp`, `IfExp`, `IterateExp`, `OperatorExp` oder eine `ReferenceExp` sein.

6.2.2 LiteralExp

`LiteralExp` repräsentiert alle Literale:

- Integer-Literale: `1`, `5`, `-29`
- Real-Literale: `3.8`, `-31.e2`
- String-Literale: `"Hello"`, `'World'`
- Boolean-Literale: `false`, `true`
- Collection-Literale: `Bag{1, 3, 4}`
- Tuple-Literale: `Tuple{x : Integer = 0, y : String = 'a'}`

6.2.3 OperatorExp

`OperatorExp` behandelt alle arithmetischen und booleschen Operationen, die sowohl in Infix-Notation als auch mittels `.`-Operator geschrieben werden können. Hierzu einige Beispiele:

- `sum + credits`
- `sum.+(credits)`
- `exp1 and exp2`
- `not exp1`
- `exp1.not()`
- `-exp1`

6.2.4 ReferenceExp

`ReferenceExp` ist eine Sammlung verschiedener Expressions der Spezifikation, da diese sich grammatikalisch nicht unterscheiden und die Unterscheidung auf semantischer Ebene erfolgen muss. `ReferenceExp` beinhaltet folgende Ausdrücke:

- Operationen: `getName()`, `self.getName()`, `c.oclIsKindOf(ExerciseCourse)`
- Properties: `name`, `self.name`, `c.points`
- Enumerationen: `Gender::male`
- Statische Attribute: `Course::minStudents`, `ExerciesCourse`, `java::lang::String`
- Variable-Referenzen: `c`, `self`
- Collection-Operationen: `courses->size()`, `points->including(5)`
- Itertor-Operationen: `courses->collect(credits)`, `courses->collect(c : Course | c.credits)`

6.2.5 IterateExp

`IterateExp` behandelt die `iterate`-Operation und wird nicht als Iterator-Operation in `ReferenceExp` behandelt, da sich die Syntax etwas unterscheidet.
`courses->iterate(c : Course; sum : Integer = 0 | sum+c.credits)`

6.2.6 OclMessageExp

Obwohl OCL-Messages durch `OclMessageExp` syntaktisch erlaubt sind, werden diese vom Parser nicht akzeptiert, wie bereits in Abschnitt 6.1 beschrieben wurde.

6.2.7 IfExp

`IfExp` repräsentiert die If-Anweisung in OCL.
`if self.age >= 18 then self.isGrownUp else self.isChild endif`

6.2.8 LetExp

`LetExp` behandelt die Let-Anweisung, welche es erlaubt, für OCL-Ausdrücken einen Alias zu definieren.
`let nChilds : Integer = self.childs->size() in nChilds >= 0`

6.2.9 Operatorrangfolge

Die Rangreihenfolge ist in der Grammatik nicht implizit enthalten und muss deshalb explizit angegeben werden. Die Rangreihenfolge der OCL-Operationen ist wie folgt:

1. `^^`, `^`
2. `. ->`
3. `not` unäres `-`
4. `* /`
5. `+ -`
6. `< > <= >=`
7. `= <>`
8. `and`
9. `or`
10. `xor`
11. `implies`
12. `let`

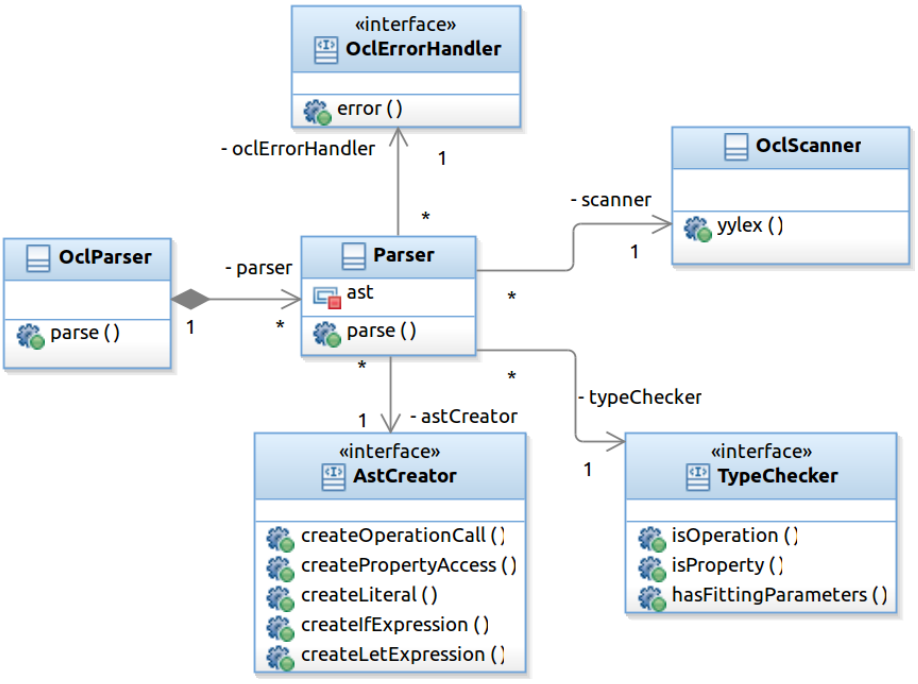


Abbildung 6.2: Überblick über die wichtigsten Klassen des Parsers

6.3 Architektur

In diesem Abschnitt wird die Architektur des Parsers beschrieben. Einen Überblick über die Architektur mit dem Zusammenspiel der wichtigsten Klassen bietet die Abbildung 6.2.

Bei der Architektur wurde darauf geachtet, dass der Parser unabhängig vom erzeugten AST ist und damit die Erzeugung des ASTs leicht austauschbar ist. So sind alle in Abbildung 6.2 gezeigten Klassen und Interfaces unabhängig von ARL.

Da *TypeChecker* und *AstCreator* zusammenarbeiten müssen (d.h. sie müssen die gleiche AST-Repräsentation verwenden), werden diese gemeinsam über eine *OclCodeGenerationFactory* erzeugt. Dies entspricht dem Design-Pattern "Abstract Factory" [13, S. 87 ff.] und gewährleistet, dass *TypeChecker* und *AstCreator* zusammenpassen.

6.3.1 OclParser

OclParser ist die Fassade zum gesamten Parser. Die Klasse steuert die Initialisierung der anderen Klassen mit Hilfe der *OclCodeGenerationFactory* und das Starten des Parse-Vorgangs.

6.3.2 Parser

Die *Parser*-Klasse ist das Kernstück des Parsers. Hier wird die im Abschnitt 6.2 beschriebene Grammatik zur syntaktischen Analyse verwendet. Der Parser erhält vom *OclScanner* die Tokens des zu parsenden Programms. Ergibt der Strom der Tokens eine syntaktisch korrektes Programm wird mittels *TypeChecker* das Programm auf Typverletzungen überprüft und mit dem *AstCreator* ein AST erzeugt.
Die Klasse wurde mit dem Parser-Generator jacc [14] aus einer Beschreibung der Grammatik und semantischen Aktionen generiert.

6.3.3 OclScanner

Die *OclScanner*-Klasse führt die lexikalische Analyse des Programms durch und erstellt dabei die Tokens, die der *Parser* zum Übersetzen nutzt. Diese Klasse wurde mit dem Scanner-Generator JFLex [15] aus einer Beschreibung der lexikalischen Symbole erzeugt.

6.3.4 AstCreator

Der *AstCreator* ist zur Erzeugung des ASTs zuständig, welcher das Ergebnis des Parse-Vorgangs ist. Dabei entspricht der *AstCreator* dem Builder-Design-Pattern [13, S. 97 ff.].

6.3.5 TypeChecker

Der *TypeChecker* dienst zur Überprüfung ob eine Typverletzung vorliegt. Er bietet Operationen zur Überprüfung, ob das angegebenen Element eine Operation, ein Property, eine Enumeration oder ein statisches Attribute ist und ob die tatsächlichen Parameter einer Operation den formalen Parametern zugewiesen werden können.

6.3.6 Erzegung eines Abstract Syntax Trees für ARL

Zur Erzeugung eines AST für ARL gibt es eigene Implementierungen von *AstCreator* und *TypeChecker*, die mit ARL arbeiten. Ein Klassendiagramm dieser ist in Abbildung 6.3 zu sehen.

Ein Problem, dass während der Entwicklung aufgetreten ist, entsteht dadurch, dass ARL keine Typinformationen speichert. Also müssen diese Informationen anders gespeichert werden. Die Lösung dieses Problems bietet das *MappingTypeChecker*-Interface in Zusammenspiel mit dem *TypeFillingAstCreator*. Ein *MappingTypeChecker* bietet die Möglichkeit, Objekte auf Typen abzubilden. Dafür weist der *TypeFillingAstCreator* jedem erzeugten Objekt einen Typ zu und übergibt dieses Paar dem *MappingTypeChecker*, welcher die Abbildung speichert. Dies ist deshalb möglich, da bei der Erzeugung eines Objekts dessen Typ bekannt ist. Wird der Typ eines Objektes benötigt, kann dieser über die Abbildung ermittelt werden. Die Erzeugung des ASTs selbst delegiert der *TypeFillingAstCreator* an einen anderen *AstCreator* (bei ARL an einen *ArlAstCreator*).

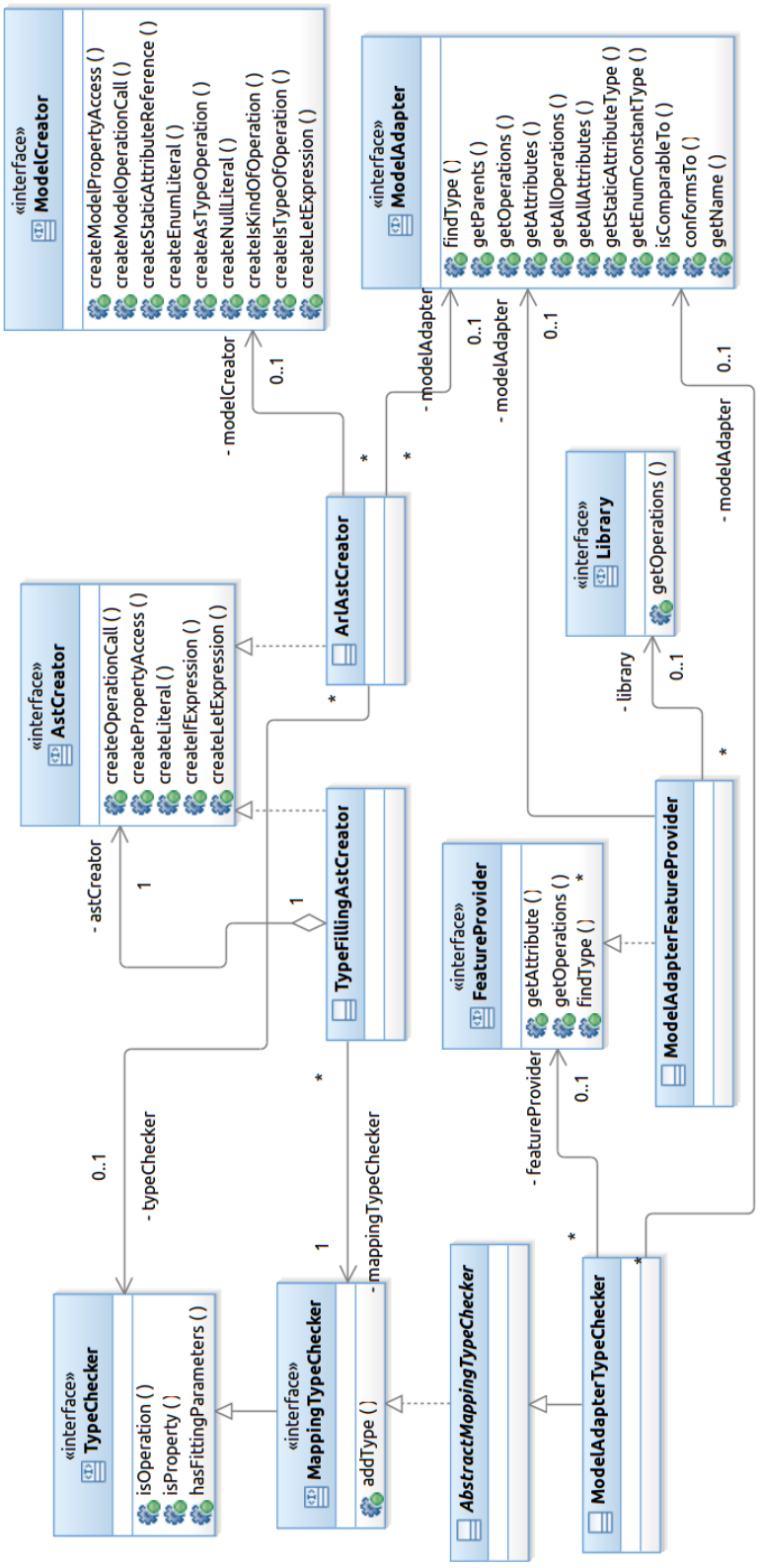


Abbildung 6.3: Implementierung von AstCreator und TypeChecker für ARL

6.3.7 Library

Mit dem *Library*-Interface und dessen derzeitige Implementierung *OclStandardLibrary* können alle Operationen der OCL Standard Library abgerufen werden und auf Typverletzungen überprüft werden.

Die Überprüfung dieser Typverletzungen wird jedoch etwas erschert. Ein Beispiel dafür ist der folgende OCL-Ausdruck:

```
self.courses->collect(c : Course | c.credits)->sum() >= 30
```

Welchen Typ hat das Ergebnis der `collect()`-Funktion? Die Funktion sammelt die Ergebnisse des Body-Ausdrucks (`c.credits`) in einer neuen Collection. Diese neue Collection hat nun entweder `Bag` oder `Sequence` als Typ und als Element-Typ den Typ des Body-Ausdrucks. In unserem Fall ist der gesuchte Typ also `Bag(ModelInteger)`. Hier ergibt sich also der Typ aus dem Objekt, auf dem die Operation angewendet wird (Source-Objekt) und dem Typ des Body-Ausdrucks als Parameter.

Als etwas anders zu ermitteln erweist sich der Ergebnistyp der `sum()`-Funktion. Da die Funktion die Elemente der Collection aufsummiert, besitzt das Ergebnis den Typ der Collection-Elemente. Im obigen Beispiel wird die `sum()`-Funktion auf ein Objekt des Typs `Bag(ModelInteger)` angewandt, wodurch das Ergebnis den Typ `ModelInteger` besitzt.

Umgesetzt werden diese Funktionen mittels *Generics*, wie sie auch aus anderen Programmiersprachen bekannt sind. Dafür wird zusätzlich zu den in Abschnitt 3.1.2 beschriebenen OCL-Datentypen der Typ `Generic` eingeführt. Zur Compilezeit werden dann alle Parameter und Rückgabewerte, die vom Typ `Generic` sind, aufgelöst und so die Typüberprüfung ermöglicht.

Ein weiteres Problem im Zusammenhang mit Typsicherheit stellen die Funktionen `oclAsType()`, `oclIsKindOf()` und `oclIsTypeOf()` dar. Wie man im Beispiel

```
self.courses->iterate(c : Course; points : Bag(Integer) = Bag{} |
  if c.oclIsKindOf(ExerciseCourse) then
      points->including(c.oclAsType(ExerciseCourse).exercisePoints)
  else
      points
  endif)
```

erkennen kann, akzeptieren diese Funktionen nur Typen als Parameter. Laut OCL-Spezifikation [4, S. 141] muss der Parameter vom Typ `Classifier` sein. Allerdings gibt es in OCL keinen `Classifier`-Typ. Würde man nun beliebige Typen erlauben, entsteht neben der fehlenden Typprüfung der Parameter ein weiteres Problem: Welchen Ergebnistyp hat die Methode `oclAsType()`. Um dieses Problem zu lösen wurde ein weiterer OCL-Typ `Classifier` eingeführt. Diese enthält zusätzlich die Information, welchen Typ der Classifier repräsentiert. So kann nun ermittelt werden, welche Typ das Ergebnis der Funktion `oclAsType()` hat.

6.3.8 FeatureProvider

Der `FeatureProvider` bündelt die Features (Operationen und Attribute) der OCL-Typen, also der OCL Standard Library, und des Modells. Dadurch muss der `TypeChecker` keine Unterscheidung zwischen Modell- und OCL-Operationen machen. Darüber hinaus wird der FeatureProvider für die Code-Completion verwendet (siehe Abschnitt 6.6).

6.3.9 Schnittstelle zum Modell

Ziel dieser Arbeit ist es, einen OCL-Parser zu erstellen, der mit einem dynamischen Metamodell umgehen kann. Dass das Metamodell dynamisch ist, bedeutet, dass zur Compilezeit abgefragt werden muss, welche Typen und Operationen es in dem Metamodell gibt. Für diese Abfrage wurde das `ModelAdapter`-Interface erstellt. Je nach eingestetztem Metamodell wird dabei eine andere Implementierung verwendet. Dabei wurde darauf geachtet, das Interface möglichst klein und einfach zu halten, weshalb auch einige Features aus OCL nicht implementiert werden konnten (siehe Abschnitt 6.1).

Das `ModelAdapter`-Interface bildet die Schnittstelle zum Abfragen des Metamodells, jedoch nicht zum Erstellen des ASTs. Um ARL zu ermöglichen mit beliebigen Metamodellen zu arbeiten, wird zur Erstellung der modellbezogenen Operationen eine weitere Schnittstelle benötigt. Diese Schnittstelle ist das `ModelCreator`-Interface.

Die Schnittstelle zum Modell ist im Klassendiagramm der Abbildung 6.3 auf der rechten Seite zu sehen.

6.4 Environments

In dem Beispiel

```
self.courses->collect(c : Course | c.credits)->sum() >= 30
```

wird bei der Iterator-Operation eine neue Variable `c` erstellt. Diese Variable ist allerdings nur bis zum Ende der Operation gültig. Diese Gültigkeitsbereiche von Variablen werden über Environments geregelt. Ein Environment enthält alle Variablen, die in seinem Gülltigkeitsbereich definiert sind. Environments sind hierarchisch gegliedert und es gibt immer ein äußerstes Environment. Dieses äußerste Environment enthält die Variable `self`, die auf das Kontext-Element verweist.

In unserem Beispiel existieren somit 2 Environments:

1. Das äußerste Environment mit der Variable `self`. Dieses Environment ist über den gesamten Ausdruck gültig
2. Das Environment des `collect()`-Iterators mit der Variable `c`. Dieses Environment ist nur innerhalb des Body-Ausdrucks der `collect()`-Operation gültig.

Parameter

Wie im Kapitel 1 beschrieben wurde, ist der Parser in der Lage, neue OCL-Operationen zu definieren. Dies ist aber nur dann von Nutzen, wenn in dieser Operation auch Parameter definiert werden können. Um den Einsatz von Parametern in der Definition einer OCL-Operation zu ermöglichen, kann dem Parser neben dem Kontext-Element noch benannte Parameter angegeben werden. Dadurch enthält das äußerste Environment zusätzlich zur Variable `self` Variablen, die die Parameter repräsentieren.

Variablen ohne explizite Typangabe

In dem oben gezeigten Beispiel wird für die Variable `c` der Typ `Course` definiert. Doch diese Definition ist eigentlich nicht nötig, da der Typ der Variable aus der Collection, auf die die Operation angewandt wird, bekannt ist. Aus diesem Grund ist es in OCL erlaubt, die Typdefinition bei Iterator-Variablen wegzulassen. Dadurch lässt sich das Beispiel auch wie folgt schreiben:

```
self.courses->collect(c | c.credits)->sum() >= 30
```

In diesem Beispiel erhält die Variable `c` implizit den Typ `Course`.

Implizite Variablen

Eine weitere Erleichterung beim Entwickeln bieten implizite Variablen. Environments können eine implizite Variable enthalten. Operationen, die nicht explizit auf ein Objekt angewandt werden, werden auf die implizite Variable angewandt. Dies erlaubt das Weglassen der Variablen im obigen Beispiel:

```
courses->collect(credits)->sum() >= 30
```

Im äußeren Environment ist `self` die implizite Variable. Im inneren Environment der `collect()`-Operation ist die nun unbenannte Iterator-Variable die implizite Variable. Ist eine bestimmte Operation auf die implizite Variable eines Environments nicht gültig, wird die implizte Variable des nächst äußeren Environments verwendet. So wäre beispielsweise auch der folgende OCL-Ausdruck gültig:

```
courses->collect(credits + studentNumber)
```

Hierbei wird `credits` der Iterator-Variable verwendet und `studentNumber` von `self`.

Implizite Variablen und das Weglassen der Typdefinition sind auch bei der `iterate()`-Operation erlaubt:

```
self.courses->iterate(sum : Integer = 0 |
    sum+credits) >= 30
```

Das Weglassen der Typangabe und implizite Variablen sind sehr praktisch und ersparen einiges an Schreibarbeit, führen beim Bottom-Up-Parsing allerdings zu einem Problem. Das Erstellen des Environments und damit der impliziten Variable kann erst erfolgen, wenn die Iterator-Operation geparst wurde. Da ein Bottom-Up-Parser allerdings von unten nach oben oder in

unserem Fall von innen nach außen parst, wird der Body-Ausdruck vor der Iterator-Operation geparst. Dadurch stehen im Body-Ausdruck die benötigten Variablen nicht zur Verfügung. Deshalb wurde in der Grammatik die Produktion der Iterator-Operationen in 2 Produktionen aufgeteilt:

```
CollectionOpExp     := CollectionOperation '(' VarDeclaration
                       '|' OclExpression ')'
CollectionOperation := OclExpression '->' SimpleName
```

Dadurch wird ermöglicht, beim Erkennen von `CollectionOperation` zu ermitteln, ob es sich dabei um eine Iterator-Operation handelt und gegebenenfalls ein Environment mit dazugehöriger impliziter Variable zu definieren. Ist eine Variablendeklaration vorhanden, wird die implizite Variable verworfen und die neue Variable zum aktuellen Environment hinzugefügt. Gibt es für die Variable keine Typdeklaration wird der Typ der impliziten Variable verwendet.

6.5 Typ-Modell

Wenn man aus dem Beispiel

```
self.courses->iterate(c : Course; sum : Integer = 0 |
    sum+c.credits) >= 30
```

die Addition `sum+c.credits` betrachtet, fällt auf, dass hierbei 2 unterschiedliche Typen addiert werden; der OCL-Typ *Integer* und der Typ des Modells *ModelInteger*. Tatsächlich findet hier ein Aufruf der Operation `+` statt: `sum.+(c.credits)`. Diese Operation akzeptiert als Parameter *Integer* oder *Real*. Allerdings sollte sie auch `ModelInteger` akzeptieren. Um dies zu ermöglichen, müssen die beiden unabhängigen Typsysteme (OCL, Modell) miteinander verwoben werden. Um dies zu realisieren, ergibt sich der Typ eines Objekts im OCL-Parser aus einem OCL-Typ und einem Modell-Typ. Eine Ausnahme bilden hier Variablen wie `sum` oder Literale in OCL, da sie über keinen Modell-Typ verfügen.

Daraus ergeben sich nun neue Regeln für die Zuweisungskompatibiltät. Als Notation für die Regeln wird die Tupel-Schreibweise verwendet. Ein Typ mit OCL-Typ *O1* und Modell-Typ *M1* kann somit als *(O1,M1)* geschrieben werden. Gibt es keinen Modell-Typ wird *null* als Modell-Typ angegeben. Es ergeben sich folgende Regeln:

1. *(O1,M1)* ist zu *(O2,M2)* zuweisungskompatibel, wenn *O1* zu *O2* zuweisungskompatibel ist und *M1* zu *M2* zuweisungskompatibel ist.
2. *(O1,M1)* ist zu *(O3,null)* zuweisungskompatibel, wenn *O1* zu *O3* zuweisungskompatibel ist.
3. *(O3,null)* ist zu *(O1,M1)* zuweisungskompatibel, wenn *O3* zu *O1* zuweisungskompatibel ist und *O3* zu *M1* zuweisungskompatibel ist.

Die Regeln der Zuweisungskompatibilität für OCL-Typen sind wie in Abschnitt 3.1.2 beschrieben. Zur Überprüfung der 1. und der 3. Regel muss die Zuweisungskompatibiltät zwischen Modell-Typen und von einem

OCL-Typ zu einem Modell-Typ überprüft werden. Diese Überprüfung findet über die Methoden `conformsTo(OCL-Typ, Modell-Typ)` und `conformsTo(Modell-Typ, Modell-Typ)` des *ModelAdapters* statt.

Wenn man nun das obige Beispiel nochmal betrachtet ergeben sich für `sum` der Typ *(Integer,null)* und für `c.credits` der Typ *(Integer,ModelInteger)*. Dadurch ist c.credits nun auch ein *Integer* und ist somit als Funktionsparameter gültig.

6.6 Code-Completion

Grundsätzlich gibt es 2 Möglichkeiten, Code-Completion anzubieten:

1. Es werden an der aktuellen Position alle gültigen Tokens angeboten
2. Es werden nur an bestimmten Postionen gewisse gültige Tokens angeboten

In dieser Arbeit wurde der zweite Ansatz gewählt, da dieser wesentlich einfacher zu implementieren ist und in den meisten Fällen ausreicht. Konkret wird nur bei Operationsaufrufen und Attributzugriffe nach `.` oder `->` Code-Completion angeboten.

Wie bereits erwähnt, wurde der OCL-Editor als Eclipse-basiertes Plug-In entwickelt. Die Darstellung und interne Repräsentation des Texts sowie die Anzeige und Steuerung der Code-Completion wird von Eclipse verwaltet. Dadurch ist es lediglich nötig, bei Aufruf der Code-Completion Eclipse eine Liste der Kandidaten (Proposals) für das Einfügen in den Text zu liefern. Das Hauptproblem ist dabei, wie man den Typ des Objekts, welches vor dem Punkt- oder Pfeil-Operator steht, ermitteln kann. Dafür wird der Text bis zur aktuellen Cursorposition geparst, sobald die Proposals angefordert werden. Der Parser speichert sich dabei zu jeder Produktion den Typ der Produktion und die Position an dem die Produktion zu Ende ist. Beispielsweise wird bei `self.courses->collect(credits)` der Typ *(Bag(Integer,ModelInteger),null)* und die Position von `)` gespeichert. Nun kann über die Funktion `getTypeAt(int)` der Typ an einer Position abgefragt werden.

Ist der Typ ermittelt, können damit die gültigen Operationen und Attribute über den FeatureProvider (siehe Abbildung 6.3) ermittelt und daraus die Prosposals erzeugt werden.

Zur Ermittlung des Typs muss der zu parsende OCL-Ausdruck nicht komplett korrekt sein. Es reicht, wenn der Ausdruck bis zur Position des Cursors korrekt geparst werden kann. Eventuell auftretende Fehlermeldungen des Parsers werden von der Code-Completion ignoriert.

6.7 Syntax-Highlighting

Beim Syntax-Highlighting des OCL-Editors findet kein Parse-Vorgang statt, sondern die Ermittlung der unterschiedlich darzustellenden Tokens wird nur mittels Scanner durchgeführt. Diese Vorgangsweise hat den Nachteil, dass beispielsweise keine Unterscheidung zwischen Variablenname und Properties gemacht wird und auch keine syntaktischen Fehler angezeigt werden können.

Der Grund für diese Implementierung ist die Tatsache, dass wenn ein gültiger AST vorliegen soll, dieser bei jeder Code-Änderung angepasst werden muss. Um dabei performant zu bleiben, kann nicht jedes Mal der gesamte Ausdruck neu geparst werden, was bedeutet, dass der Parser teilweise parsen müsste. Derzeit ist der OCL-Parser zu so einem teilweisen Parsen nicht in der Lage.

Kapitel 7

Evaluierung

Dieses Kapitel befasst sich mit der Evaluierung der Umsetzung. Die Evaluierung erfolgte getrennt für die drei Teilbereiche der Arbeit (Parser, Code-Completion, Syntax-Highlighting).

7.1 Parser

Für den Parser wurden zwei Faktoren evaluiert. Zum einen wurde die Korrektheit des Parsers validiert und zum anderen das Laufzeitverhalten.

7.1.1 Korrektheit

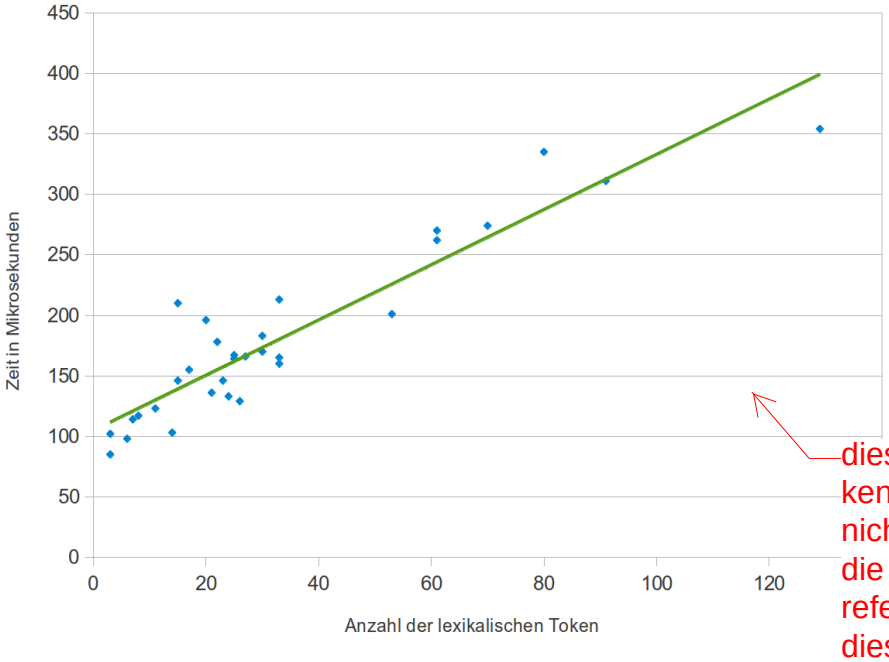
Zur Überprüfung der Korrektheit wurde eine Reihe von Testfällen herangezogen. Das Ergebnis dieser Testfälle wurden mit dem Ergebnis eines Referenzparsers verglichen. Als Referenzparser diente der Eclipse OCL-Parser [7]. Das Metamodell dabei war Eclipse UML2. Eine Liste aller Testfälle ist im Anhang B zu finden. Alle getesteten Testfälle wurden vom Parser erfolgreich absolviert.

7.1.2 Performanz

Ein weiterer wichtiger Qualitätsfaktor ist die Performanz des Parsers. Auch hier wurde als Referenzparser der Eclipse OCL-Parser [7] verwendet. Es wurden vier verschieden komplexe OCL-Ausdrücke gewählt, um die Leistung der Parser für unterschiedliche OCL-Ausdrücke vergleichen zu können. Als Schätzung für die Komplexität eines OCL-Ausdrucks wurde die Anzahl der lexikalischen Symbole in einem Ausdruck gewählt, da diese einfach und automatisiert ermittelt werden kann.

Um Einflüsse durch JIT-Compilation und Garbage Collection der Java Virtual Machine oder Einflussfaktoren des Betriebssystems zu reduzieren, wurden die getesteten OCL-Ausdrücke öfters hintereinander geparst und die Gesamtzeit durch die Anzahl der Parse-Vorgänge dividiert. Diese Vorgehensweise erhöht zusätzlich die Genauigkeit der Messergebnisse. Konkret wurde zur Ermittlung der Performanz jeder OCL-Ausdruck 100-mal geparst.

Die Ergebnisse der Tests sind in der folgenden Tabelle zusammengefasst:



dieses ergebnis kenne ich noch gar nicht. vielleicht auch die zeit des referenzparsers in diese tabelle. warum ist der referenzparser so langsam?

Abbildung 7.1: Laufzeit des Parsers bei unterschiedlich kom OCL-Ausdrücken

Anzahl Symbole	Zeit	Referenzzeit
15	460 µs	5560 µs
33	1340 µs	67100 µs
33	1500 µs	91780 µs
129	1750 µs	147160 µs

Wie aus der Tabelle erkennbar ist, arbeitet der erstellte Parser wesentlich schneller als der Referenzparser.

7.1.3 Skalierbarkeit

Als letztes Qualitätsmerkmal wurde die Skalierbarkeit des Parsers getestet. Dabei soll geklärt werden, ob der Parser sich bei komplexen OCL-Ausdrücken genau so effizient verhält, wie bei einfacheren Ausdrücken.

Um die Einflüsse des Metamodells auf die Tests zu reduzieren, wurde ein sehr einfaches Metamodell erstellt und die im Anhang B gezeigten Testfälle an das Test-Metamodell angepasst.

Als Messmethode wurde die im Abschnitt 7.1.2 vorgestellte Methode verwendet. Die Anzahl der Parse-Vorgänge konnte allerdings auf Grund des einfacheren Metamodells und der Tatsache, dass kein Referenzparser verwendet wurde, auf 20000 erhöht werden.

Die Ergebnisse der Auswertung sind in Abbildung 7.1 zu sehen. Dabei ist zu erkennen, dass die Laufzeit mit steigender Komplexität lediglich linear steigt.

7.2 Code-Completion

Die Evaluierung der Code-Completion besteht lediglich aus der Validierung der Korrektheit. Die Performanz wurde nicht im Detail untersucht, da sie mit der Performanz des Parsers korreliert. Tests zeigen jedoch, dass die Proposals mit kaum merkbarer Verzögerung angezeigt werden.

Die Validierung der Korrektheit wurde mit den in Anhang B beschriebenen Testfällen durchgeführt. Dabei wurde der Testausdruck nach `.` und `->` gescannt. Wurde ein solcher Operator gefunden, werden für diese Stelle die Proposals berechnet. Ist der hinter dem Operator stehende Operations- oder Propertyname auch in den Proposals vorhanden, war der Test erfolgreich. Die Code-Completion war in der Lage alle Testfälle erfolgreich zu absolvieren.

7.3 Syntax-Highlighting

Für das Syntax-Highlighting beschränkte sich die Evaluierung auf händische Tests, da dabei automatisierte Tests aufwendig sind.

Kapitel 8

Fazit und Ausblick

Es ist mit dieser Arbeit gelungen, einen OCL-Parser mit dazugehörigem Editor zu schaffen, die mit einem zur Laufzeit veränderbaren Metamodell umgehen können. Dabei wurde die Schnittstelle zum Metamodell einfach gehalten, um so mit wenig Aufwand beliebige Metamodelle zu unterstützen. Derzeit gibt es erst wenige Metamodelle, für die das Interface implementiert ist. So kann über die Adäquatheit der Schnittstelle noch keine exakte Aussage getroffen, sondern muss sich mit steigender Anzahl an Metamodellen beweisen müssen. Der Parser erwies sich als sehr performant und skalierbar. Er unterstützt den Großteil der offiziellen OCL-Spezifikation sowie die OCL Standard Library.

Der Editor unterstützt den Entwickler mit Syntax-Highlighting und Code-Completion. Dem Entwickler werden durch die Code-Completion Operationen und Properties des Modells, als auch der OCL Standard Libray angeboten. Diese Code-Completion könnte jedoch noch auf weitere Elemente erweitert werden. Vor allem interessant wäre die Unterstützung von Code-Completion für implizite Variablen. Auch im Syntax-Highlighting steckt noch Verbesserungspotential. Eine Unterscheidung zwischen Variablen und Properties und die Anzeige von Fehlern würden dem Entwickler das Programmieren zusätzlich erleichtern. Damit das funktioniert, müsste das Syntax-Highlighting auf Basis des ASTs umgesetzt werden, da nur dort eine Unterscheidung zwischen Variablen und Properties möglich ist.

ich denke ein noch größeres thema wäre was mit einem constraint passieren sollen wenn sich das meta model ändernt. das wäre z.b. eine tolle masters thesis!

Literaturverzeichnis

[1] Ian Sommerville. *Software Engineering.* Pearson Education, neunte Auflage, 2011.

[2] Alexander Reder und Alexander Egyed. "Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML". In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10)*, Seiten 347–348. ACM, 2010.

[3] Andreas Demuth, Roberto E. Lopez-Herrejon und Alexander Egyed. "Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking". In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11)*, Seiten 452–455. ACM, 2011.

[4] OMC OCL v. 2.2 specification. http://www.omg.org/spec/OCL/2.2/ (abgerufen am 06.02.2012)

[5] Niklaus Wirth. *Compiler Construction.* Addison-Wesley, 1996.

[6] Alfred V. Aho, Ravi Sethi und Jefrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[7] Eclipse OCL. http://www.eclipse.org/modeling/mdt/?project=ocl (abgerufen am 07.02.2012)

[8] Dresden OCL. http://www.dresden-ocl.org (abgerufen am 07.02.2012)

[9] Squeak. http://squeak.org/ (abgerufen am 08.02.2012)

[10] RoelTyper. http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/ (abgerufen am 08.02.2012)

[11] Eclipse. http://www.eclipse.org/ (abgerufen am 08.02.2012)

[12] Romain Robbes und Michele Lanza. "How Program History Can Improve Code Completion". In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE 2008)*, Seiten 317–326. ACM, 2008.

[13] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 2009.

[14] jacc: just another compiler compiler for Java. http://web.cecs.pdx.edu/ mpj/jacc/ (abgerufen am 15.02.2012)

[15] JFlex. http://www.jflex.de/ (abgerufen am 15.02.2012)

[16] Eclipse UML2. http://www.eclipse.org/modeling/mdt/?project=uml2 (abgerufen am 19.02.2012)

Anhang A

OCL-Grammatik

```
OclExpression       : LiteralExp
                    | LetExp
                    | OclMessageExp
                    | IfExp
                    | IterateExp
                    | OperatorExp
                    | ReferenceExp
                    ;

SimpleName          : NAME
                    ;

PathName            : SimpleName
                    | PathName '::' SimpleName
                    ;

LiteralExp          : PrimitiveLiteralExp
                    | CollectionLiteralExp
                    | TupleLiteralExp
                    ;

CollectionLiteralExp    : CollectionLiteralTypeIdentifier '{'
                            CollectionLiteralParts '}'
                        | CollectionLiteralTypeIdentifier '{' '}'
                        ;

CollectionLiteralTypeIdentifier : 'Set'
                                | 'Bag'
                                | 'Sequence'
                                | 'OrderedSet'
                                ;

CollectionTypeIdentifier    : CollectionLiteralTypeIdentifier
                            | 'Collection'
                            ;
```

```
CollectionLiteralParts  : CollectionLiteralPart
                        | CollectionLiteralParts ','
                            CollectionLiteralPart
                        ;

CollectionLiteralPart   : CollectionRange
                        | OclExpression
                        ;

CollectionRange         : OclExpression '..' OclExpression
                        ;

PrimitiveLiteralExp     : IntegerLiteralExp
                        | RealLiteralExp
                        | StringLiteralExp
                        | BooleanLiteralExp
                        | NullLiteralExp
                        | InvalidLiteralExp
                        ;

TupleLiteralExp     : 'Tuple' '{' TuplePartList '}'
                    ;

IntegerLiteralExp   : INTEGER
                    ;

RealLiteralExp      : REAL
                    ;

StringLiteralExp    : STRING
                    ;

NullLiteralExp      : 'null'
                    ;

InvalidLiteralExp   : 'invalid'
                    ;

BooleanLiteralExp   : 'false'
                    | 'true'
                    ;

IterateExp          : IterateOperation '(' IteratorVarDeclaration
                        ';' InitVarDeclaration '|' OclExpression ')'
                    | IterateOperation '(' InitVarDeclaration
                        '|' OclExpression ')'
                    ;
```

```
IterateOperation    : OclExpression '->' 'iterate'
                    ;

IteratorVarDeclaration : SimpleName
            | SimpleName ':' Type
                       ;

VariableDeclaration     : SimpleName
                        | SimpleName ':' Type
                        ;

VariableDeclarationList : VariableDeclaration
                        | VariableDeclarationList ','
                              VariableDeclaration
                        ;

TuplePart     : SimpleName ':' Type
                    | SimpleName '=' OclExpression
                    | SimpleName ':' Type '=' OclExpression
                    ;

TuplePartList       : TuplePart
                    | TuplePartList ',' TuplePart
                    ;

Type                : PathName
                    | CollectionType
                    | TupleType
                    ;

CollectionType      : CollectionTypeIdentifier '(' Type ')'
                    ;

TupleType           : 'Tuple' '(' ')'
                    | 'Tuple' '(' VariableDeclarationList ')'
                    ;

ReferenceExp        : SimpleName
                    | SimpleName IsMarkedPre
                    | SimpleName IsMarkedPre '(' ')'
                    | SimpleName IsMarkedPre '(' Arguments ')'
                    | SimpleName '[' Arguments ']'
                    | SimpleName '[' Arguments ']' IsMarkedPre
                    | PathName '(' ')'
                    | PathName '(' Arguments ')'
                    | PathName '::' SimpleName
                    | OclExpression '.' SimpleName
                    | OclExpression '.' SimpleName IsMarkedPre
                    | OclExpression '.' SimpleName '('  ')'
                    | OclExpression '.'
```

```
                        SimpleName IsMarkedPre '(' ')'
                    | OclExpression '.'
                        SimpleName '(' Arguments ')'
                    | OclExpression '.' SimpleName IsMarkedPre
                        '(' Arguments ')'
                    | OclExpression '.' SimpleName
                        '[' Arguments ']'
                    | OclExpression '.' SimpleName
                        '[' Arguments ']' IsMarkedPre
                    | OclExpression '.' PathName '::' SimpleName
                    | OclExpression '.' PathName '::'
                        SimpleName IsMarkedPre
                    | OclExpression '.' PathName '::'
                        SimpleName '(' ')'
                    | OclExpression '.' PathName '::'
                        SimpleName IsMarkedPre '(' ')'
                    | OclExpression '.' PathName '::'
                        SimpleName '(' Arguments ')'
                    | OclExpression '.' PathName '::'
                        SimpleName IsMarkedPre '(' Arguments ')'
                    | CollectionOperation '(' ')'
                    | CollectionOperation '(' Arguments ')'
                    | CollectionOperation '(' IteratorVarDeclaration
                      '|' OclExpression ')'
            | MultiVarIterator '(' OclExpression ')'
            | MultiVarIterator '(' IteratorVarDeclaration
                        '|' OclExpression ')'
                    | MultiVarIterator '(' IteratorVarDeclaration
                        ',' IteratorVarDeclaration
                        '|' OclExpression ')'
                    ;

CollectionOperation   : OclExpression '->' SimpleName
                      ;

MultiVarIterator   : OclExpression '->' MULTIVARITERATOR
                     ;

IsMarkedPre         : '@' 'pre'
                    ;

Arguments           : OclExpression
                    | OclExpression ',' Arguments
                    ;

OperatorExp         : OclExpression '+' OclExpression
            | OclExpression '.' '+' '(' OclExpression ')'
                    | OclExpression '-' OclExpression
                    | OclExpression '.' '-' '(' OclExpression ')'
                    | OclExpression '*' OclExpression
```

```
                    | OclExpression '.' '*' '(' OclExpression ')'
                    | OclExpression '/' OclExpression
                    | OclExpression '.' '/' '(' OclExpression ')'
                    | OclExpression 'implies' OclExpression
                    | OclExpression '.' 'implies'
                       '(' OclExpression ')'
                    | OclExpression 'xor' OclExpression
                    | OclExpression '.' 'xor' '(' OclExpression ')'
                    | OclExpression 'or' OclExpression
                    | OclExpression '.' 'or' '(' OclExpression ')'
                    | OclExpression 'and' OclExpression
                    | OclExpression '.' 'and' '(' OclExpression ')'
                    | OclExpression '<>' OclExpression
                    | OclExpression '.' '<>' '(' OclExpression ')'
                    | OclExpression '=' OclExpression
                    | OclExpression '.' '=' '(' OclExpression ')'
                    | OclExpression '<' OclExpression
                    | OclExpression '.' '<' '(' OclExpression ')'
                    | OclExpression '>' OclExpression
                    | OclExpression '.' '>' '(' OclExpression ')'
                    | OclExpression '<=' OclExpression
                    | OclExpression '.' '<=' '(' OclExpression ')'
                    | OclExpression '>=' OclExpression
                    | OclExpression '.' '>=' '(' OclExpression ')'
                    | '(' OclExpression ')'
                    | '-' OclExpression
                    | OclExpression '.' '-' '(' ')'
                    | 'not' OclExpression
                    | OclExpression '.' 'not' '(' ')'
                    ;

LetExp              : LetKeyWord InitVarDeclaration LetExpSub
                    ;

LetKeyWord          : 'let'
                    ;

InitVarDeclaration  : SimpleName ':' Type '=' OclExpression
                    ;

LetExpSub           : ',' InitVarDeclaration LetExpSub
                    | 'in' OclExpression
                    ;

OclMessageExp       : OclExpression '^' SimpleName '(' ')'
                    | OclExpression '^' SimpleName '('
                       OclMessageArguments ')'
                    | OclExpression '^^' SimpleName '(' ')'
                    | OclExpression '^^' SimpleName '('
                       OclMessageArguments ')'
```

```
                    ;

OclMessageArguments : OclMessageArg
                    | OclMessageArguments ',' OclMessageArg
                    ;

OclMessageArg       : '?'
                    | '?' ':' Type
                    | OclExpression
                    ;

IfExp               : 'if' OclExpression
                        'then' OclExpression
                        'else' OclExpression 'endif'
                    ;
```

A.1 Lexikalische Symbole

```
Character = [a-zA-Z_0-9]
DecDigit = [0-9]
Sign = [-]
NAME = {Character}+
INTEGER = {Sign}? {DecDigit}+
REAL = {Sign}? {DecDigit} + "." {DecDigit}+ ? ([eE] {Sign}? {DecDigit}+ )?
STRING = [\"\'] [^\n\r\"\']+ ? [\"\']
MULTIVARITERATOR = "forAll" | "exists"
```

Anhang B

Testfälle

Die folgenden Testfälle wurden mit dem OCL-Parser und Eclipe UML2 als Metamodell getestet.

Legende

```
<Kontext>:    <OCL-Ausdruck>
<Kontext>(<Parameter>):   <OCL-Ausdruck>
```

Liste aller Testfälle

```
Message:      self.messageKind
Class:        6 + 7
Class:        6 - 7
Class:        6 * 7
Class:        6 / 7
Class:        6 = 7
Class:        6 <> 7
Class:        6 < 7
Class:        6 > 7
Class:        6 <= 7
Class:        6 >= 7
Class:        -6
Class:        6 * 28.4
Class:        6 - 28.4
Class:        6 + 28.4
Class:        38.58 + 48.29e3
Class:        38.58 - 48.29e3
Class:        38.58 * 48.29e3
Class:        38.58 / 48.29e3
Class:        38.58 = 48.29e3
Class:        38.58 <> 48.29e3
Class:        38.58 < 48.29e3
Class:        38.58 > 48.29e3
Class:        38.58 <= 48.29e3
Class:        38.58 >= 48.29e3
```

```
Class:        -38.58
Class:        true implies false
Class:        true xor false
Class:        true or false
Class:        true and false
Class:        true = false
Class:        true <> false
Class:        not true
Class:        'test'
Class:        'test' = 'test'
Class:        'test' <> 'test'
Class:        'te' + 'st'
Class:        let x : Collection(Integer) = Bag{1, 2} in
                  x->forAll(i | i<3)
Class:        let x : Collection(Integer) = Bag{1, 2} in
                  x->forAll(i, i2 | i<>i2)
Class:        let x : Collection(Integer) = Bag{1, 2} in
                  x->includes(2)
Class:        let x : Collection(Integer) = Bag{1, 2} in
                  x->size()
Class:        Set{1, 2}->select(i | i=3)
Class:        Set{1, 2}->union(Bag{2, 3})
Class:        Set{1, 2}->union(Set{2, 3})
Class:        self.getAllOperations()->collect(datatype)
Class:        Bag{1, 2}->select(i | i=3)
Class:        Bag{1, 2}->union(Bag{2, 3})
Class:        Bag{1, 2}->union(Set{2, 3})
Class:        Bag{1, 2}->collect(x | x)
Class:        OrderedSet{1, 2}->select(i | i=3)
Class:        OrderedSet{'String', 'Test'}->at(1)
Class:        OrderedSet{'String', 'Test'}->indexOf('Test')
Class:        OrderedSet{1, 2}->union(Bag{2, 3})
Class:        OrderedSet{1, 2}->union(Set{2, 3})
Class:        OrderedSet{1, 2}->union(OrderedSet{2, 3})
Class:        OrderedSet{1, 2}->collect(x | x)
Class:        Sequence{1, 2}->select(i | i=3)
Class:        Sequence{'String', 'Test'}->at(1)
Class:        Sequence{'String', 'Test'}->indexOf('Test')
Class:        Sequence{1, 2}->union(Bag{2, 3})
Class:        Sequence{1, 2}->union(Set{2, 3})
Class:        Sequence{1, 2}->union(Sequence{2, 3})
Class:        Sequence{1, 2}->collect(o|o)
Class:        Sequence{1, 2}->collect(o|o)->first()
Class:        let x2 : Integer = 5 in x2 = 5
Class:        let x5 : Real = 5 in x5
Class:        let x6 : Bag(Integer) = Bag{1, 2, 3} in
                  x6->includes(2)
Class:        let x8 : Bag(Integer) = Bag{} in x8
Class:        let x9 : Bag(Integer) = Bag{}, y9 : Integer = 5 in
                   x9->including(y9)
```

```
Class:        getAllOperations()->iterate(x: Operation;
                  acc : Bag(Operation) = Bag{} |
                  acc->including(x) )
Class:        self.getAllOperations()->forAll(o : Operation |
                  o.isTemplate())
Class:        getAllOperations()->forAll(isLeaf)
Class:        getAllOperations()->iterate(acc : Bag(Integer)
                  = Bag{} | acc->including(lower))

Class(x : Integer, y : Integer):
x + y

Class(x : Integer, y : Integer):
self.getAllOperations()->forAll(o | o.getLower() < x * y)

Class:
let children:Set(NamedElement) =
    self.namespace.oclAsType(Package).packagedElement->
select(pe:PackageableElement|pe.oclIsTypeOf(Class) and
pe.oclAsType(Class).allParents()->includes(self)) in
self.ownedAttribute->forAll(p:Property|
            p.type.oclIsTypeOf(Class) implies
            children->excludes(p.type.oclAsType(Class)))

Message:
self.receiveEvent.oclAsType(InteractionFragment).covered->
forAll(represents.type.oclAsType(Class).ownedOperation->
exists(name=self.name))

Message:
self.receiveEvent.oclAsType(InteractionFragment).covered->
  exists(let rc:Class=represents.type.oclAsType(Class) in
    self.sendEvent.oclAsType(InteractionFragment).covered->
  exists(let sc:Class=represents.type.oclAsType(Class) in
sc.ownedAttribute->
exists(association<>null implies type=rc)
  )
  )

Transition:
self.owner.oclAsType(Region).stateMachine<>null implies
let classifier:BehavioredClassifier=
        self.owner.oclAsType(Region).stateMachine.getContext() in
classifier.oclIsTypeOf(Class) implies
classifier.oclAsType(Class).ownedOperation->
            exists(o:Operation|o.name=self.name)

Association:
self.memberEnd->forAll(p1:Property|
self.memberEnd->forAll(p2:Property|
```

```
        p1<>p2 implies p1.name<>p2.name))

Association
self.memberEnd->size()>0 implies self.memberEnd->select(p |
    p.aggregation<>AggregationKind::none)->size()<=1

Association:
self.memberEnd<>null and self.memberEnd->forAll(p|
    p.type<>null and p.type.namespace=self.namespace)

Property:
(self.association<>null and
    self.aggregation=AggregationKind::composite) implies
(self.upper>=0 and self.upper <=1)

Operation:
self.ownedParameter->forAll(p1|
    self.ownedParameter->forAll(p2|
        p1<>p2 implies p1.name<>p2.name))

Operation:
self.ownedParameter->forAll(p:Parameter|
    p.type<>null implies
    p.type.namespace=self.owner.oclAsType(Class).namespace)

Class:
self.ownedAttribute->forAll(p1:Property |
    self.ownedAttribute->forAll(p2:Property|
        p1<>p2 implies p1.name<>p2.name))

Class:
self.ownedOperation->forAll(o1:Operation|
  self.ownedOperation->forAll(o2:Operation|o1<>o2 implies
(o1.name <> o2.name or
  o1.ownedParameter->size() <> o2.ownedParameter->size() or
let ops1:Collection(Type)=
          o1.ownedParameter->collect(i:Parameter | i.type) in
let ops2:Collection(Type)=
          o2.ownedParameter->collect(i:Parameter | i.type) in
  ops1->exists(t:Type|ops2->excludes(t)) or
  ops2->exists(t:Type|ops1->excludes(t)))))

Package:
self.ownedElement->forAll(e1:Element|
self.ownedElement->forAll(e2:Element|
        (e1<>e2 and e1.oclIsKindOf(NamedElement) and
            e2.oclIsKindOf(NamedElement)) implies
(e1.oclAsType(NamedElement).name <>
            e2.oclAsType(NamedElement).name)))
```

```
Interface:
self.ownedAttribute->forAll(pr:Property|
    pr.association<>null or
    pr.visibility=VisibilityKind::public) and
self.ownedOperation->forAll(o:Operation|
        o.visibility=VisibilityKind::public)

Class:
not allParents()->includes(self)

Generalization:
self.source->forAll(e1:Element|e1.oclIsKindOf(NamedElement)
    implies self.target->forAll(e2:Element|
        e2.oclIsKindOf(NamedElement) and
e1.oclAsType(NamedElement).namespace =
        e2.oclAsType(NamedElement).namespace))

Class:
let children:Set(NamedElement) =
    self.namespace.oclAsType(Package).packagedElement->
select(pe:PackageableElement|pe.oclIsTypeOf(Class) and
pe.oclAsType(Class).allParents()->includes(self)) in
self.ownedOperation->forAll(o:Operation|
                o.ownedParameter->forAll(p:Parameter|
                    p.type.oclIsTypeOf(Class) implies
                    children->excludes(p.type.oclAsType(Class))))

Class:
self.allParents()->forAll(c:Classifier|
    c.oclAsType(Class).ownedAttribute->forAll(p:Property|
        self.ownedAttribute->collect(name)->excludes(p.name)))

Generalization:
5 > 3 * 4.3 / (8 + 2) - -6 implies
true xor 7 < 8 or not true and 5.3 <= 58 = ('Test' <> 'Hallo')

Class:
Tuple { x:Class, y:Generalization, z:Interface }.x.allParents()

Generalization:
5 > 3 implies
true xor 7 < 8 or not true and 5 <= 58 = ('Test' <> 'Hallo')

Generalization:
Bag { 4, 5, 3}->forAll(i:Integer | i < 6)
```

Testfälle, bei denen der Parser einen Fehler melden sollte

```
TestingStuff: self.messageKind
Class:        self.messageKind
```

```
Class:        let x1 : Integer = 'test' in x1 = 5
Class:        let x3 = 5 in x3 = 5
Class:        let x4 : Integer in x4 = 5
Class:        let x7 : Bag(Integer) = Bag{true, false} in
                    x7->includes(2)
Class:        self.getAllOperations()->iterate(x : Class;
                    acc : Bag(Class) = Bag{} | acc->including(x))
Class:        self.getAllOperations()->forAll(o : Class |
                    o.getAllOperations()->isEmpty())
```