Technische Universität München

Fakultät für Informatik

Systementwicklungsprojekt
(Projektbericht)

Erweiterung und Optimierung des AQuA-Systems

Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dr. Bernhard Schätz
Bearbeiter:	Florian Hölzl
Bearbeitungszeitraum:	Januar 2004 – Juli 2004
Erweiterungen	Oktober 2004 – Januar 2005
Berichtsversion:	1.02
Berichtsdatum:	20. Januar 2005

Inhalt

Kapitel 1 Einführung 5
1.1. Hintergrund: AutoFocus, Quest und ODL 5
1.2. Zielsetzung und Struktur des Projektberichts 5
Kapitel 2 Motivation 7
Kapitel 3 Neustrukturierung des Typsystems 9
3.1. Begrenzte, unbegrenzte und post-universe Typen 9
3.2. Verwendung der Typklassen mit den Quantoren 10
3.3. Das neue Typsystem in der ODL-Grammatik 11
Kapitel 4 Konstruktion von Fixpunktmengen 13
4.1. Theoretische Grundlagen 13
4.2. Implementierung der Fixpunktoperatoren 13
4.3. Grammatik und Syntax der Fixpunktoperatoren 14
4.4. Beispiel: Auflösen einer Subkomponentenstruktur 15
Kapitel 5 Definition von Abbildungen 17
5.1. Nutzen und Eigenschaften 17
5.2. Implementierung des Abbildungsoperators 18
5.3. Verwendung von map in benamten Prädikaten 21
5.4. Beispiel 21
Kapitel 6 Beispiel: Tiefenkopie einer Komponentenstruktur 23
Kapitel 7 Änderungen und Ergänzungen des Interpreters 25
7.1. Nicht-strikte Auswertung der logischen Operatoren 25
7.2. Generierung von Zeugen und Gegenbeispielen 26
7.3. Framework für Funktionen 29
Kapitel 8 Erweiterungen und weitere Einsatzgebiete 33
8.1. Zusätzliche Funktionen 33
8.2. Benutzerdefinierte Funktionen 33
8.3. Menge erreichbarer Automatenzustände 33
Kapitel 9 Post-SEP Entwicklungen und aktueller Stand 34
9.1. Generierungsmethode und Evaluationsmethode 34
9.2. Less, Union, Intersect 34
9.3. Monotonie-Prüfung 35
9.4. Element Elimination 35
Anhang A: ODL Grammatik Version 0.6.3 40
Anhang B: ODL Ausdruck für die Tiefenkopie mit Ports und Kanälen 45
Anhang C: Notwendige Umformung von ODL Ausdrücken 47
Anhang D: Literatur 48

Kapitel 1 Einführung

1.1. Hintergrund: AutoFocus, Quest und ODL

AutoFocus ist ein am Lehrstuhl für Software and Systems Engineering Professor Broy der TU München entwickeltes CASE-Tool, zur Modellierung eingebetteter und verteilter Systeme. Es stützt sich dabei eine Reihe verschiedener Sichten, mit Hilfe derer die einzelnen Aspekte solcher System, wie deren Struktur, Nachrichtendatentypen und Kommunikationswege, beschrieben werden. AutoFocus legt dabei großes Gewicht auf die konsistente Spezifizikation und Verfizierbarkeit der erstellten Modelle.
Weiterführende Informationen zu AutoFocus finden sich im Internet auf [AutoFocus].

Mit dem Quest-Projekt wurde der Aspekt der Validation und Verifikation von Systemspezifikationen stärker hervorgehoben. Quest bietet Möglichkeiten AutoFocus-Modelle mit Hilfe formaler Techniken zu verfizieren. Quest basiert dabei auf einem Metamodell, dass die Elemente eines in AutoFocus erstellten Modells und deren Zusammenhänge beschreibt. Über diesem Metamodell wurde die in diesem Systementwicklungsprojekt weiterentwickelte, formale Sprache ODL definiert.
Näheres über Quest findet sich auf der Projekthomepage [Quest].

Die Operation Definition Language (ODL) ist eine auf der Prädikatenlogik erster Stufe basierende Sprache. Sie erlaubt, ähnlich wie SQL im Bereich der Datenbanken, die Formulierung von Abfragen und Transformationen auf Autofocus/Quest-Modellen.
Die theoretischen Grundlagen finden sich in [Schätz].

Das vorliegende Dokument stellt den Projektbericht zu meinem Systementwicklungsprojekt dar. Meine Arbeit baut dabei aus theoretischer Sicht auf den obigen Vorarbeiten und Projekten auf. Aus praktischer Sicht basiert sie zum einen auf der Bachelor-Thesis von David Pasch, der die erste Implementierung der ODL in Form eines Interpreters durchführte (vgl. [Pasch]).
Die zweite Grundlage wurde durch die Diplomarbeit von David Trachtenherz gelegt, der die ODL um einige Sprachkonstrukte, wie Mengen, Produkttypen, und andere, erweiterte (vgl. [Trachtenherz]). Der ODL-Interpreter erhielt durch ihn auch eine interaktive Schnittstelle für Benutzereingaben.

Meine Aufgabe bestand nun darin den Sprachumfang von ODL zu erweitern und dabei eine möglichst effiziente Auswertung sicher zu stellen. Die Einführung des Fixpunktoperators war dabei der erste Schritt zu einem konkreten ODL-Problem (vgl. Kapitel 2), wodurch sich zusätzliche Teilaufgaben ergaben. Dies waren die Einführung eines Operators zur Konstruktion von Abbildungen und eine Überarbeitung des der ODL zugrunde liegenden Typsystems.

1.2. Zielsetzung und Struktur des Projektberichts

Dieser Projektbericht verfolgt ausgehend von den Einzelaufgaben drei wesentliche Zielsetzungen:

- Verbindung zwischen den theoretischen Grundlagen und der Implementierung
- Exemplarische Darstellung der neuen ODL Sprachkonstrukte
- Dokumentation der Implementierung für die Weiterentwicklung

Diese Ziele finden sich im Aufbau des Berichts wieder. Im zweiten Kapitel gebe ich eine kurze Einführung in das AQuA-Framework sowie kurzes motivierendes Beispiel, nämlich das

Kopieren einer hierarchischen Struktur von Komponenten (Tiefenkopie). Im sechsten Kapitel wird dieses Beispiel dann exemplarisch in ODL umgesetzt.

Zuvor werde ich jedoch, in den Kapiteln drei mit fünf, die dafür notwendigen neuen ODL Sprachkonstrukte und Veränderungen erläutern. Dabei handelt es sich um das neue ODL-Typsystem (Kapitel 3), den Operator zur Konstruktion von Fixpunktmengen (Kapitel 4) und den Operator für die Definition von Abbildungen (Kapitel 5). In diesen Abschnitten werde ich jeweils die Brücke zwischen den theoretische Grundlagen und der Umsetzung schlagen. Abgerundet werden diese Kapitel jeweils durch kleine Beispiele, die Teillösungen des Tiefenkopieproblems darstellen.

Im Verlauf des Projekts ergaben sich zusätzlich einige kleinere Änderungen und Ergänzungen am bestehenden System. Diese werden im siebten Kapitel dokumentiert. Ich möchte den Leser hierauf besonders aufmerksam machen, da diese Anpassungen nicht nur Auswirkungen auf die weitere Entwicklung haben, sondern, wie im Falle der veränderten Auswertungsstrategie der prädikatenlogischen Quantoren, auch dem anwendungsorientierten Leser bekannt sein müssen. Zusammengefasst finden sich die benutzerrelevanten Änderungen im Anhang C.

Im achten Kapitel möchte ich abschließend einige Gedanken zu weiteren Entwicklungsmöglichkeiten geben.

Der Anhang dokumentiert die aktuelle ODL Grammatik, eine erweiterte Version der Tiefenkopie, die benutzerrelevanten Änderungen und die Literaturangaben.

Kapitel 2 Motivation

In AutoFocus wird die Architektur eines Software Systems als eine Ansammlung von miteinander kommunizierenden Komponenten modelliert. Jede Komponente hat Ein- und Ausgabeschnittstellen, die Ports, und kann über daran geknüpfte Kanäle mit anderen Komponenten Nachrichten austauschen. Um Systeme zusätzlich auf verschiedenen Detaillierungsebenen beschreiben zu können, haben Komponenten eine hierarchische Struktur, d.h. jede Komponente kann wiederum aus Subkomponenten zusammengesetzt sein.

Komponenten, Kanäle und Ports bilden zusammen einen Teil des Quest-Metamodells. Die ODL ist über diesem Metamodell definiert. In ODL existieren eine Reihe von Metatypen, für jedes Metamodellelement einer (z.B. Component für Komponenten, Channel für Kanäle). Ein Metatyp beschreibt dabei die Menge aller in einem konkreten Produktmodell vorkommenden Teile dieses Typ, z.B. die Menge aller Komponenten eines Geldautomaten.

Neben den konkreten Objekten, gibt es im Metamodell Beziehungen zwischen den einzelnen Metatypen. Diese Beziehungen sind aus Sicht von ODL als Relationen definiert. Besteht eine Komponente A beispielsweise aus zwei Komponenten B und C, so sind die beiden Tupel (A, B) und (A, C) in der Relation SubComponents enthalten. Die gleiche relationale Sicht findet sich bei Attributen der MetaTypen. Hat eine Komponente A den Namen X, so ist das Tupel (A, X) in der Relation Name enthalten.
Abbildung 1 zeigt den gerade beschriebenen Auschnitt des Metamodells. Die im Folgenden verwendeten Beispiele und ODL-Ausdrücke beziehen sich auf diesen Metamodellteil.

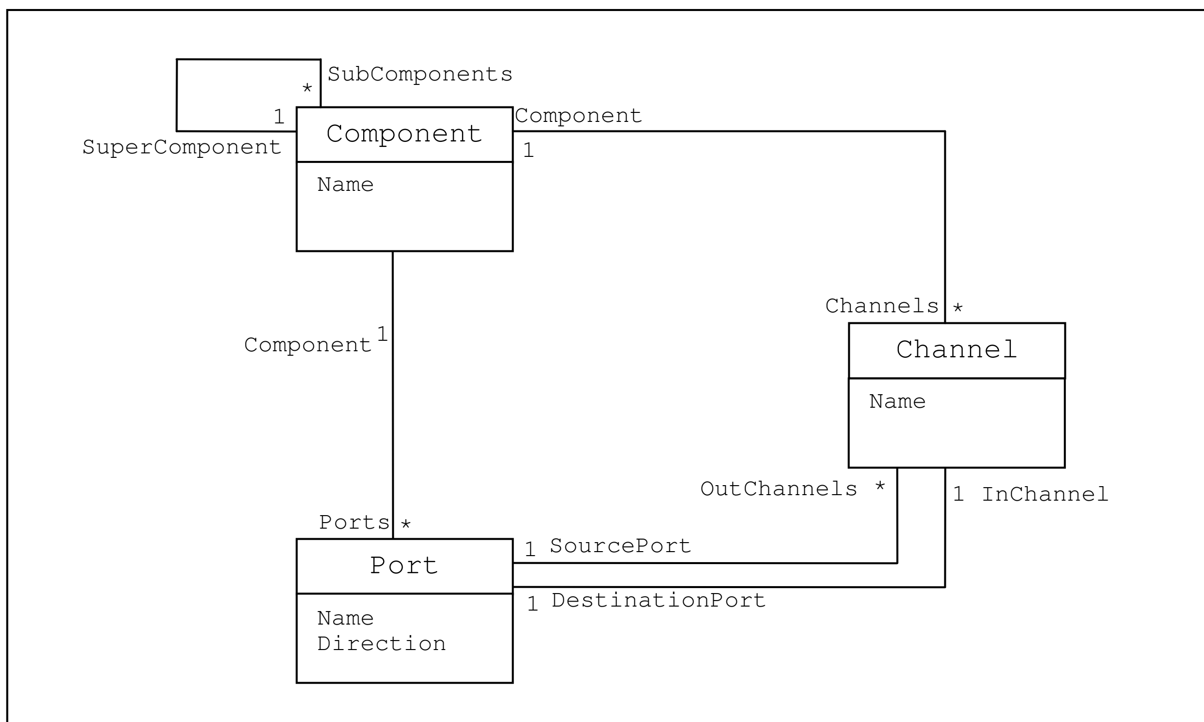


Abbildung 1: Quest Metamodell für Systemarchitekturen

Mit ODL können nun sowohl Abfragen wie auch Operationen, d.h. Ausdrücke, die Veränderungen am Produktmodell vornehmen, formuliert werden. Ein einfaches Beispiel wäre das Umbenennen einer Komponente. Hier ist der ODL Ausdruck, der vom Benutzer die Auswahl

einer Komponente und die Eingabe eines neuen Namens verlangt und dann die Umbenennung vornimmt:

```
context pair:(comp:Component, newName:String) .
result has Name(comp, newName)
```

Der Grund für die Durchführung meines Systementwicklungsprojekts war eine Aufgabenstellung, die mit den bisherigen Möglichkeiten in ODL nicht gelöst werden konnte, nämlich die Verarbeitung rekursiver Strukturen im Metamodell, wie sie zum Beispiel bei der Subkomponentenrelation bestehen. Ganz konkret fassten wir die Aufgabe so, dass man einen ODL Ausdruck formulieren können sollte, der eine Hierarchie von Komponenten und Subkomponenten kopiert.

Aus informatischer Sicht stehen im Prinzip zwei Ansätze zur Verfügung. Der operationale Ansatz geht dabei nach dem Divide-And-Conquer-Prinzip vor. Jede Komponente weist zunächst ihre Subkomponenten an sich zu kopieren und dupliziert sich mit den Ergebnissen dann selbst. Diese Vorgehensweise war für ODL aber nicht akzeptabel, da sie nicht mit der mengen- und relationenorientierten Sicht auf der das Metamodell beruht zusammenpasste.

Die umgesetzte Variante bedient sich der Fixpunkttheorie und wählt zur Lösung des Kopieproblems folgenden Weg. Mit Hilfe einer Fixpunktmengenkonstruktion werden die in der Hierarchie vorhandenen Komponenten eingesammelt. Für jede in der Fixpunktmenge enthaltene Komponente wird nun eine Kopie angelegt und in einer temporären Relation gespeichert. Damit wird die bestehende Subkomponentenstruktur auf die Kopien rekonstruiert.

Die Definition der dazu notwendigen Sprachkonstrukte, deren Implementierung und Test waren die Aufgaben meines Systementwicklungsprojektes. Darüberhinaus sind verschiedene Teile des ODL Interpreters neu strukturiert und überarbeitet worden. Meine Ausführungen werden sich dabei primär auf das hier vorgestellte Problem und den gezeigten Ausschnitt des Quest-Metamodells beschänken.

Kapitel 3 Neustrukturierung des Typsystems

Im Zuge der Einführung der beiden neuen ODL Typen für Fixpunktmengen (Kapitel 4) und Abbildungen (Kapitel 5), wurde das Typsystem auf eine neue Basis gestellt. Ziel war es dabei eine saubere Trennung der einzelnen Typklassen zu erreichen. Hierbei ergaben sich nicht nur umfangreiche Änderungen in dem Teil der ODL Grammatik, der sich auf die Typen bezieht; es mussten auch die Verwendung von Typkonstrukten in Zusammenhang mit den verschiedenen Quantoren (forall, exists, context) überarbeitet werden.

Ich werde im Folgenden die Grundlage des jetztigen Drei-Säulen-Typsystems darstellen und, im Hinblick auf die Weiterentwicklung, die neue Struktur in der Grammatik erläutern.

3.1. Begrenzte, unbegrenzte und post-universe Typen

Jeder Typ beschreibt eine Menge von Objekten, seinen Instanzen. Die einzelnen Typen lassen sich danach unterscheiden, ob sie eine begrenzte, d.h. endliche, Menge, z.B. die Menge der boolschen Werte (Boolean = {true, false}), oder eine unbegrenzte, d.h. unendliche Menge, z.B. die Menge aller ganzen Zahlen (Int = {..., -1, 0, 1, 2, ...}), beschreiben.

In ODL ist diese Unterscheidung wichtig, da die Quantoren forall und exists bei der Auswertung über alle Typinstanzen iterieren. Deshalb dürfen auch nur begrenzte Typen mit diesen Quantoren benutzt werden, da hierbei die Iteration garantiert in endlicher Zeit beendet ist und somit nur terminierende ODL Ausdrücke werden können.

Bei einem context-Quantor sind neben den begrenzten Typen die unbegrenzten sehr wohl erlaubt, da der Wert der durch diesen Quantor gebundenen Variable vom Benutzer bestimmt wird, also eine Reduktion der unendlich vielen Instanzen auf eine einzige gewährleistet ist.

Aus dieser mengeorientierten Sicht auf ODL Typen ergeben sich die ersten beiden Typklassen.

- Begrenzte Typen, bounded types:
  - Basistypen: Boolean, ModelElement (Component, Port, Channel, ...)
  - Erweiterte Typen: Set, Product, Restricted

- Unbegrenzte Typen, unbounded types:
  - Basistypen: String, Int
  - Erweiterte Typen: Set, Product, Restricted

Hierbei ist zu beachten, dass sich die erweiterten Typen im begrenzten Falle nur aus begrenzten Typen zusammensetzten dürfen. Erweiterte Typen im unbegrenzten Falle dürfen sehr wohl zu Teilen auch aus begrenzten Typen bestehen. Anders ausgedrückt: es bestimmt sich die Zugehörigkeit eines erweiterten Typs durch die verwendeten Basistypteile.

Beispiel eines begrenzten, erweiterten Typs:

```
set (c:Component, p:Port)
```

Jede Typinstanz beschreibt eine Menge von Zweiertupeln, die aus einer Komponente und einem Port bestehen.

Beispiel eines unbegrenzten erweiterten Typs:

```
{rt:(c:Component, s:String) | rt.c.Name = rt.s}
```

Jede Typinstanz ist ein Zweiertupel, bestehend aus einer Komponente und einer Zeichenkette, wobei Einschränkung fordert, dass die Zeichenkette dem Namen der Komponente entspricht.

Durch ODL Ausdrücke können nicht nur Eigenschaften des zu Grunde liegenden Produktmodells beschrieben werden, sondern auch Operationen, welche dieses Modell verändern (Sprachkonstrukte: result has, new). Somit kann ein ODL Ausdruck auch als eine Relation verstanden werden, nämlich einer Relation zwischen Produktmodellen vor der Ausführung (pre-universe) und den Produktmodellen nach der Ausführung (post-universe) des ODL Ausdrucks.

Aus dieser Sicht leitet man zwei weitere Klassen von ODL Typen ab, die pre-universe und die post-universe Typen. Zu ersteren gehören die oben bereits aufgeführten Modellelementtypen. Diese sind begrenzt, denn vor Ausführung eines Ausdruck steht eine endliches Produktmodell bereit, aus dem die konkreten Instanzen ermittelt werden können.
Die post-universe Typen beziehen sich ebenfalls auf die Modellelementtypen, jedoch umfassen die Instanzenmengen alle neu eingführten Modellelemente. Diese Menge ist daher prinzipiell unendlich, unterscheidet sich aber insofern von den unbegrenzten Typen, als dass die Semantik für die Verwendung in Verbindung mit Quantoren exakt definiert werden muss. Insbesondere muss hier eine Begrenzung auf eine endliche Menge von Typinstanzen stattfinden, um die Terminierung zu garantieren.

Da die pre-universe Typen bereits in der Klasse der begrenzten Typen enthalten ist, erhält man als dritte Typklasse die post-universe Typen.

- Post-universe Typen:
  - Basistypen: new ModelElement (new Component, new Port, new Channel, ...)
  - Erweiterte Typen: Map

3.2. Verwendung der Typklassen mit den Quantoren

Bevor ich das Drei-Säulen-Typsystem in der ODL Grammatik erläutere, möchte ich noch das Zusammenspiel der Typklassen mit den ODL Quantoren im Einzelnen darstellen.

Allquantor: Der Allquantor kann nur auf begrenzte Typen, angewandt werden. Im Falle der unbegrenzten und der post-universe Typen kann die Iteration nicht in endlicher Zeit durchgeführt werden, weshalb deren Verwendung ausgeschlossen wurde.

Existenzquantor: Zusätzlich zu begrenzten Typen kann der Existenzquantor auch auf post-universe Typen angewandt werden. Dies ist möglich, da für jeden post-universe Typ eine Semantik definiert wird, wie die Reduktion der unendlich vielen Typinstanzen auf eine endliche Anzahl geschieht. Der post-universe Basistyp new ModelElement liefert in Verbindung mit dem Existenzquantor genau ein neues Element. Das Verhalten des erweiterten Typs Map findet sich in Kapitel fünf.

Kontextquantor: Durch den Kontextquantor wird die Entscheidung, welche Typinstanz an die qunatifizierte Variable gebunden wird, in die Hände des Benutzers gelegt. Deshalb sind neben den begrenzten Typen auch die unbegrenzten erlaubt. Eine Verwendung der post-universe Typen wurde nicht als nützlich angesehen und deswegen nicht zugelassen.

New-Quantor: Der New-Quantor wurde abgeschafft und durch die existentielle Quantifizierung in Verbindung mit dem post-universe Basistyp new ersetzt.
Statt

```
new c:Component . <EXPR>
```

verwendet man nun

```
exists nc: new Component . <EXPR>
```

3.3. Das neue Typsystem in der ODL-Grammatik

3.3.1. Das Drei-Säulen-Typsystem

In ODL Ausdrücken tritt das Typsystem in Verbindung mit der Einführung einer gebundenen Variable durch einen Quantor auf. Die drei Säulen des Typsystems spiegeln sich daher in den drei Grammatikelementen zur Variablendefinition wider.

```
bounded_variable_definition = variable colon bounded_type;

unbounded_variable_definition = variable colon
  unbounded_type;

post_universe_variable_definition = variable colon
  post_universe_type;
```

Da das Typsystem habe ich es für die Darstellung an dieser Stelle gekürzt. Die vollständige Grammatik findet sich im Anhang A.

```
bounded_type = bounded_product_type |
 bounded_restricted_type_definition |
 bounded_set_type_definition |
 element defined_variable |
 element ( expression );

bounded_unary_type =
 bounded_basic_type |
 model_element_type;

bounded_basic_type = bool_type;
model_element_type = identifier;


unbounded_type =
 unbounded_product_type |
 unbounded_restricted_type_definition |
 unbounded_set_type_definition;

unbounded_unary_type = unbounded_basic_type;
```

```
unbounded_basic_type = int_type | string_type;


post_universe_type =
 post_universe_model_type |
 post_universe_map_type_definition;

post_universe_model_type = new model_element_type;
```

3.3.2. Die ODL-Quantoren

Ausgehend von der in im ersten Abschnitt beschriebenen Verbindung zwischen den Quantoren und den Typklassen, sowie den drei Einstiegspunkten über die Variablen-definitionen, wurde das neu organisierte Typsystem in die Grammatik für ODL Ausdrücke integriert.

```
unary_proposition =
 neg unary_proposition |
 forall_proposition    |
 exists_proposition    |
 context_proposition   |
 named_predicate_call  |
 term;

forall_proposition =
 forall bounded_variable_definition . unary_proposition;

exists_proposition =
 exists bounded_variable_definition . unary_proposition |
 exists post_universe_variable_definition .
     unary_proposition;

context_proposition =
 context bounded_variable_definition . unary_proposition |
 context unbounded_variable_definition . unary_proposition;
```

Man beachte, dass die in Einschränkungen und benamten Prädikaten verwendeten CCL Ausdrücke schon immer auf begrenzte Typen beschränkt waren. Hier fand daher lediglich eine Namensanpassung statt.

```
ccl_unary_proposition =
 neg ccl_unary_proposition |
 ccl_quantifier bounded_variable_definition .
     ccl_unary_proposition |
 named_predicate_call |
 ccl_term;

ccl_quantifier = forall | exists;
```

Kapitel 4 Konstruktion von Fixpunktmengen

Wie in dem motivierenden Beispiel gezeigt, erlaubt das AutoFocus/Quest-Metamodell Systeme als Zusammensetzung von Komponenten und Unterkomponenten zu beschreiben. Die somit mögliche hierarchische Gliederung konnte in der bisherigen Implementierung von ODL nicht hinreichend verarbeitet werden, da bisher kein Operator zur Verfügung stand, der mit einer rekursiven Struktur umgehen konnte. Das erste Ziel dieses System-entwicklungsprojekts war daher die Implementierung eines solchen Operators. Diese Implementierung fand noch auf Basis des ursprünglichen Typsystems statt. Sie wurde jedoch im Zuge der späteren Umarbeiten kaum verändert.

4.1. Theoretische Grundlagen

Um in ODL rekursive Ausdrücke formulieren zu können, wurde die Konstruktion von Fixpunktmengen als ein zusätzlicher Typ eingeführt. Dieser sollte dabei zunächst die schon bestehende Konstruktion von Mengen ausnutzen:

```
fix ident set_type.
```

Hierbei ist "fix" das Operatorschlüsselwort und "ident" ein Identifikator für die Fixpunktmenge, welcher in der nachfolgenden Mengenkonstruktion ("set_type") verwendet werden kann und somit die rekursiven Bezug erlaubt.

Die Auflösung der Komponentenstruktur ließe sich dann mittels einer eingeschränkten Mengenkonstruktion lösen:

```
fix FS set {c:Component | (c.Name="TopLevelComp" or
     exists sc:element FS. is SuperComponent(sc, c))}
```

Wie ich gleich erläutern werde, führte dieser Ansatz nicht zum gewünschten Ergebnis. Zuvor soll jedoch das algorithmische Prinzip des Fixpunktoperators kurz dargestellt werden.

Der Fixpunktoperator arbeitet nach folgendem Prinzip:

- Initialisierung: Dem Identifikator wird die leere Menge zugewiesen.
- Iteration: Nun wird die Definition der Mengenkonstruktion ausgewertet und das Ergebnis mit der aktuellen Identifikatorbelegung verglichen. Ist die konstruierte Menge und die bisher dem Identifikator zugewiesene Menge gleich, so wurden keine neuen Elemente mehr hinzugefügt und der Fixpunktoperator liefert diese Menge als Ergebnis zurück. Sollten jedoch neue Elemente hinzugekommen sein, wird der Identifikator mit der neu berechneten Menge belegt und ein weiterer Iterationsschritt ausgeführt.

4.2. Implementierung der Fixpunktoperatoren

Der Versuch den Fixpunktoperator, wie im vorangegangenen Abschnitt erklärt, zu programmieren, also unter Verwendung der schon bestehenden Mengenkonstruktion, lieferte kein akzeptables Ergebnis.

Es wäre für den Fixpunktoperator ein spezieller, zweiter Analyselauf des Interpreters notwendig gewesen, da der Typ der Fixpunktmenge erst feststeht, sobald die Mengenkonstruktion analysiert ist. Diese aber kann nur typisiert werden, wenn der der Typ der Fixpunktmenge bekannt ist. Die Implementierung eines zweiten Analyselaufs hätte eine Vielzahl von kleinen

Änderungen in den ODL Sprachkonstrukten zur Folge gehabt und somit das Risiko ungewollter Seiteneffekte erhöht, den Quellcode unnötig verkompliziert und die Ausführungszeit von ODL Abfragen verschlechtert.

Die Implementierung des Fixpunktoperators führt daher Mengenkonstruktion nach folgendem Verfahren selbst durch:

- **Initialisierung**: Die leere Menge wird als initiale Fixpunktmenge an den Identifikator zugewiesen.
- **Innere Iteration**: Die innere Iteration übernimmt die Aufgabe der Mengenkonstruktion. Es werden die Elemente des zugrunde liegenden Basistyps (`Component` im obigen Beispiel) nacheinander aufgezählt und jeweils getestet, ob sie die in der `with`-Klausel angegebene Bedingung erfüllen. Ist dies der Fall, so wird das Element der Fixpunktmenge hinzugefügt. Sollte das Element schon in der Fixpunktmenge sein, so wird der Test nicht noch einmal durchgeführt.
- **Äußere Iteration**: Die äußere Iteration übernimmt die Aufgabe des Mengevergleichs. Da die Konstruktion der Fixpunktmenge durch den Fixpunktoperator selbst geschieht, wird der Mengenvergleich nicht explizit durchgeführt. Statt dessen informiert die innere Iteration die äußere Iteration mittels einer lokalen, boolschen Indikatorvariable, falls sie ein Element hinzugefügt hat. Solange die Indikatorvariable mit true belegt ist, wird sie auf false gesetzt und die innere Iteration begonnen. Dies verhindert unnötiges Kopieren der Fixpunktmenge vor und das Vergleichen nach der inneren Iteration. Die äußere Iteration ist beendet, sobald die Indikatorvariable nach Ausführung der inneren Iteration noch mit false belegt ist, also keine neuen Elemente hinzugefügt wurden.

Das dargestellte Verfahren beseitigte nicht nur die Notwendigkeit eines zweiten Analyselaufs, sondern hatte noch den weiteren Vorteil, dass die Konstruktion der Fixpunktmenge unnötige Tests der Basistypelemente verhindert und daher wesentlich effizienter ist. In Abschnitt vier gehe ich darauf anhand des Beispiels noch genauer ein.

Wie die Überschrift dieses Abschnittes vermuten lässt, gibt es in der aktuellen Interpreterversion mehrere Fixpunktoperatoren. Der `lfp`-Operator berechnet den kleinsten Fixpunkt (`lfp` = least fix point) und arbeitet wie gerade beschrieben. Der `gfp`-Operator berechnet den größten Fixpunkt (`gfp` = greatest fix point) und arbeitet nach einem analogen Verfahren. Er beginnt, statt mit der leeren Menge, mit der Menge aller Basistypelemente und entfernt die Elemente, welche die Bedingung (`with`-Klausel) nicht erfüllen.

Abschließend muss ich noch auf eine Besonderheit der Fixpunktmengenkonstruktion eingehen. Wenn das durch die with-Klausel festgelegte Prädikat nicht monoton ist, so liefert die Konstruktion einen der kleinen bzw. großen Fixpunkte, jedoch nicht den kleinsten bzw. größten, da dieser nicht existiert. Eine automatische Erkennung nicht-monotoner Prädikate und eine entsprechende Benutzerunterstützung ist Gegenstand weiterer Entwicklungsarbeit.

4.3. Grammatik und Syntax der Fixpunktoperatoren

In diesem Abschnitt stelle ich kurz die Syntax der Fixpunktoperatoren durch die Definition ihrer ODL Grammatik dar. Es handelt sich bei dem Fixpunktmengetyp um eine Erweiterung des schon bestehenden Mengentyps `set`.

```
fixed_point_set_type_definition ::=
  lfp <identifier1> set <identifier2> : <type>
```

```
      with ( <proposition> ) |
    gfp <identifier1> set <identifier2> : <type>
      with ( <proposition> )
```

Hierbei ist `<identifier1>` der Identifikator für die Fixpunktmenge, `<identifier2>` der Identifikator der Elemente des Basistyps `<type>` und `<proposition>` die Bedingung zur Konstruktion der Fixpunktmenge.

4.4. Beispiel: Auflösen einer Subkomponentenstruktur

Abschließend möchte ich ein Beispiel für die Verwendung des Fixpunktoperators geben. Es handelt sich dabei gleichzeitig um den ersten Schritt zur Lösung des Tiefenkopieproblems, nämlich das Auflösen einer Struktur von Komponenten.

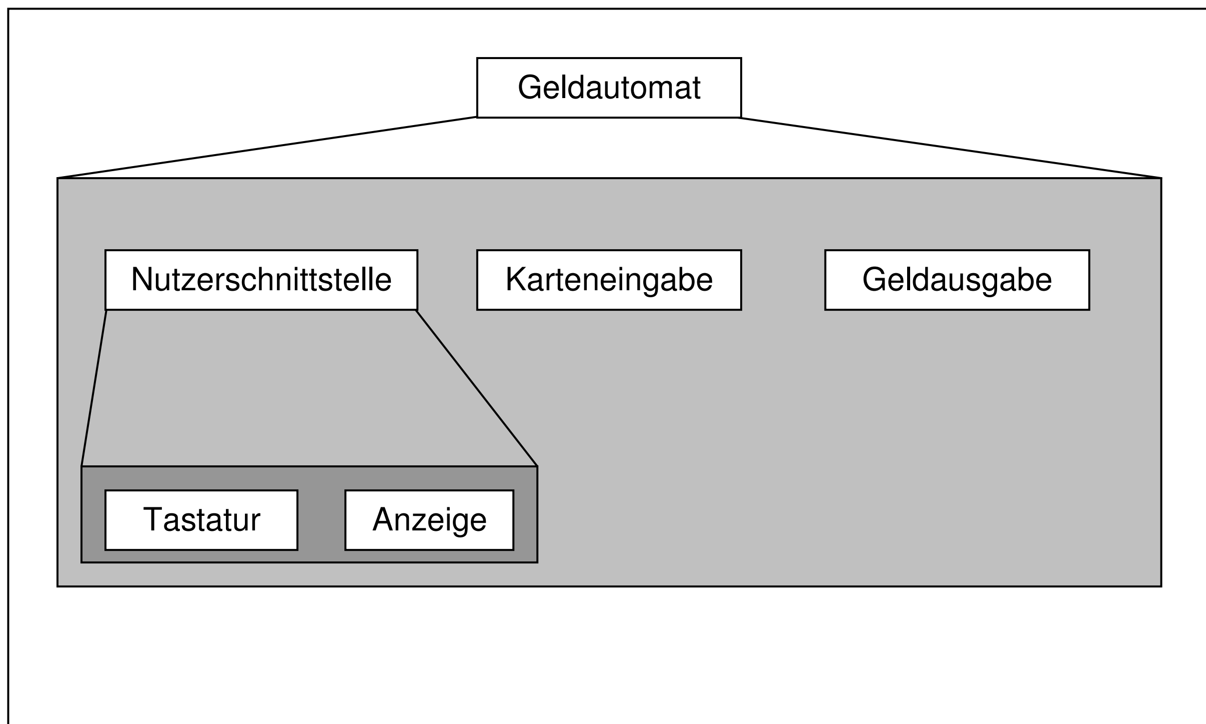


Abbildung 2 : Komponentenstruktur eines Geldautomaten

Abbildung 2 zeigt beispielhaft die Struktur eines Geldautomaten. Der Fixpunktoperator muss nun durch die Konstruktionsbedingung gesagt bekommen, dass in der Fixpunktmenge das oberste Element, also der Geldautomat, und alle Unterkomponenten enthalten sein sollen.
Die erste Aussage lässt sich durch einen einfachen Namensvergleich formulieren. Für den zweiten Teil verwenden wir die Relation SubComponents, die eine Komponente mit ihren Unterkomponenten in Beziehung setzt und nutzen die rekursive Eigenschaft des Fixpunktoperators.
Es ergibt sich folgender ODL Ausdruck:

```
exists compSet : lfp FS set c:Component with
  (c.Name = "Geldautomat" or
   exists e:element FS . is SubComponents(e,c)) . true
```

Wie man sieht, wird in der `with`-Klausel gefordert, dass die in der Menge `FS` (bzw. später `compSet`) enthaltenen Komponenten entweder den Namen "Geldautomat" haben oder in der Menge `FS` bereits eine Oberkomponente der aktuell betrachteten enthalten ist.

Wieviele Auswertungsschritte, d.h. äußere Iterationen notwendig sind, hängt davon ab, in welcher Reihenfolge die Komponenten durch den Basistyp "Component" aufgezählt werden. Ich verdeutliche dies durch die Betrachtung des schlechtesten und des besten Falles.

Schlechtester Fall:

Tastatur, Anzeige, Nutzerschnittstelle, Karteneingabe, Geldausgabe, Geldautomat

Bei dieser Reihenfolge benötigt der Fixpunktoperator vier äußere Iterationen.

1. Iteration: "Geldautomat" wird in die Fixpunktmenge aufgenommen.
2. Iteration: "Nutzerschnittstelle, Karteneingabe, Geldausgabe" kommen hinzu.
3. Iteration: "Tastatur, Anzeige" werden hinzugefügt.
4. Iteration: keine neuen Elemente aufgenommen, somit ist die Fixpunktmenge berechnet.

Bester Fall:

Geldautomat, Geldausgabe, Karteneingabe, Nutzerschnittstelle, Anzeige, Tastatur

Im besten Fall benötigt der Operator nur zwei äußere Iterationen, nämlich dann, wenn die Komponenten in umgekehrter Reihenfolge zu oben aufgezählt werden. In der ersten äußeren Iteration werden bereits alle Komponenten in die Fixpunktmenge aufgenommen:
1. innere Iteration: Aufnahme von Geldautomat
2. innere Iteration: Aufnahme von Geldausgabe, denn Geldautomat ist bereits enthalten
...
4. innere Iteration: Nutzerschnittstelle wird hinzugefügt
5. innere Iteration: Aufnahme von Anzeige, da Nutzerschnittstelle bereits enthalten
...

Die zweite äußere Iteration ist notwendig um sicherzustellen, dass keine neuen Elemente mehr hinzugefügt wurden. Diese nimmt allerdings auch nicht sonderlich viel Zeit in Anspruch, da bereits enthaltene Elemente nicht noch einmal getestet werden.

Mit Hilfe der interaktiven Schnittstelle von [Trachtenherz] könnte man die Auswahl der obersten Komponente dem Benutzer überlassen:

```
context userComp:Component . exists compSet : lfp FS set
  c:Component with (c = userComp or
    exists e:element FS . is SubComponents(e,c)) . true
```

Kapitel 5 Definition von Abbildungen

In den vorhergehenden Kapiteln haben ich gezeigt, wie man mit Hilfe des Fixpunktoperators Komponentenhierarchien auflösen kann und ich habe das neue Typsystem erläutert. In diesem Kapitel werde ich den komplexen, post-universe Typ `map` besprechen, der zur Lösung des Tiefenkopieproblems in der hier gewählten Variante unverzichtbar ist.

5.1. Nutzen und Eigenschaften

Hat man mit Hilfe des Fixpunktoperators alle zu kopierenden Modellelemente zusammengesammelt, man kann die Subkomponentenstruktur aus dieser Menge nicht mehr rekonstruieren. Um diese Struktur wieder herzustellen, benötigen wir eine Abbildung bzw. Menge von Tupeln, die eine Beziehung von bestehenden Elementen zu neu erzeugten Elementen herstellt. Dies wurde durch die Einführung eines Typs für Abbildungen, den `map`-Typ, gelöst.

Dieser Typ stellt eine Zuordnung von Elementen eines gegebenen Quelltyps zu Elementen eines Zieltyps dar. Folgender Ausdruck würde beispielsweise eine Zuordnung von Komponenten zu den boolschen Werten ergeben:

```
map src:Component to dst:Boolean
```

Aus mathematischer Sicht liefert dieser Ausdruck nacheinander alle rechtseindeutigen, rechtrstotalen Abbildungen zwischen den beiden gegebenen Typen. Technisch gesehen ist eine einzelne Abbildung eine Menge von Zweiertupeln (`Component`, `Boolean`), deren Anzahl der Anzahl von Elementen des Quelltyps, also `Component`, entspricht. Es gibt also eine Beziehung von `map` zu den schon vorhandenen Mengen von Produkttypen in ODL, wobei `map` sich aber auf totale Funktionen beschränkt. Diese Beziehung wird bei der Verwendung von Abbildungen als Parameter in benamten Prädikaten noch wichtig (vgl. Abschnitt 5.3).

Durch Einbettung des map-Typs in einen Existenzquantor, kann man logische Eigenschaften der Abbildung fordern:

```
exists myMap: map comp:Component to hasSubComp:Boolean .
  forall elem: element myMap .
    (exists c:Component .
       is SubComponents(elem.comp, c) equiv
       (elem.hasSubComp=true)
    )
```

Dieser Ausdruck liefert genau die Abbildung, welche jeder existierenden Komponente den boolschen Wert `true` zuordnet, falls diese mindestens eine Subkomponente hat und `false`, wenn dies nicht der Fall ist.

Das eben gezeigte Beispiel verwendet map als einen begrenzten Typen, da sowohl Quelltyp als auch Zieltyp in die Klasse der begrenzten Typen fallen (vgl. Kapitel 3). `map` wurde aber primär zur Definition von Mengen neu erzeugter Modellelemente, bzw. Abbildungen von bestehenden Modellelementen auf neue, eingeführt. Da der Zieltyp somit in die Klasse der post-universe Typen fällt, muss für map ein spezielles Verhalten festgelegt werden, welches garantiert, dass ODL-Ausdrücke terminieren.

Um die endliche Laufzeit zu garantieren, wurde zunächst der Quelltyp der Abbildung auf begrenzte Typen beschränkt. Als Zieltyp lassen wir über die begrenzten Typen hinaus auch den post-universe Typ new zu.

`new` erzeugte, in Verbindung mit einem Existenzquantor, genau eines neuen Element. Im Kontext eines map-Typs, wurde die Erzeugung auf höchstens N neue Modellelemente begrenzt, wobei N die Anzahl der Quelltypinstanzen, und somit endlich, ist. Dies ist korrekt, da map eine Funktion beschreibt und nicht eine allgemeine Relation. Es werden daher also höchstens N Elemente benötigt um alle Abbildungen auf neu erzeugte Modellelemente konstruieren zu können.

5.2. Implementierung des Abbildungsoperators

5.2.1. Definition und Auswertungsalgorithmus

Ein map-Typ wird durch vier Angaben definiert:

```
map <srcIdent>: <srcType> to <dstIdent>:<dstType>
```

Dies sind zum einen der Quellidentifikator (srcIdent) und der Quelltyp (srcType), zum anderen der Zielidentifikator (dstIdent) und der Zieltyp (dstType). Die Klasse des Zieltyps (begrenzt oder post-universe) entscheidet dabei über die Klasse des map-Typs.

Wie bei jedem ODL-Typ werden die konkreten Instanzen durch einen java.util.Iterator bereitgestellt, welcher mittels der `instances(Assignment)`-Methode vom Typobjekt geholt werden kann.
Die eigentliche Arbeit, das Erzeugen der Abbildungen, geschieht durch den MapTypeIterator. Dieser erhält bei seiner Instanziierung neben einer Referenz auf ein Objekt der Java-Klasse MetaProductType, welches durch die eben erklärten vier Angaben definiert ist, ein Feld mit den Instanzen des Quelltyps und einen Iterator über die Instanzen des Zieltyps.

Als lokale Hilfsdaten benutzt der MapTypeIterator ein Objektfeld, den Cache, in dem die durch den Zieltypiterator erhaltenen Objekte gespeichert sind. Um zu wissen welche Abbildungen schon generiert wurden, benutzt der MapTypeIterator ein lokales Feld von ganzen Zahlen, das für jede Quelltypinstanz einen Eintrag besitzt. Die an der i-ten Stelle gespeicherte Zahl stellt den Index in den Cache dar, dem das i-te Quellelement in der aktuellen Abbildung zugeordnet ist.

Anfänglich ist der Cache leer, d.h. die Cachegröße ist null, und das Indexfeld mit Nullen initialisiert. Von dieser Ausgangssituation lassen sich nun alle Abbildungen sukzessive, mittels des folgenden Algorithmus, generieren:

1 > Betrachte des letzte Quellelement LAST
2 > Ist LAST dem letzten Cacheelement zugeordnet (index >= cachesize-1)?
3 > Nein. Ordne LAST dem nächsten Cacheelement zu. FERTIG.
4 > Hat der Zieliterator ein weiteres Element ?
5 > Ja, dann vergrößere den Cache mit diesem Element. Ordne LAST dem nächsten Cacheelement zu. FERTIG.
6 > Da der Cache bereits vollständig gefüllt ist, suche im Indexfeld von hinten ein Quellelement, das noch nicht dem letzten Cacheelement zugeordnet ist.

7 > Gibt es ein solches Element?
8 > Ja, dann ordne dieses dem Nächsten zu; alle nachfolgenden beginnen wieder bei Null.
FERTIG.
9 > Nein, somit ist der Cache komplett und alle Quellelemente sind dem letzten
Cacheelement zugeordnet. Es wurden alle Abbildungen generiert.

Um den Ablauf zu verdeutlichen, soll kurz der folgende Ausdruck genauer betrachtet werden, der alle vier Abbildungen von Boolean nach Boolean ermittelt:

```
forall m:map b1:Boolean to b2:Boolean . true
```

Initialisierung
Quelltypinstanzen = {true, false}
Cache = { }
cachesize = 0;
map = {0, -1} *// die -1 hat implementierungstechnische Gründe*
dstIter = {true, false} *// übrige Zieliteratorelemente*

1.Abbildung:
Da "map[1] >= cachesize - 1" und noch übrige Elemente vorhanden wird Zeile 5 ausgeführt, der Cache also um ein Element erweitert.

Cache = {true}; cachesize = 1; map = {0, 0}; dstIter = {false}
Resultat: {(true, true), (false, true)}

2.Abbildung:
Da "map[1] >= cachesize - 1" und noch übrige Elemente vorhanden wird Zeile 5 ausgeführt, der Cache also um ein Element erweitert.

Cache = {true, false}; cachesize = 2; map = {0, 1}; dstIter = { }
Resultat: {(true, true), (false, false)}

3.Abbildung:
Da "map[1] >= cachesize - 1" aber keine Elemente mehr, jedoch "map[0] < cachesize - 1", wird Zeile 8 ausgeführt, d.h. in diesem Fall wird das erste Quellelement dem zweiten.Zielelement und das zweite Quellelement wieder dem ersten Zielelement zugeordnet.

Cache = {true, false}; cachesize = 2; map = {1, 0}; dstIter = { }
Resultat: {(true, false), (false, true)}

4.Abbildung:
Da "map[1] < cachesize - 1" wird Zeile 3 ausgeführt, also das letzte Quellelement dem nächsten Zielelement zugeordnet.

Cache = {true, false}; cachesize = 2; map = {1, 1}; dstIter = { }
Resultat: {(true, false), (false, false)}

5.Abbildung:

Da "map[1] >= cachesize - 1", keine weiteren Zielelemente mehr übrig und "map[0] >= cachesize - 1" wird Zeile 9 ausgeführt, d.h. derAlgorithmus hat alle möglichen Abbildungen generiert.
Resultat: Keine weiteren Abbildungen mehr.

Dieser Algorithmus generiert alle Abildungen. Im schlechtesten Fall sind dies, wie oben bereits erwähnt, "Anzahl Zielelemente" hoch "Anzahl Quellelemente" Abbildungen.

Ein erster Test, der versuchte für lediglich elf Komponenten eine injektive Abbildung auf elf neu erzeugte Elemente zu finden, zeigte dass der gerade beschriebene Algorithmus nicht akzeptabel war. Unter der Annahme ein Abbildung in einer Millisekunde generieren und testen zu können, wäre ein Zeitbedarf von ca. 11 Stunden notwendig, bis die erste injektive Abbildung gefunden würde.

Daher war es notwendig eine Verbesserung des Suchverfahrens zu schaffen, welche den Fall, zu einer gegebenen Menge von Elementen je ein neues Element zu erzeugen, in einem Schritt löst. Ich erläutere dieses Verfahren im dritten Abschnitt.

5.2.2. Verbesserung der Auswertungsalgorithmus

Die Grundidee des im letzten Abschnitt vorgestellten Algorithmus, ist alle Abbildungen angefangen bei dem Indexfeld {0, 0, ..., 0} bis zum Indexfeld {N, N, ..., N} durchzuprobieren, wobei N die Anzahl der Zielelemente ist.

Die Verbesserung besteht nun darin, diesen Suchbereich in einer im Hinblick auf die gewünschte injektive Abbildung günstigeren Weise zu durchsuchen. Es wird daher mit der ersten möglichen injektiven Abbildung begonnen. Zunächst wird versucht soviele Zielelemente in den Cache zu laden, wie Quellelemente vorhanden sind.

Gelingt dies nicht, so gibt es weniger Ziel- als Quellelemente, folglich existiert gar keine injektive Abbildung und es wird der bisherige Algorithmus benutzt. Der bereits vorgefüllte Cache stellt sich hierbei zusätzlich als vorteilhaft dar, da wir auf die Prüfung weiterer Zielelemente (siehe Algorithmus, Zeilen 4 und 5) verzichten können.

Konnten ausreichend Elemente in den Cache geladen werden, so wird mit der ersten injektiven Abbildung (0, 1, 2, ..., M-1) ein Versuch gestartet. M ist hierbei die Anzahl der Quellelemente.
Ist diese injektive Abbildung nicht akzeptiert worden, so beginnen wird wieder das bisherige Verfahren verwendet, also bei (0, ..., 0) fortgesetzt.

Dieses Verfahren wurde gewählt, da die Intention des Abbildungsoperators in erster Linie die Generierung von injektiven Abbildungen auf neue Elemente war. Die Verwendung zur Definition anderer Abbildungen sollte genau überlegt werden, da hierbei, wie bereits erwähnt, extrem lange Ausführungszeiten entstehen können. Im Kapitel neun gehe ich kurz auf einen Vorschlag zur Definition von Sammlungen (`collect`) ein, der ähnliche Definitionen wie Abbildungen erlaubt, jedoch für dieses Projekt zu komplex war.

5.2.3. Änderungen am Typsystem

Zur Vollständigkeit der Ausführungen über die Implementierung, gebe ich in diesem Abschnitt die Grammatikerweiterungen an.

```
bounded_map_type_definition =
  map <from_variable> : <bounded_type>
     to <to_variable> : <bounded_type>;

post_universe_map_type_definition =
  map <from_variable> : <bounded_type>
     to <to_variable> : <post_universe_model_type>;
```

Ein map-Typ ist genau wie der Fixpunktmengentyp eine spezielle Form des set-Typs, nämlich eine Menge von Zweiertupeln (<from_variable>:<bounded_type>, to_variable>:<...>).

5.3. Verwendung von map in benamten Prädikaten

Bei der Verwendung des Abbildungstyps map als Parameter für benamte Prädikate ist Folgendes zu beachten:

- Setzt sich die Abbildung nur aus schon bestehenden Modellelementen und Basistypen zusammen, d.h. enthält sie kein new im Zieltyp, so kann diese Maptypdefinition prinzipiell auch als Parametertyp in benamten Prädikaten verwendet werden.
- Werden durch die verwendete Abbildung neue Modellelemente eingeführt (Verwendung von new), kann diese Maptypdefinition nicht als Parametertyp verwendet werden, sondern muss in einen passenden Typ umgeformt werden. Diese Umformung funktioniert auch für den ersten Fall, weshalb ich sie grundsätzlich bei der Verwendung von map in benamten Prädikaten empfehle.

Wie schon erwähnt ist eine Abbildung eine Menge von Zweiertupeln. Im Kontext der Definition eines benamten Prädikats wird diese Tatsache ausgenutzt und ein Parameter vom Typ map in den entsprechenden Typ umgeformt.

Betrachtet man die Abbildung

```
map aVar : aType to bVar : [new] bType
```

so ergibt sich der entsprechende Parametertyp zu

```
set (aVar : aType, bVar : bType).
```

und zwar unabhängig, ob `new` im Maptyp verwendet wurde oder nicht. Dies funktioniert, da die Typangabe im Kontext des benamten Prädikats nicht zur Iteration über die Typinstanzen verwendet wird, sondern nur zur Überprüfung der Argumenttypen bei der späteren Verwendung des Prädikats. Besonders möchte ich hervorheben, dass die Selektoren (oben: aVar, bVar) in der Definition des Prädikats und in der später als Parameter verwendeten Maptypvariable den gleichen Namen haben müssen.

5.4. Beispiel

Die zur Lösung des Tiefenkopieproblems benötigte Abbildung, ist durch folgenden ODL-Ausdruck definiert. Die Bedingung des Existenzquantors entspricht der Injektivitätsbedin-

gung für Abbildungen, d.h. für jedes alte soll eindeutig auf genau ein neues Element abgebildet werden.

```
exists myCopyMap: map old:Component to copy:new Component .
 (forall e1: element myCopyMap .
    forall e2: element myCopyMap .
      (e1.copy = e2.copy implies e1.old = e2.old)
 )
```

Unter Verwendung eines benamten Prädikats, wie im vorherigen Abschnitt erläutert, ergibt sich folgender ODL Ausdruck:

```
injective(myMap : set (old:Component, copy:Component)) :=
  (forall e1: element myMap .
    forall e2: element myMap .
      (e1.copy = e2.copy implies e1.old = e2.old)
 )
exists myCopyMap: map old:Component to copy:new Component .
  call injective(myCopyMap).
```

Kapitel 6 Beispiel: Tiefenkopie einer Komponentenstruktur

In diesem Kapitel werden wir die zuvor eingeführten ODL Sprachkonstrukte verwenden um eine Struktur von Komponenten und Unterkomponenten zu kopieren. Ich beschränke mich hier auf die Kopie der Komponenten. Im Anhang findet sich der ODL Ausdruck, der zusätzlich die Ports und die Kanäle kopiert.

Zunächst nutzen wir die interaktive Schnittstelle und lassen den Benutzer das Projekt und die zu kopierende Komponente wählen:

```
context proj:Project . context userComp:Component .
```

Nun benutzen wir den Fixpunktoperator um ausgehend von der gewählten Komponente rekursiv alle Subkomponenten in einer Menge aufzusammeln:

```
exists compSet: lfp cs set csc:Component
  with (csc = userComp or
  exists cse:element cs . is SubComponents(cse, csc)
) .
```

Als nächstes erzeugen wir für jede aufgesammelte Komponente mit Hilfe des Mapoperators ein neues Modellelement. Durch die Injektivitätsforderung erhalten wir eine Abbildung, die jeder zu kopierenden Komponente genau eine neu erzeugte Komponente zuordnet:

```
exists compMap: map orig:element compSet to
  copy:new Component . (
    (forall e1:element compMap .
       forall e2:element compMap . (
         e1.copy = e2.copy implies e1.orig = e2.orig
       )
    ) and
```

Jetzt übernehmen wir den Namen der Originalkomponenten (Relation `Name`) und übertragen die Subkomponentenbeziehung (Relation `SubComponents`):

```
forall mapElem1:element compMap . (
  result has Name(mapElem1.copy, mapElem1.orig.Name)and
  forall mapElem2: element compMap . (
    is SubComponents(mapElem1.orig,mapElem2.orig) implies
    has SubComponents(mapElem1.copy, mapElem2.copy)
  )
) and
```

Schließlich fügen wir die neu erzeugte Komponentenstruktur in das Projekt ein, indem wir zunächst die vom Benutzer gewählte Komponente in unserer Abbildung suchen und die damit assoziierte Komponente mit dem Projekt über die Relation `Components` verknüpfen:

```
exists topLevelMapElem: element compMap . (
  topLevelMapElem.orig = userComp and
  result has Components(proj, topLevelMapElem.copy)
```

```
    )
)
```

Zusammengesetzt ergibt sich folgender ODL Ausdruck:

```
context proj:Project . context userComp:Component .
exists compSet: lfp cs set csc:Component with (
  csc = userComp or exists cse:element cs .
    is SubComponents(cse, csc)
) .
exists compMap: map orig:element compSet to
  copy:new Component . (
    (forall e1:element compMap .
        forall e2:element compMap . (
            e1.copy = e2.copy implies e1.orig = e2.orig
        )
    ) and
    forall mapElem1:element compMap . (
      result has Name(mapElem1.copy,mapElem1.orig.Name) and
      forall mapElem2: element compMap . (
        is SubComponents(mapElem1.orig,mapElem2.orig) implies
        result has SubComponents(mapElem1.copy,mapElem2.copy)
      )
    ) and
    exists topLevelMapElem: element compMap . (
         (topLevelMapElem.orig = userComp) and
         (result has Components(proj, topLevelMapElem.copy)
    )
  )
)
```

Kapitel 7 Änderungen und Ergänzungen des Interpreters

Im Verlauf des Projektes ergaben sich einige Änderungen und Ergänzungen des bestehenden ODL Interpreters, die nicht explizit Teil des Systementwicklungsprojektes waren. Das Framework für Funktionen baute ich beispielsweise nur, weil ich es für sinnvoll hielt eine Funktion zum Aneinanderhängen zweier Zeichenketten zur Verfügung zu haben. Die Änderungen an der Implementierung der ODL zugrunde liegenden Aussagen- und Prädikatenlogik wurden zur Steigerung der Effizienz durchgeführt. Diese Massnahmen führten im Weiteren zur Veränderung der Evaluationskomponente und auch des ODL Editors. Im Folgenden werde ich die vorgenommenen Änderungen darstellen.

7.1. Nicht-strikte Auswertung der logischen Operatoren

Die Effizienzsteigerung im Bereich des logischen Basissystems der ODL wurden durch den Übergang von einer strikten Evaluierung zur nicht-strikten erzielt. Dieser Übergang wurde sowohl für das auf den aussagenlogischen Operatoren basierende als auch für das auf den prädikatenlogischen Quantoren basierende Teilsystem durchgeführt.

7.1.1. Optimierte, nicht-strikte Auswertung von aussagenlogischen Operatoren

Die bisherige Implementierung stützte sich die Auswertung der logischen Operatoren or, implies und equiv auf die Operatoren and und neg ab. Dies bedeutet, dass im Evaluationsbaum zwar ein Knoten für die zuerst genannten Konstrukte vorhanden ist, diese jedoch nur als Kapsel für entsprechende Teilbäume dienen. Am Beispiel der Implikation lässt sich dies sehr schön demonstrieren. Statt "A implies B" direkt auszuwerten, wird die Arbeit zunächst an den Disjunktionsoperator ("neg A or B") und dann an den Konjunktionsoperator weitergegeben ("neg(neg neg A and neg B)").

Diese Variante der Implementierung hat zwar den Vorteil einen sehr einfachen Quellcode zu besitzen, führt in der Ausführung jedoch zu unnötigen zusätzlichen Objekten und Methodenaufrufen, was eine schlechtere Ausführungsgeschwindigkeit zur Folge hat. Deshalb habe ich die Delegation der Arbeit, im Falle von implies und or, aufgehoben und durch eine direkte Auswertung ersetzt.
Lediglich der Operator equiv deligiert die Arbeit noch, was seine Begründung in der im anschließenden Abschnitt zwei ausgeführten Generierung von Zeugen und Gegenbeispielen findet. Das Ergebnis der Äquivalenz kann zwar durch einen Vergleich der Ergebnisse der Teilausdrücke errechnet werden, die Ermittlung der Zeugen und der Gegenbeispiele gestaltet sich jedoch wesentlich komplizierter. Durch die Delegation wird in diesem speziellen Fall ein Stück Laufzeiteffizienz der Verständlichkeit des Quellcodes geopfert.

Um die Effizienz der Auswertung logischer Operatoren weiter zu erhöhen, wurden diese von der strikten Variante in die nicht-strikte umgewandelt. Dies bedeutet, dass beispielsweise im Falle des logischen Und die Auswertung mit false abgebrochen wird, wenn schon der erste Operand false als Ergebnis lieferte. Hierdurch konnte wiederum die Zahl der notwendigen Methodenaufrufe reduziert werden.

7.1.2. Nicht-striktes Verhalten der prädikatenlogischen Quantoren

Im Grunde unterscheidet sich die Änderung des Auswertungsverfahrens der Quantoren nicht von der der logischen Operatoren. Der Existenzquantor bricht die Auswertung ab, sobald er das erste Element des Basistyps gefunden hat, welches das Bedingungsprädikat erfüllt, und liefert dieses Element samt dem Ergebnis true zurück. Der Universalquantor arbeitet analog: er bricht ab, sobald ein Element das Bedingungsprädikat nicht erfüllt. Beide Quantoren werden also nicht strikt ausgewertet, was unnötiges Evaluierungen vermeidet.

An dieser Stelle muss ich aber einen wichtigen Unterschied zur bisherigen Implementierung verdeutlichen. Bisher war es mit Hilfe des Existenzquantors möglich, alle erfüllenden Belegungen nacheinander zu verarbeiten, d.h. es wurde nach der ersten erfüllenden Belegung tortzdem weitergesucht.
Mittels des ODL Ausdrucks

```
exists c:Component . (c.Name = "CompOld" and
  result has Name(c, "CompNew"))
```

konnte man alle Komponenten mit dem Namen CompOld in CompNew umbenennen. Durch die nicht-strikte Auswertung würde nun lediglich die erste gefundene Komponente dieses Namens umbenannt. Um den bisherigen Effekt zu erzielen, kann man sich aber der eingeschränkten Typen bedienen:

```
forall {c:Component | c.Name = "CompOld"} .
  result has Name(c, "CompNew"))
```

Hierbei wird zunächst der Typ `Component` auf die Instanzen reduziert, die „OldComp“ als Namen haben. Auf diesen wird dann die Umbenennung durchgeführt.

7.2. Generierung von Zeugen und Gegenbeispielen

7.2.1. Motivation

Neben dem boolschen Ergebnis wurden bei der Auswertung eines ODL Ausdrucks auch die Belegungen der enthaltenen Variablen, welche den Ausdruck erfüllen, berechnet. Dies führte häufig zu sehr umfangreichen Ausgaben an den Benutzer. Offensichtlich ist der ODL Ausdruck `forall b1:Boolean.forall b2:Boolean.true` wahr. Hierbei ist es unnötig dem Benutzer alle vier erfüllenden Belegungen mitzuteilen, da ja alle Kombinationen von Belegungen aufgrund der beiden Allquantoren wahr sein müssen.

Um die Ausgabe an den Benutzer informativer zu gestalten, wurde die Berechnung der erfüllenden Belegungen durch die Generierung von Zeugen und Gegenbeispielen ersetzt. Anhand von vier grundlegenden Fällen möchte ich dies veranschaulichen:

1.Fall: `forall a:A . forall b:B . EXPR`
Das Interesse besteht in diesem Ausdruck an den Variablenbelegungen (a, b), für die der Ausdruck `EXPR` zu false evaluiert. Wir suchen also Gegenbeispiele.

2.Fall: `forall a:A . exists b:B . EXPR`

Auch hier suchen wir wieder Gegenbeispiele. Wird ein solches Gegenbeispiel gefunden, so bedeutet dies, dass alle Belegungen der Variable b den Ausdruck EXPR für die gegebene Belegung von a zu false evaluierten. Deshalb besteht Interesse an den Belegungen der Variablen b. Das Gegenbeispiel ist durch das Tupel (a) ausreichend beschrieben.

3.Fall: `exists a:A . exists b:B . EXPR`
Hierbei suchen wir die Belegungen (a, b), für die `EXPR` zu true evaluiert. Es werden also Zeugen generiert. Man beachte, dass in Verbindung mit der nicht-strikten Auswertung der Quantoren, wie zuvor beschrieben, hier die erste erfüllende Belegung und nicht, wie bisher, alle berechnet werden.

4.Fall: `exists a:A . forall b:B . EXPR`
In diesem Ausdruck ist das Ergebnis durch das Tupel (a) ausreichend beschrieben, denn für diese Belegung müssen alle Belegungen von b den Ausdruck `EXPR` erfüllen. Deshalb müssen die Belegungen von b nicht an den Benutzer weitergegeben werden.

7.2.2. Spezifikation der Generierung

Um die Generierung von Zeugen und Gegenbeispielen durchzuführen, wird während der Evaluation ein spezielles Objekt, neben der Menge der aktuellen Variablenbelegungen, von jedem Knoten des Evaluationsbaumes weitergegeben. Dieses Objekt kann zunächst zwei Werte haben: Zeugengenerierung und Gegenbeispielgenerierung.

Bei einem Existenzquantorknoten wird eine erfüllende Belegung nur zurückgegeben, falls eine Zeugengenerierung verlangt wurde. Der Allquantor liefert ein Gegenbeispiel nur, wenn eine Gegenbeispielgenerierung gegeben war.

Da ein negierter All- einem Existenzquantor und ein negierter Existenz- einem Allquantor entspricht, muss der Negationsoperator den Wert "invertieren".

Es ergibt sich folgende Spezifikation für den Typ dieses speziellen Objekts:

```
SPEC AssignmentGeneration {
   sort AGEN;

   witness: -> AGEN
   counter-example: -> AGEN
   no-generation: -> AGEN

   invert: AGEN -> AGEN

   generated by witness, counter-example, no-generation

   invert(witness) = counter-example
   invert(counter-example) = witness
   invert(no-generation) = no-generation
 }
```

Der hinzugfügte Wert `no-generation` verhindert die Berechnung von Zeugen und Gegenbeispielen unabhängig vom jeweiligen Quantor. Dies ist beispielsweise bei der Auswertung der Bedingung innerhalb eines eingeschränkten Typs oder zur Auswertung der

`with`-Klausel im Fixpunktoperator nützlich, da hierdurch unnötiger Rechenaufwand gespart wird.

7.2.3. Änderungen an der ODL-Komponente Evaluation

Die Auswertung eines ODL Ausdrucks wird unter Verwendung der Methode `createEvaluationModel` der Klasse `EvaluationModelGenerator` durchgeführt. Bisher genügte hierzu die Angabe des Ausdrucks in Form eines Strings und der Metamodellkontext. Diese Schnittstelle wurde nun um eine weiteren Parameter vom Typ String erweitert, dessen in folgender Weise interpretiert wird (Groß-/Kleinschreibung spielt keine Rolle):

"witness"	Die Auswertung startet mit der Zeugengenerierung
"counter-example"	Die Auswertung startet mit der Gegenbeispielsgenerierung
Sonst	Es werden keine Zeugen oder Gegenbeispiele generiert.

Die vorherigen Abschnitt beschriebene Spezifikation wurde in der Klasse `quest.odl.evaluation.model..AssignmentGeneration` implementiert.

7.2.4. Änderungen an der ODL-Komponente Editor

Dem ODL Interpreter fehlen bisher Optimierungsverfahren, insbesondere die Erzeugung der Pränexform des vom Benutzer eingegebenen Ausdrucks. Deshalb kann die Entscheidung mit welcher Generierungsvariante an der Evaluationsbaumwurzel begonnen werden soll, noch nicht automatisch durchgeführt werden.

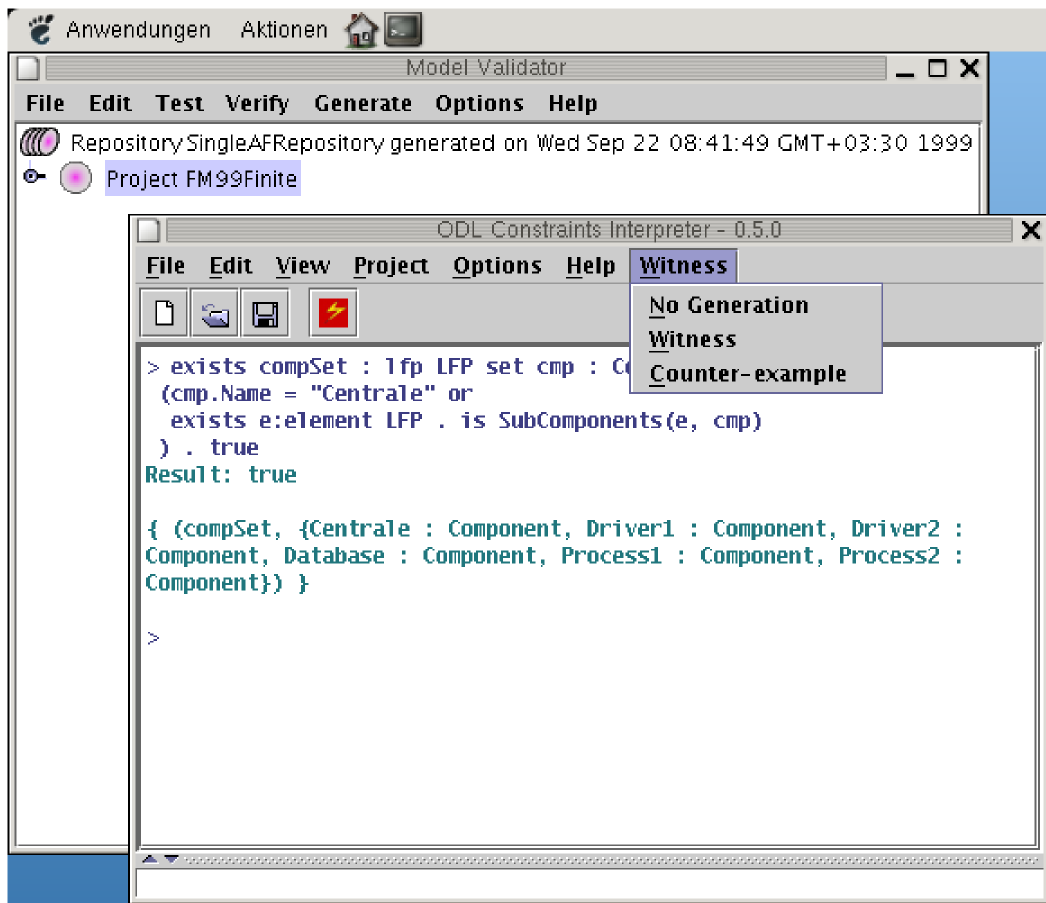


Abbildung 3: Auswahlmenü für die Generierungsvariante

Die Entscheidung wurde daher dem Benutzer überlassen und eine entsprechende Menüsteuerung in die grafische Interpreteroberfläche integriert. Das zusätzliche Menü, welches dem Anwender die entsprechende Auswahl ermöglicht, ist in Abbildung 3 dargestellt.

7.3. Framework für Funktionen

In diesem Abschnitt erläutere ich die Erweiterung des ODL Interpreters um ein Framework für Funktionen. Die folgenden Ausführungen zielen in erster Linie auf die Weiterentwicklung und nicht auf die Anwendung des Interpreters ab. Daher sind sie auch sehr nah am Quellcode gehalten, insbesondere handelt es sich bei den abschließenden Beispielen nicht um die Verwendung der beiden Funktionen `concat` und `isEmptySet` in einem ODL-Ausdruck, sondern um deren Implementierung auf Basis des beschriebenen Frameworks.

7.3.1. Motivation

Die Einführung des Frameworks für Funktionen in ODL hatte mehrere Gründe.
Der erste und wichtigste Grund lag in der bisherigen Vorgehensweise zur Implementierung von Funktionen (ODL verfügt bereits über die beiden Funktionen: `size`, `isEmpty`). Die Funktion `isEmpty` ist explizit als Element der ODL Grammatik definiert und dient dazu festzustellen, ob eine Menge leer ist oder nicht. Die Implementierung von Funktionen als grammatikalische Elemente erfordert jedoch einen deutlich höheren Aufwand als mit Hilfe des Frameworks und ist zudem sehr anfällig für Tippfehler.

Zum ersten war die Verwendung von Funktionen in Ausdrücken, also beispielsweise eine Funktion zum Konkatenieren zweier Zeichenketten, in der Grammatik bereits vorgesehen, der Interpreter brach jedoch die Ausführung eines entsprechenden ODL Ausdrucks mit dem Hinweis auf die fehlende Implementierung von Funktionen ab.

Zum zweiten erschien mir die Konkatenationsfunktion besonders im Hinblick auf das Kopieren oder Erzeugen von Modellelementen als nützlich, um z.B. die Namen der erzeugten Komponenten aus den Namen der Originalkomponenten durch Erweiterung zu generieren.

7.3.2. Grundkonzept des Frameworks

Im Zentrum des Frameworks stehen Funktionen unter zwei wesentlichen Gesichtspunkten. Zunächst muss für eine Funktion festgelegt sein, wie sie deklariert ist. Die Deklaration umfasst den Typ des Rückgabewerts und welche Anzahl von Parametern und deren Typen die Funktion akzeptiert. Für das Konkatenieren von zwei Zeichenketten wäre dies also zwei Parameter vom Typ Zeichenkette und eine Zeichenkette als Rückgabewert. Weiterhin zeichnet sich jede Funktion durch ihren Namen aus.

Zum zweiten muss die Funktion angewendet werden können. Der Gesichtspunkt der Funktionsapplikation ist also das Berechnen des Funktionsergebnisses ausgehend von konkreten Parameterwerten.

Ein Besonderheit für das Framework ergibt sich aus der Definition eines ODL Ausdrucks. Die Verwendung von Funktionen ist an zwei Stellen sinnvoll. Innerhalb eines Ausdrucks, der eine beliebiges Ergebnis haben darf, beispielsweise eine Berechnung in der die Größe einer Menge ein Rolle spielt, und innerhalb eines Terms, der ein boolsches Ergebnis haben muss.

Das Framework lässt grundsätzlich die Verwendung jeder Funktion in einem Ausdruck zu, natürlich vorausgesetzt, dass die Typprüfung erfolgreich ist. In einem ODL Term wird allerdings durch eine Kapselklasse gesichert, dass nur Funktionen mit dem Rückgabetyp `Boolean` verwendet werden können. Eine nicht typkonforme Anwendung einer Funktion in ODL führt zu einem entsprechenden Fehlerhinweis des Interpreters.

7.3.3. Implementierung des Frameworks

Das Framework wurde durch drei Klassen im Packet quest.odl.evaluation.model definiert:

- Function
- FunctionTerm
- FunctionFactory

Die abstrakte Klasse Function definiert die Funktionsdeklaration über die folgende Schnittstelle:

- String getFunctionName() liefert den Namen der Funktion
- MetaType getResultType() liefert den Typ des Rückgabewerts
- boolean isApplicable() bestimmt, ob die Funktion auf die gegebene Anzahl von Parametertypen angewendet werden darf.

Hinzu kommen noch die beiden Zugriffsmethoden zum Setzen der Parametertypen und der Argumentausdrücke:

- void setArgumentTypes(MetaType[]) setzt die Paramtertypen
- void setArguments(Expression[]) setzt die Argumentausdrücke

Während der Analysephase des Interpreters werden obige Methoden verwendet um die Typprüfung durchzuführen und gegebenenfalls den Benutzer über eine Typverletzung zu informieren. In Abbildung 4 ist dieser Teilablauf des Interpreters als Sequenzdiagramm dargestellt.

Als erstes wird die Hilfsklasse zur Übersetzung von Funktionsnamen, die Klasse FunctionFactory, benutzt um eine neue Instanz der dem Namen entsprechenden Function-Klasse zu erhalten (*getFunctionByName()*). Hierbei gilt die Konvention, dass eine ODL-Funktion mit dem Namen abcd in der Klasse `quest.odl.evaluation.model.functions.Abcd` implementiert sein muss. Die FunctionFactory-Klasse wurde so gestaltet, dass neue ODL-Funktionen implementiert werden können, ohne dass hier Änderungen notwendig sind. Es genügt die neue Funktion von Function abzuleiten und der Konvention entsprechend zu benennen.

Nachdem nun das Function-Objekt zur Verfügung steht werden im die Typen der Parameter mitgeteilt (*setArgumentTypes()*) und anschließend abgefragt, ob die Funktion auf Argumente mit den gegebenen Typen in der gegebenen Reihenfolge anwendbar ist (*isApplicable()*). Diese Parametertypprüfung muss in zwei Methodenaufrufen stattfinden, wie gleich erläutert wird.

Als drittes wird das Function-Objekt für die Auswertung durch Setzen der Argumentausdrücke vorbereitet (*setArguments()*). Durch die Implementierung der Schnittstelle Expression ist die Auswertung einer Funktion in den normalen Evaluationsablauf des ODL Interpreters eingegliedert. Im Evaluationsbaum ist die Funktionsapplikation also ein Knoten, der selbst die ihm übergebenen Argumentausdrücke auswerten muss. Mit dem Ergebnis kann die Funktion dann ihr Ergebnis berechnen. Die Auswertung der Argumente und die Berechnung ist vom Entwickler zu implementieren (*evaluate(…)*).

Bevor jedoch die Auswertung vorgenommen werden kann, findet während der Analyse des Vaterknotens eine Typprüfung statt. Über *getResultType()* muss das Function-Objekt den Rückgabetyp mitteilen. Im Falle von Funktionen wie `isEmptySet` kann dieser Typ statisch angegeben werden. In speziellen Fällen hängt der Rückgabetyp aber von den Parametertypen ab, wie zum Beispiel bei der Mengenvereinigung. Werden zwei Mengen von Komponenten vereinigt, so ist der Rückgabetyp wiederum eine Menge von Komponenten. Werden hingegen zwei Mengen von Kanälen vereinigt, entsteht selbstverständlich wieder ein Menge von Kanälen. Die Vereinigungsfunktion ist also nicht nur in den Parametern polymorph.

```
union(Set A, Set A, …) Set A
```

Dies erklärt, wieso zuvor ein Setzen der Parametertypen notwendig war und es nicht ausreicht, für jede Funktion einen statischen Rückgabetyp festzulegen.

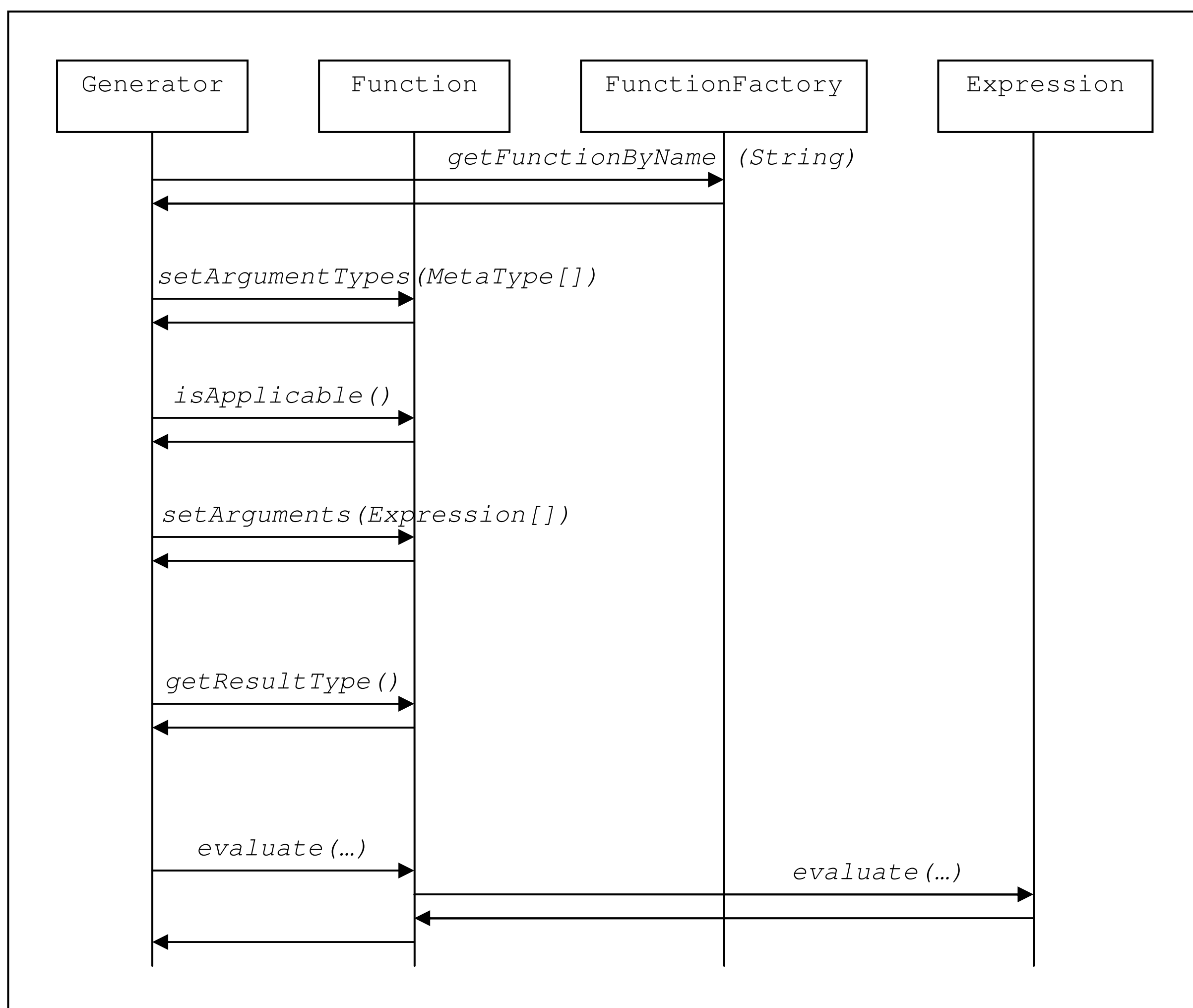


Abbildung 4: Analyse- und Evaluationsablauf für Funktionen

Wie bereits erläutert, können nicht alle Funktionen in ODL Termen verwendet werden. Die Klasse FunctionTerm dient hierbei als die Kapselklasse für Function-Objekte. Während des Aufbaus des Evaluationsbaums wird überprüft, ob die vom Benutzer in einem Term verwendete Funktion wirklich den Rückgabetyp Boolean hat und falls nicht eine entsprechende Fehlermeldung erzeugt.

7.3.4. Beispiel: concat und isEmptySet

Abschließend möchte ich kurz die beiden Beispielimplementierungen skizzieren.

`concat: quest.odl.evaluation.model.functions.Concat`

- `getResultType()`: `concat` liefert ein Ergebnis vom Metatyp `MetaString`
- `isApplicable()`: `concat` akzeptiert beliebig viele Parameter, solange alle vom Metatyp `MetaString` sind.
- `getFunctionName()`: liefert "concat".
- `evaluate()`: evaluiert die Parameterausdrücke und liefert die in der Reihenfolge zusammengefügte Zeichenkette zurück.

`isEmptySet: quest.odl.evaluation.model.functions.IsEmptySet`

- `getResultType()`: `isEmptySet` liefert ein Ergebnis vom Metatyp `MetaBool`.
- `isApplicable()`: `isEmptySet` akzeptiert genau einen Parameter vom Metatyp `MetaSetType`.
- `getFunctionName()`: liefert "isEmptySet".
- `evaluate()`: evaluiert den Parameterausdruck und prüft, ob die gelieferte Menge leer ist oder nicht.

Kapitel 8 Erweiterungen und weitere Einsatzgebiete

In diesem Kapitel möchte ich abschließend einige Gedanken zu Erweiterungsmöglichkeiten der ODL und zu weiteren Einsatzmöglichkeiten der zuvor vorgestellten neuen Sprachkonstrukte skizzieren.

8.1. Zusätzliche Funktionen

Das in Kapitel 7.3 vorgestellte Framework für Funktionen erlaubt es, die ODL mit wenig Aufwand um zusätzliche Funktionen zu erweitern. In [Trachtenherz] wurden bereits einige Vorschläge (Abschnitt 6.1.2. Mengenoperationen, Seite 97) für Funktionen über Mengen gemacht: `union`, `intersection`, `difference`.

8.2. Benutzerdefinierte Funktionen

Das Framework für Funktionen kann nicht nur um zusätzliche Funktionen erweitert werden. Es ist auch denkbar die Sprache ODL dahingehend auszubauen, dass der Benutzer selbst Funktionen definieren kann.
Im Einzelnen sind dazu folgende Schritte notwendig:

- Erweiterung der ODL Grammatik um eine Syntax zur Definition von Funktionen
- Implementierung einer generischen Funktion, welche eine Funktionsdefinition auf die Frameworkschnittstelle umsetzt.
- Erweiterung der FunctionFactory um die Möglichkeit, die benutzerdefinierten Funktionen bekannt zu machen, sodass das bisherige Aufrufverfahren angewandt werden kann.

Die Entscheidung, die ODL von der Abfragesprache in die Richtung einer Programmiersprache, durch das Zulassen von Benutzerfunktionen , zu erweitern, liegt nicht bei mir. Ich möchte, lediglich zeigen, dass das Framework eine Möglichkeit bietet, diesen Weg zu beschreiten.

Es wäre beispielsweise denkbar, eine eigene Sprache für die Definition von Benutzerfunktionen zu definieren, welche mittels der generischen Funktion analysiert und kompiliert würde. Diese Sprache könnte dann nicht nur über den ODL Interpreter eingegeben werden sondern auch zur Definition von Funktionen, wie in Abschnitt eins, verwendet werden.

8.3. Menge erreichbarer Automatenzustände

Ein zum Tiefenkopie ähnliches Problem findet sich in AQuA bei der Beschreibung von Komponentenverhalten durch Zustandsautomaten. Diese bestehen aus den beiden Modellelementen `State` (Zustand) und `Transition` (Übergang). Letztere verbinden jeweils zwei Zustände miteinander und definieren somit eine Relation zwischen Zuständen.
Das Problem besteht nun darin festzustellen, welche Zustände von einem gegebenen Zustand aus über Transitionen in einem oder mehreren Schritten erreichbar sind. Hierfür benötigt man wiederum ein rekursives Verfahren. Es kann also wieder der `lfp`-Operator zur Anwendung kommen.

Kapitel 9 Post-SEP Entwicklungen und aktueller Stand

Die folgenden Unterabschnitte sind für die Weiterentwicklung gedacht und dokumentieren die wesentlichen Änderungen und Erweiterungen, die nach Abschluß der im SEP festgelegten Aufgaben, vorgenommen wurden.

9.1. Generierungsmethode und Evaluationsmethode

Die in Abschnitt 7.2. eingeführte Generierung von Zeugen und Gegenbeispielen wurde weiter verallgemeinert, indem es nun auch möglich ist, sowohl Zeugen als auch Gegenbeispiele zu generieren (Generierungsmethode).
Desweiteren wird das Generierungsobjekt zusätzlich dazu benutzt festzulegen, ob die Auswertung der Quantoren und der logischen Operatoren strikt oder nicht strikt durchgeführt werden soll (Evaluationsmethode). Diese Auswahl ist analog über ein entsprechendes Menü im ODL-Interpreter zugänglich und wird durch die Invertierungsfunktion nicht beeinflusst.

Aus diesen neuen Grundlagen für das Generierungsobjekt ergibt sich folgende Spezifikation:

```
SPEC AssignmentGeneration {
   sort AGEN;

   witness: BOOL -> AGEN
   counter-example: BOOL -> AGEN
   no-generation: BOOL -> AGEN
   both: BOOL -> AGEN

   invert: AGEN -> AGEN
   isStrict: AGEN -> BOOL

   generated by witness, counter-example, no-generation, both

   invert(witness(x)) = counter-example(x)
   invert(counter-example(x)) = witness(x)
   invert(no-generation(x)) = no-generation(x)
   invert(both(x)) = both(x)

   isStrict(witness(x)) = x
   isStrict(counter-example(x)) = x
   isStrict(no-generation(x)) = x
   isStrict(both(x)) = x
 }
```

Umbauarbeiten am Sourcecode umfassten die Anpassung der logischen Operatoren `Conjunction`, `Disjunction` und `Implication`, sowie der beiden Quantoren `forall` und `exists`.

9.2. Less, Union, Intersect

Die oben angedeuteten Mengen-Operationen wurden mit Hilfe des FunctionFrameworks implementiert. Die Durchführung der jeweiligen Mengenoperationen wird durch entsprechende

Methoden in der Klasse `SetValue`, welche Mengen konkreter ODL Objekte repräsentiert, realisiert.

9.3. Monotonie-Prüfung

Die in Abschnitt 4.2 erwähnte Prüfung der with-Klauselprädikate auf Monotonie ist implementiert. Während der Konstruktion des Evaluationsbaumes wird, sobald die with-Klausel interpretiert wurde, ein spezieller TreeWalker (`quest.odl.evaluation.model.analysis.MonotonyChecker`) gestartet. Dieser prüft, ob über die Elemente der Fixpunktmenge in einem Universalquantor oder einem negierten Existenzquantor iteriert wird. Ist dies der Fall, so ist die with-Klausel kein monotones Prädikat; es gibt folglich keinen eindeutigen kleinsten bzw. größten Fixpunkt. Der Benutzer wird durch eine entsprechende Fehlermeldung auf diese Unzulässigkeit hingewiesen.

9.4. Element Elimination

Die syntaktische Ergänzung `element` ist nicht mehr implementiert. Für den Zugriff auf eine definierte Variable, die eine Menge beschreibt, genügt es diese direkt hinzuschreiben.
Statt wie bisher

```
context someSet: set Port . context e:element someSet . true
```

lautet der äquivalente Ausdruck nun

```
context someSet: set Port . context e:someSet . true
```

Analoges gilt für das `element(...)`-Konstrukt. Das Schlüsselwort fällt weg, die Klammern müssen aber bestehen bleiben, falls ein Selektorzugriff gemacht wird:

```
context c:Component . context e:(c.Ports) . true
```

Der Grund liegt im Punkt, der den Quantor abschließt bzw. Selektoren trennt.

Mengenfunktionen benötigen ebenfalls keine Klammern mehr:

```
context someSet1:set Port . context someSet2:set Port .
  context e:union(someSet1, someSet2) . true
```

Aus Code-Sicht ergaben sich nur kleine Änderungen in der Grammatik und der Übersetzungsklasse. Der Fall eines Modellelementtyps und einer definierten Variable (beides nur Identifikatoren) wird nun durch die Übersetzung aufgelöst, indem zunächst geprüft wird, ob ein Modellelement mit entsprechendem Identifikator existiert, und, wenn dies nicht der Fall ist, erst danach die Liste der definierten Variablen überprüft. Dies bedeutet, dass der Benutzer keine Variablen definieren darf, die gleichlauten wie Modellelemente, da eer auf diese nicht zugreifen kann (Verschattung).

Der geklammerte Elementzugriff erwartet nun schon grammatikalisch einen Selektorausdruck und nicht wie bisher einen allgemeinen Ausdruck (`expression`).

9.5. Immunität von ODL gegen primitive Datentypen im Metamodell

Da sich im Metamodell an verschiedenen Stellen Attribute eingeschlichen haben, die primitive Java-Datentypen (int, float, boolean, etc.) als Typ haben, konnten bisher solche Attribute nicht in einer "result has"-Klausel gesetzt werden. Die ODL-Typprüfung, die eine Namensgleichheit voraussetzt, meldete hier einen scheinbar bestehenden Typkonflikt (bspw. boolean vs. Boolean). Die Java-Reflection-API, welche für den Aufruf zum Setzen des Attributwertes benutzt wird, ist hier flexibler.

Die vorgenommenen Änderungen machen ODL nun immun gegen diese Art von Typkonflikten. Dies wurde durch die Erweiterung der ODL-Typprüfung erreicht. Sollte keine Namensgleichheit vorliegen, wird zusätzlich die übliche Abbildung von Java überprüft. Weitere Anpassungen waren im Bereich der Aufrufe mittels Java-Reflection notwendig.

ODL ist nun nach aussen typsicher, d.h. primitive Datentypen können vom ODL-Nutzer wie nicht-primitive behandelt werden. Der ODL-Nutzer verwendet folglich nur die bekannten Typen: Boolean, Integer.

9.6. Verbesserter Umgang mit Objekthierarchien des Metamodells

Beim Überprüfen und Setzen einer Relation verlangte ODL bisher, dass das beide Objekte den Metamodelltyp hatten, der direkt an der Relation bestand. Hier wurde auch nur auf Namensgleichheit getestet. Es war folglich nicht möglich die Beziehung zwischen zwei Objekten festzustellen, falls diese über eine Relation in Superklassen verbunden waren.

BSP.: DataDef ist Subklasse von TypeDef, AbstractType ist Subklasse von TConst
TConst ist über eine Relation TypeDef mit TypeDef verbunden. Nicht möglich aufgrund von verlangter Namensgleichheit war folgender ODL-Ausdruck:

```
  exists dd:DataDef . exists at:AbstractType . at.TypeDef = dd
```

Diese Problematik wurde, unter Rückgriff auf die im Metamodell hinterlegte Subklasseninformation, gelöst. Es ist jetzt möglich in ODL auch mit Metamodelltyphierarchien umgehen zu können.

9.7. indexOf-Funktion

Sie dient zur Handhabung von geordneten MM-Beziehungen.

Einige wenige Relationen im Metamodell sind als geordnet markiert. Bisher konnte diese Ordnung aus Sicht von ODL nicht sichtbar gemacht werden. Die Funktion indexOf schafft hier eine Übergangslösung:

```
  indexOf: Zielelement x Basiselement x Relation -> Integer
```

Beispiel:

```
  exists c:Constructor. exists s:(c.Selectors).
    indexOf(s, c, "Selectors") = 0
```

Es sei angemerkt, dass das Beispiel sicherstellt, dass s in der Relation Selectors von c aus enthalten ist. Dies ist eine Vorbedingung für das Funktionieren der indexOf-Funktion. Eine allgemeinere Form des Beispiel, bei der s über alle Selektoren (exists s:Selector) quantifiziert, wurde nicht zugelassen, da hier die Fehlersemantik nicht sinnvoll definiert werden konnte.

Die indexOf-Funktion stellt eine Möglichkeit dar, die interne Ordung von Relationen sichtbar zu machen. Sie sollte jedoch mit äusserster Vorsicht eingesetzt werden.

9.8. filter-select Mechanismus

Die Anzeigefunktion des ODL Interpreters wurde dahingehend geändert, dass der Benutzer die anzuzeigenden Variablenbelegungen filtern bzw. einzelne Variablen gezielt selektieren kann.

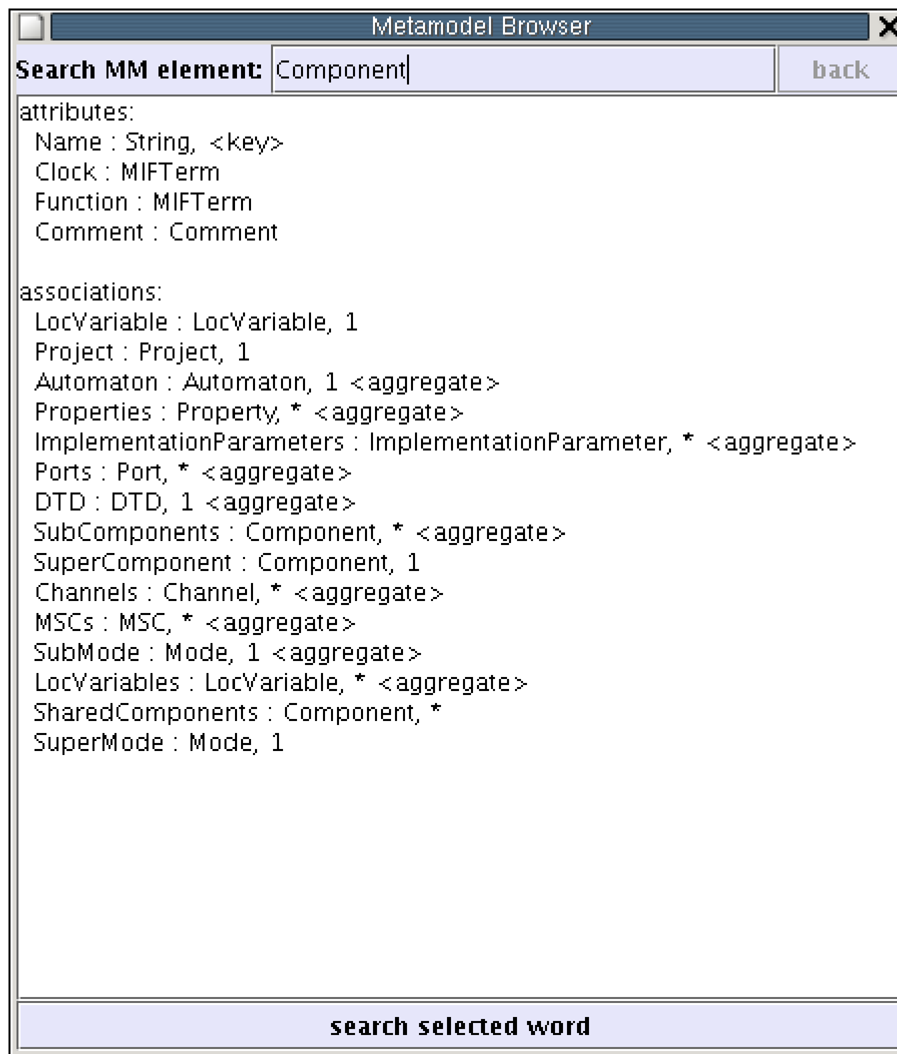
Er gibt dies zu Beginn seines ODL-Ausdrucks mittels der Schlüsselworte filter und select, gefolgt von einer in runde Klammern gesetzten Liste von Variablennamen (Kommatrennung), an.

Beispiele (strikte Auswertung, witness-Generierung):

```
>
filter (b)
exists b:Boolean . exists b2:Boolean . true
Result: true

{ (b2, false) }
{ (b2, true) }
{ (b2, false) }
{ (b2, true) }

>
select (b)
exists b:Boolean . exists b2:Boolean . true

Result: true

{ (b, false) }
{ (b, false) }
{ (b, true) }
{ (b, true) }
```

9.9. Metamodell Browser

Im Hilfemenü des ODL-Intpreters findet sich der Zugang zum Metamodell-Browser. Dieser ist ein einfaches Unterstützungswerkzeug zur Darstellung der internen Repräsentation des Metamodells. Wie die Abbildung zeigt, bietet er die rudimentären Browserfunktionen an. In der Kopfzeile findet sich ein Eingabefeld zum Suchen von Metamodellelementen. Daneben gibt es einen Zurück-Knopf. Die Schaltfläche am unteren Rand dient dazu, den im Anzeige-fenster markierten Text zu suchen. Dies ermöglicht eine einfache Browse-Funktion mittels der Doppelklick-Markierungsfunktion.



Der MMBrowser ist, wie der gesamte ODL-Interpreter, über die QUESTModelMapper-Klasse mit dem Metamodell verbunden. Das Auslesen der MM-Informationen geschieht über diese Schnittstelle und ist daher transparent gegenüber Änderungen des Metamodells.

9.10. define-run-Subroutinen-Konstrukte

Die define-run Konstrukte stellen das Analogon zu den benamten Prädikaten dar. Letztere erlaubten nur die eingeschränkten CCL-Propositions. Daher konnte man mit ihnen keine neuen Elemente anlegen bzw. Relationen bearbeiten. Dies ist mit den define-run-Konstrukten nun möglich, da sie normale Propositions erlauben. Sie haben jedoch nur lokale Bedeutung, d.h. sie können nur innerhalb eines ODL-Ausdrucks definiert und aufgerufen werden.

Syntax:

```
define_statement ::=
  define <ident> ( <paramlist> ) as ( <proposition> ) .
  <unary_proposition>

run_statement ::=
  run <ident> ( <arguments> )
```

Grammatikalisch sind beide Anweisungen als unary_proposition zu betrachten. Es ist möglich innerhalb einer define-Anweisung eine weitere zu verschachteln. Dies macht jedoch keinen Unterschied, da es weder nicht möglich ist, rekursive Konstruktionen zu bauen, noch von der inneren define-Anweisung auf die Parameter der äußeren zuzgreifen. run-Anweisungen dürfen sich nur auf bereits vollständig definierte Subroutinen beziehen. Dies verhindert rekursive Aufrufe.

Beispiele:

```
// small test //
define mySub (c:Component) as (
  c.Name = "Counter"
) .
exists co:Component . run mySub(co)

// c is not visible from within mySub2 //
// undefined variable error is shown   //
define mySelf (c:Component) as (
  define mySelf2 (c2:Component) as (
    c.Name = "Counter"
  ) .
  true
) .
exists co:Component . run mySelf2(co)

// unbounded type parameters are allowed//
define mySub(s:String) as (
  true
) .
exists co:Component . run mySub(co.Name)
```

Die hier dargestellten Beispiele sollen lediglich die Syntax verdeutlichen. Eine gute Einsatzmöglichkeit für Subroutinen sind ODL Ausdrücke, in denen an mehreren Stellen kleine Aufgaben, wie z.B. das Setzen aller Attribute eines neuen Elements, durchgeführt werden müssen.

Anhang A: ODL Grammatik Version 0.6.3

Die Grammatik entspricht der Quellcodeversion.Es wurden lediglich einige, für die automatische Parsergenerierung notwendige, technische Angaben entfernt. Terminalsymbole und Schlüsselworte sind **fett** gedruckt.

```
odl_start = proposition |
            named_predicate_declaration;

named_predicate_declaration =
  identifier ( bounded_type_list ) := ccl_proposition;

proposition =
  unary_proposition |
  proposition and unary_proposition |
  proposition or unary_proposition |
  proposition implies unary_proposition |
  proposition equiv unary_proposition;

ccl_proposition =
  ccl_unary_proposition |
  ccl_proposition and ccl_unary_proposition |
  ccl_proposition or ccl_unary_proposition |
  ccl_proposition implies ccl_unary_proposition |
  ccl_proposition equiv ccl_unary_proposition;

unary_proposition =
  neg unary_proposition |
  forall_proposition    |
  exists_proposition    |
  context_proposition   |
  named_predicate_call  |
  term;

forall_proposition =
  forall bounded_variable_definition . unary_proposition;

exists_proposition =
  exists bounded_variable_definition . unary_proposition |
  exists post_universe_variable_definition . uary_proposition;

context_proposition =
  context context_extension?
    bounded_variable_definition . unary_proposition |
  context context_extension?
    unbounded_variable_definition . unary_proposition;

ccl_unary_proposition =
  neg ccl_unary_proposition |
  ccl_quantifier bounded_variable_definition .
    ccl_unary_proposition |
  named_predicate_call |
  ccl_term;
```

```
ccl_quantifier = forall | exists;

context_extension = [ hint_extension? ];
hint_extension = hint = string_constant_expr_list;

string_constant_expr_list =
  string_constant_expr string_constant_expr_list_tail*;
string_constant_expr_list_tail = , string_constant_expr;

term = basic_proposition | ( proposition );

ccl_term = ccl_basic_proposition | ( ccl_proposition );

basic_proposition =
  relation |
  bool_proposition |
  functional_proposition ;

ccl_basic_proposition =
  ccl_relation |
  bool_proposition |
  functional_proposition;

functional_proposition = call_expression;

bool_proposition =
  equal_expression |
  bigger_smaller_expression |
  isempty ( expression ) |
  bool_constant_expr;

bigger_smaller_expression =
  expression comparison_operator expression;

comparison_operator = > | < | >= | <=;

relation =
  pre_relation |
  post_relation;

ccl_relation =
  pre_relation;

pre_relation = is call_expression;

post_relation =
   result has call_expression |
   result not has call_expression;

call_expression = identifier ( args );

named_predicate_call = call call_expression;

args = arglist?;

arglist = arg arglist_tail*;
```

```
arglist_tail = , arg;

arg = expression;

non_constant_expression =
  functional_expression |
  selector_expression |
  defined_variable;

expression =
  non_constant_expression |
  arithmetic_expression |
  constant_expression;

constant_expression =
  bool_constant_expr |
  string_constant_expr;

bool_constant_expr = bool_constant;

int_constant_expr = sign? int_constant;

string_constant_expr = string_constant;

sign = + | -;

functional_expression = call_expression;

selector_expression = defined_variable . selection;

selection = selector selection_tail*;

selection_tail = . selector;

selector = identifier;

defined_variable = variable;

variable = identifier;

equal_expression = expression = expression;

arithmetic_expression =
  factor |
  arithmetic_expression + factor |
  arithmetic_expression - factor;

factor =
  arithmetic_term |
  factor * arithmetic_term;

arithmetic_term =
  int_constant_expr |
  ( arithmetic_expression ) |
  size ( expression );

bounded_variable_definition = variable : bounded_type;
```

```
unbounded_variable_definition = variable : unbounded_type;

post_universe_variable_definition = variable : post_universe_type;

bounded_type =
  bounded_product_type |
  bounded_restricted_type_definition |
  bounded_set_type_definition |
  element defined_variable |
  element ( expression );

bounded_restricted_type_definition =
  { variable : bounded_type | ccl_proposition };

bounded_set_type_definition =
  basic_set_type_definition |
  fixed_point_set_type_definition |
  bounded_map_type_definition;

basic_set_type_definition = set bounded_type;

fixed_point_set_type_definition =
  lfp variable set variable : bounded_type
      with ( ccl_proposition ) |
  gfp variable set variable : bounded_type
      with ( ccl_proposition );

bounded_map_type_definition =
  map variable : bounded_type
    to variable : bounded_type;

bounded_product_type = bounded_unary_type | ( bounded_type_list );

bounded_type_list = identifier : bounded_type |
  identifier : bounded_type , bounded_type_list;

bounded_unary_type = bounded_basic_type | model_element_type;

bounded_basic_type = bool_type;
model_element_type = identifier;

unbounded_type =
  unbounded_product_type |
  unbounded_restricted_type_definition |
  unbounded_set_type_definition;

unbounded_set_type_definition = set unbounded_type;

unbounded_restricted_type_definition =
  { variable : unbounded_type | ccl_proposition };

unbounded_product_type =
  unbounded_unary_type |
  ( unbounded_type_list );
```

```
unbounded_type_list =
  identifier : unbounded_type unbounded_type_list_tail* |
  identifier : bounded_type , unbounded_type_list;

unbounded_type_list_tail =
  , identifier : bounded_type |
  , identifier : unbounded_type;

unbounded_unary_type = unbounded_basic_type;

unbounded_basic_type = int_type | string_type;

post_universe_type =
  post_universe_model_type |
  post_universe_map_type_definition;

post_universe_model_type = new model_element_type;

post_universe_map_type_definition =
  map variable : bounded_type
    to variable : post_universe_model_type;
```

Anhang B: ODL Ausdruck für die Tiefenkopie mit Ports und Kanälen

```
/* Benutzereingabe: Projekt und Komponente
context proj:Project . context userComp:Component .

/* Subkomponenten rekursiv ermitteln */
exists compSet: lfp cs set csc:Component
with (csc = userComp or
  exists cse:element cs . is SubComponents(cse, csc)
) .

/* Neue Komponenten als Abbildung erzeugen */
exists compMap: map orig:element compSet
to copy:new Component . (

  /* Injektivitätsforderung */
  (forall e1:element compMap . forall e2:element compMap .
     (e1.copy = e2.copy implies e1.orig = e2.orig)
  )
  and

  /* Für alle Ports neue Ports als Abbildung erzeugen */
  exists portMap: map orig:{pMp:Port |
    exists pMpComp:element compSet.is Ports(pMpComp, pMp)}
  to copy: new Port . (

    /* Injektivitätsforderung */
    (forall e3:element portMap . forall e4:element portMap .
      (e3.copy = e4.copy implies e3.orig = e4.orig)
    )
    and

    /* Kopieren der Portattribute */
    forall prtElem: element portMap .(
      result has Name(prtElem.copy, prtElem.orig.Name) and
      result has Type(prtElem.copy, prtElem.orig.Type) and
      result has Direction(prtElem.copy, prtElem.orig.Direction)
    )
    and

    /* Kopieren der Komponenten */
    forall mapElem:element compMap . (

    /* Kopieren der Komponentenattribute */
      result has Name(mapElem.copy, mapElem.orig.Name) and

    /* Zuweisung der neu erzeugten Ports */
      forall prt:element (mapElem.orig.Ports) . (
        exists nprtElem:element portMap . (
          prt = nprtElem.orig and
          result has Ports(mapElem.copy, nprtElem.copy)
        )
      ) and
```

```
    /* Kopieren der Kanäle */
      forall ch:element (mapElem.orig.Channels) .
      exists nch: new Channel . (

      /* Kopieren der Kanalattribute */
        result has Name(nch, ch.Name) and
        result has Type(nch, ch.Type) and

      /* Zuweisung des neu erzeugten Kanals */
        result has Channels(mapElem.copy, nch) and

      /* Kanal an Ports binden */
        exists origPort:element portMap . (
          origPort.orig = ch.SourcePort and
          result has SourcePort(nch, origPort.copy)
        ) and
        exists copyPort: element portMap . (
          copyPort.orig = ch.DestinationPort and
          result has DestinationPort(nch, copyPort.copy)
        )
      )
      and

      /* Aufbau der anfangs zerlegten Subkomponentenstruktur */
      forall mapElem2: element compMap . (
        is SubComponents(mapElem.orig, mapElem2.orig) implies
          result has SubComponents(mapElem.copy, mapElem2.copy)
      )
    )
  ) and

  /* Zuweisen der Kopie an das Projekt */
  exists topLevelMapElem: element compMap . (
    (topLevelMapElem.orig = userComp) and
    (result has Components(proj, topLevelMapElem.copy))
  )
)
```

Anhang C: Notwendige Umformung von ODL Ausdrücken

In diesem Anhang fasse ich die Veränderungen von ODL Ausdrücken zu bisherigen Interpreterversionen zusammen.

Einführen neuer Elemente
Bisher:

```
context baseComp:Component . new nc:Component .
  (result has Name(nc, "NewComp") and
   result has SubComponents(nc, baseComp)
  )
```

Neu:

```
context baseComp:Component . exists nc: new Component .
  (result has Name(nc, "NewComp") and
   result has SubComponents(nc, baseComp)
  )
```

Parameterdefinition bei benamten Prädikaten für Fixpunktmengen und Abbildungen
Fixpunktmenge:

```
fixedSet : lfp LFP set cmp:Component with (...)
```

Benamtes Prädikat:

```
namedPred (fixedSetParam: set Component) := ...
```

Nicht-strikte Auswertung bei Verwendung des Existenzquantors
Bisher:

```
// benennt alle Komponenten mit dem Namen 'CompOld'
// nach 'CompNew' um
exists c:Component . (c.Name = "CompOld" and
  result has Name(c, "CompNew"))
```

Neu:

```
// jetzt müssen alle umzubenennenden Komponenten
// explizit eingesammelt werden
forall {c:Component | c.Name = "CompOld"} .
  result has Name(c, "CompNew"))
```

Anhang D: Literatur

[AFHome] *Die AutoFocus Homepage*, http://autofocus.in.tum.de. Fakultät für Informatik, Technische Universität München.

[ODLAPI] *ODL API Specification v 1.1*. Generated by javadoc from sources, 2004.

[Pasch] D. Pasch. *Konzeption und Implementierung eines ODL-Interpreters für das Auto-Focus/Quest CASE-Werkzeug*.
Bachelor Thesis, Technische Universität München, 2002.

[QUESTHome] *Software Development Project: QUEST*, http://www4.in.tum.de/proj/quest. Fakultät für Informatik, Technische Universität München.

[SableCC] É. Gagnon. *SableCC, An Object-Oriented Compiler Framework*. School of Computer
Science, McGill University, Montreal, 1998.

[Schätz] B. Schätz. *The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA. Technischer Bericht TUM-I0111.*
Institut für Informatik, Technische Universität München, 2001.

[Trachtenherz] D. Trachtenherz. *Erweiterung des AQuA-Systems: ODL-Sprachkonstrukte und interaktive Benutzerschnittstelle*. Diplomarbeit, Technische Universität München, 2003