Diplomarbeit

Entwurf und Implementation eines metamodellbasierten OCL-Compilers

bearbeitet von

Stefan Ocke

geboren am 17. Februar 1978 in Borna

Technische Universität Dresden

Fakultät Informatik

Institut für Software- und Multimediatechnik

Lehrstuhl für Softwaretechnologie

Betreuer: Dipl.-Inf. Sten Löcher

Hochschullehrer: Prof. Dr. rer. nat. habil. Heinrich Hußmann

Eingereicht am 30. Juni 2003

Abbildungsverzeichnis

1 Einleitung

Die Object Constraint Language (OCL) hat sich in den letzten Jahren als fester Bestandteil der Forschung am Lehrstuhl für Softwaretechnologie der Technischen Universität Dresden etabliert. So trugen z.B. mehrere Diplom- und Belegarbeiten ([Fin99], [Fin00], [Wie00], [Loe01]) zur Entwicklung des Dresden OCL Toolkit bei, welches einen Compiler und andere Werkzeuge für OCL 1.3 umfasst.

Ziel der vorliegenden Arbeit ist die Entwicklung eines Compilers für die gegenwärtig in der Entstehung befindliche OCL-Version 2.0. Grundlage dafür ist das in [OCL16] vorgeschlagene OCL-Metamodell.

1.1 OCL

OCL ist eine formale Spezifikationssprache zur Formulierung seiteneffektfreier Ausdrücke über UML[1]-Modellen. Sie wurde von IBM entwickelt und ist seit UML-Version 1.1 Bestandteil des UML-Standards [UML11] der OMG[2].

OCL findet nicht nur Anwendung in UML-Modellen, sondern auch in der Definition der UML selbst in Form sogenannter Wellformedness Rules (WFRs) für das UML-Metamodell. Analog lässt sich OCL für jedes Metamodell nutzen, das zur Meta Object Facility (MOF) der OMG konform ist.

Die bisherige Spezifikation der OCL gibt eine Grammatik an und beschreibt die Semantik von OCL-Ausdrücken in textueller Form. Dieser Zustand ist unbefriedigend, da eine solche Beschreibung nicht präzise genug ist und unterschiedliche Auslegungen ermöglicht. Aufgrund dieses Mangels veröffentlichte die OMG das Dokument [OCLRFP], welches ein MOF-konformes Metamodell für OCL fordert.

Boldsoft, Rational Software Corporation, IONA und Adaptive Ltd. unterbreiteten – u.a. mit Unterstützung durch die Technische Universität Dresden – einen Vorschlag für das OCL-Metamodell. Dieser enthält neben dem Metamodell auch die darauf aufbauende Spezifikation der Semantik, einerseits mit Mitteln der UML formuliert, andererseits in mathematischer Form, basierend auf [Ric01].

---

[1] UML: **U**nified **M**odeling **L**anguage
[2] OMG: **O**bject **M**anagement **G**roup

Der Vorschlag erweitert die OCL auch um einige neue Sprachkonzepte. So wurden z.B. Tupeltypen eingeführt, um zu erreichen, dass OCL zumindest die gleiche Ausdruckskraft wie SQL[1] erlangt.

Mit der damit forcierten Entwicklung zur vollwertigen Anfragesprache kommt OCL eine entscheidende Rolle im Rahmen der MDA[2] zu. So ist es z.B. möglich, OCL innerhalb einer Transformationsspezifikation zu nutzen, um Elemente eines Modells zu selektieren. Solche Anfragen an Modelle sind den weiter oben erwähnten WFRs ähnlich: In beiden Fällen werden OCL-Ausdrücke über MOF-konformen Metamodellen formuliert.

Solche OCL-Ausdrücke über Metamodellen lassen sich noch nicht als Instanzen des im Vorschlag angegebenen OCL-Metamodells darstellen, da dieses gegenwärtig auf der UML Version 1.4. basiert. Erst im Rahmen von UML 2.0 werden MOF und UML über einen gemeinsamen Kern verfügen. Sobald das OCL-Metamodell mit diesem Kern integriert ist, können auch OCL-Ausdrücke über Metamodellen als Instanzen des OCL-Metamodells beschrieben werden.

1.2 Metamodellbasierter OCL-Compiler

Der in dieser Arbeit vorgestellte OCL-Compiler greift der weiteren Entwicklung der OCL-Spezifikation voraus: Es wird das in [OCL16] vorgeschlagene OCL-Metamodell so geändert, dass eine Integration mit den Metamodellen von UML 1.5[3] und MOF 1.4 vollzogen werden kann.

Auf Grundlage dieses integrierten Metamodells sind alle Komponenten des OCL-Compilers entworfen. So erzeugt z.B. der Parser aus OCL-Ausdrücken in textueller Form Instanzen des OCL-Metamodells. Code-Generatoren transformieren diese Instanzen in Code einer Programmiersprache.

Der in dieser Arbeit entwickelte Code-Generator beschäftigt sich speziell mit der Transformation von OCL-Ausdrücken über Metamodellen. Der von ihm erzeugte Java-Code kann genutzt werden, um zu validieren, ob ein Modell die WFRs seines Meta-

[1] SQL: **S**tructured **Q**uery **L**anguage
[2] MDA: **M**odel **D**riven **A**rchitecture
[3] UML 1.5 hat bezüglich des OCL-Metamodells vernachlässigbare Unterschiede zu UML 1.4.

modells erfüllt. Des Weiteren können auch beliebige andere Anfragen in Form von OCL-Ausdrücken an Modelle gestellt werden.

1.3 Verwandte Technologien

Dresden OCL Toolkit [WWW8]

Bereits zuvor wurde am Lehrstuhl für Softwaretechnologie der Technischen Universität Dresden im Rahmen mehrerer Diplom- und Belegarbeiten ([Fin00], [Fin99], [Wie00]) ein OCL-Compiler entwickelt.

Dieser basiert auf der OCL-Version 1.3 und ermöglicht es, OCL-Invarianten sowie Vor- und Nachbedingungen in UML-Modellen zu kompilieren. Des Weiteren kann aus ihnen Java-Code erzeugt werden, welcher zur Instrumentierung der aus den UML-Modellen generierten Java-Klassen genutzt wird. Dies ermöglicht eine Auswertung der Invarianten sowie Vor- und Nachbedingungen zur Laufzeit.

Der generierte Code nutzt hierbei die in [Fin99] entwickelte Basisbibliothek. Diese wird auch in der vorliegenden Arbeit wiederverwendet.

USE – A UML-based Specification Environment [WWW9]

USE ist ein Werkzeug, das im Rahmen von [Ric01] entstanden ist. Es basiert ebenfalls auf einem Metamodell für OCL und bietet u.a. folgende Funktionalität:

- Beschreibung von Modellen in textueller Form.
- Angabe von OCL-Ausdrücken für diese Modelle.
- Erstellen von Instanzen (Snapshots) der Modelle.
- Auswertung der OCL-Ausdrücke für diese Instanzen.

Da die beschriebenen Modelle prinzipiell auch Metamodelle sein können, lassen sich auf diese Weise auch WFRs überprüfen. In [Ric01] wird dieser Vorgang als Metavalidation bezeichnet.

Es bestehen also Ähnlichkeiten zwischen der Metavalidation und der hier vorgestellten Transformation von OCL-Ausdrücken über Metamodellen in Java-Code.

Folgende Unterschiede sind jedoch hervorzuheben:

- USE basiert nicht auf den OMG-Standards. Die Beschreibung der Modelle in textueller Form erfasst nur einen Teil der in UML und MOF vorhandenen Konzepte.
- Zur Auswertung der OCL-Ausdrücke wird in USE kein Code generiert. Es findet keine Kompilierung statt, sondern eine Interpretation der Ausdrücke.

1.4 Weiterer Aufbau dieses Dokuments

Kapitel 2 erläutert die Grundlagen der Metamodellierung entsprechend der Metadatenarchitektur der OMG. Außerdem werden Technologien zum Umgang mit Metamodellen und Modellen vorgestellt. Basierend auf der Einordnung des OCL-Metamodells in die MOF-Metadatenarchitektur werden die Anforderungen an Compiler und Code-Generator präzisiert. **Kapitel 3** beschreibt die auf einem Repository basierende Architektur des Compilers. Für dieses Repository werden in **Kapitel 4** vorhandene Produkte vorgestellt, aus denen eines für die Implementation des Compilers ausgewählt wird. In **Kapitel 5** wird zunächst die Integration von OCL-Metamodell und UML-Metamodell bzw. MOF erläutert. Anschließend werden weitere Entwurfsdetails des Compilers und insbesondere des Code-Generators beschrieben. **Kapitel 6** nennt die noch nicht implementierten OCL-Sprachkonzepte und zeigt mögliche Probleme auf. **Kapitel 7** fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf potentielle Folgearbeiten. **Anlage A – OCL-Metamodell** enthält die Klassendiagramme des für die Integration mit MOF und UML geänderten OCL-Metamodells. **Anlage B – Wellformedness Rules** spezifiziert die WFRs für dieses OCL-Metamodell. **Anlage C – Spezifikation der Operationen** beschreibt alle für das Metamodell definierten Operationen. **Anlage D – Spezifikation der Codegenerierung** enthält die in OCL spezifizierte Transformation für die Code-Generierung.

2 Grundlagen

Dieses Kapitel stellt die MOF-Metadatenarchitektur der Object Management Group (OMG) und die darauf basierenden Technologien XML[1] Metadata Interchange (XMI), Java Metadata Interface (JMI) und die Abbildung auf CORBA[2] Interface Definition Language (IDL) vor. Anschließend wird das OCL-Metamodell in die Metadatenarchitektur eingeordnet. Auf dieser Grundlage werden die Anforderungen an den Compiler und insbesondere an die Transformation von OCL-Metamodellinstanzen in Java-Code erläutert.

2.1 Die MOF-Metadatenarchitektur der OMG

Für die Formalisierung der Modellierungssprache UML findet in der Spezifikation [UML15] die Technik der Metamodellierung Anwendung. Hierunter versteht man die Beschreibung von Modellen mit Hilfe anderer Modelle.

UML-Modelle werden durch das UML-Metamodell beschrieben, d.h. UML-Modelle sind Instanzen des UML-Metamodells. Während UML-Modelle typischerweise Klassen wie „Person" oder „Firma" enthalten, findet man im UML-Metamodell z.B. die Klassen „Class", „Attribute" oder „State".

Auch das UML-Metamodell selbst ist wiederum Gegenstand der Beschreibung durch ein weiteres Modell, und zwar durch das MOF-Metametamodell. Die Meta Object Facility (MOF) ist ein weiterer Standard der Object Management Group [MOF14]. Er umfasst Konstrukte zur statischen Modellierung und ist einer Teilmenge des UML-Metamodells – dem so genannten Kern (Core) – ähnlich. So findet man in MOF wiederum die Klassen „Class" und „Attribute", nicht aber die Klasse „State", die zur Modellierung von Objektverhalten dient.

Aufgrund der Ähnlichkeiten zwischen MOF und UML wird die grafische UML-Notation auch für die Darstellung von Metamodellen verwendet.

Neben dem UML-Metamodell werden auch noch eine Reihe anderer Metamodelle mit Hilfe von MOF beschrieben, so z.B. das Common Warehouse Metamodel der OMG

[1] XML: E**x**tensible **M**arkup **L**anguage
[2] CORBA: **C**ommon **O**bject **R**equest **B**roker **A**rchitecture

[CWM10]. MOF selbst ist ebenfalls eine Instanz des MOF-Metametamodells, z.B. sind die Klassen „Class“ und „Attribute“ wiederum Instanzen von „Class“.

Aus Modellen, Metamodellen und dem MOF-Metametamodell ergibt sich, wie in Abbildung 2-1 dargestellt, eine vierschichtige Metadatenarchitektur. Die einzelnen Metaebenen werden dabei mit M0 bis M3 bezeichnet. Auf Ebene M0 befinden sich laut den OMG-Standards Nutzerobjekte bzw. Nutzerdaten. Eine nähere Spezifikation wird jedoch nicht vorgenommen. Im Falle der Implementierungstechnologie Java könnte man z.B. die Objekte, die zur Laufzeit erzeugt werden, der Metaebene M0 zuordnen.

Der Begriff Metadaten rührt daher, dass die Daten auf Ebene M0 durch andere Daten auf Ebene M1 beschrieben werden, letztere bezeichnet man als Metadaten. Ein Modell besteht demzufolge aus Metadaten. Ebenso findet man auf Ebene M2 Metametadaten.

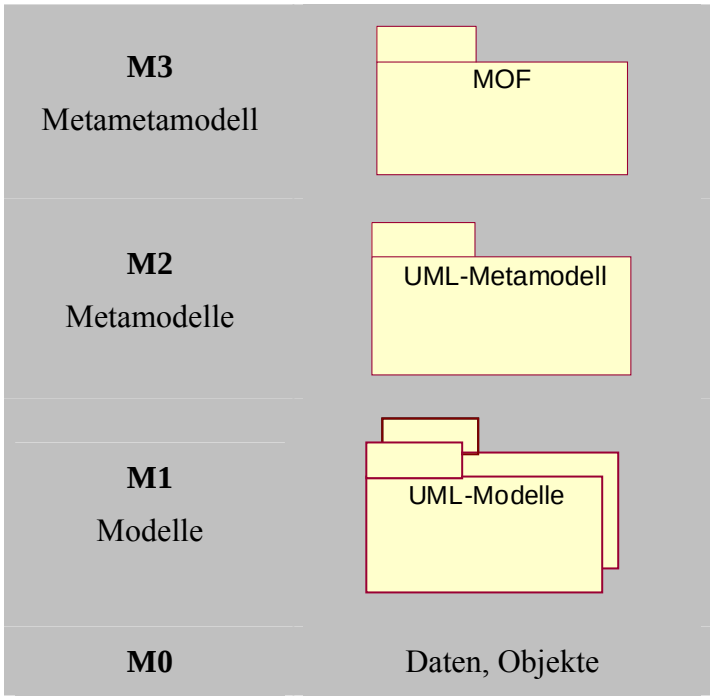


Abbildung 2-1: MOF-Metadatenarchitektur

Aufgrund der Tatsache, dass MOF als Instanz seiner selbst beschrieben wird, kann man es auch als Metamodell (M2) oder als Modell (M1) betrachten. Ebenso ist auch das UML-Metamodell ein Modell (M1).

2.2 XML Metadata Interchange

Auf Basis der MOF-Metadatenarchitektur wurde in [XMI12] standardisiert, wie Modelle (M1) zwischen Werkzeugen per XML ausgetauscht werden können. Mittels so

genannter DTD[1] Production Rules wird beschrieben, wie aus einem Metamodell (M2) eine DTD erzeugt wird. Document Production Rules beschreiben dagegen, wie ein Modell (M1) auf ein XML-Dokument abgebildet wird, welches mit Hilfe der DTD des entsprechenden Metamodells validiert werden kann. Abbildung 2-2 veranschaulicht dieses Vorgehen für ein UML-Modell.

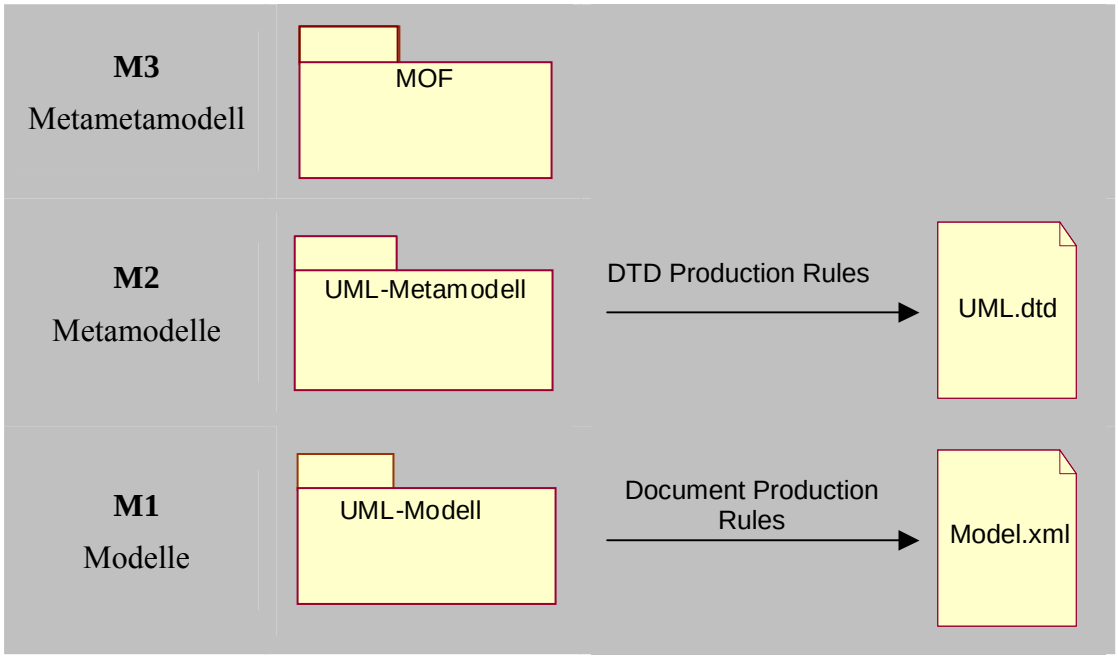


Abbildung 2-2: XMI für ein UML-Modell

Eine so erzeugte DTD beschreibt ein Metamodell nicht vollständig. Ein XML-Dokument, das mit Hilfe der DTD validiert wird, ist nicht zwangsläufig eine Instanz des Metamodells. Will man also Metamodelle zwischen Werkzeugen austauschen, ist die DTD als ungeeignet anzusehen. Da jedoch MOF eine Instanz von sich selbst ist, kann man jedes Metamodell auch als Modell betrachten und somit ein XML-Dokument aus ihm erzeugen, wie dies Abbildung 2-3 für das UML-Metamodell illustriert. Betrachtet man nun schließlich noch MOF als Modell auf Ebene M1, so erhält man das XML-Dokument für MOF.

Die XML-Dokumente für MOF und UML sind auf den Webseiten der OMG erhältlich [WWW1], [WWW2]. Hierbei sind folgende Versionen gegenwärtig aktuell:

- Das XML-Dokument für MOF enthält Version 1.4 von MOF als Instanz von sich selbst.

[1] DTD: **D**ocument **T**ype **D**escription

- Das XML-Dokument für UML enthält Version 1.5 als Instanz von MOF 1.3.

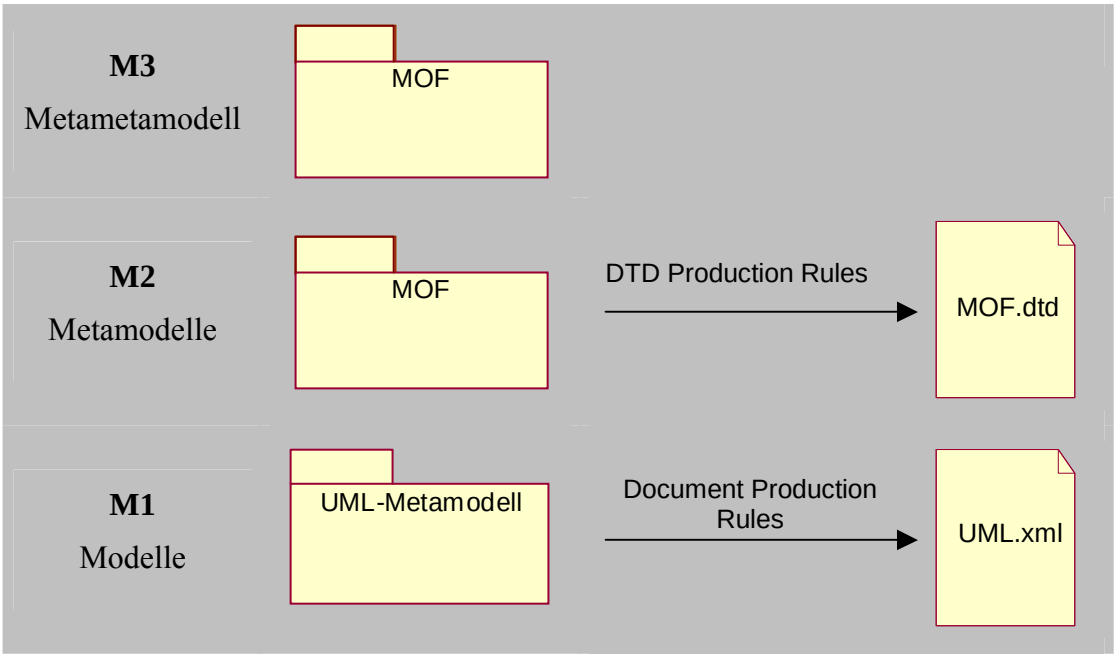


Abbildung 2-3: XMI für das UML-Metamodell

Werkzeugunterstützung

Während mittlerweile der größte Teil der vorhandenen UML-CASE[1]-Tools den Import und Export von UML-Modellen unterstützt, bietet gegenwärtig nur Rational Rose mit Unisys XMI Add-in [WWW3] diese Funktionalität auch für Metamodelle an.

Das Add-in ermöglicht es, ein Modell in Rose als Instanz von MOF, Version 1.3 zu exportieren oder zu importieren. Rose eignet sich also somit dafür, das OCL-Metamodell grafisch zu erstellen und – analog zu Abbildung 2-3 – ein XML-Dokument daraus zu erzeugen.

2.3 Java Metadata Interface und MOF-IDL-Abbildung

Während XMI eine lose Kopplung von Werkzeugen mittels Datenaustausch ermöglicht, handelt es sich bei JMI und MOF-IDL-Abbildung um zwei Technologien, die eine Datenteilung [Lis00] unterstützen, also den Zugriff von Werkzeugen auf Daten in einem gemeinsamen Repository. Dies heißt konkret, dass Abbildungen definiert werden, die für ein beliebiges Metamodell (M2) Schnittstellen erzeugen. Diese Schnittstellen kön-

[1] CASE: **C**omputer **A**ided **S**oftware **E**ngineering

nen dazu verwendet werden, auf Instanzen (M1) dieses Metamodells zuzugreifen oder diese zu manipulieren.

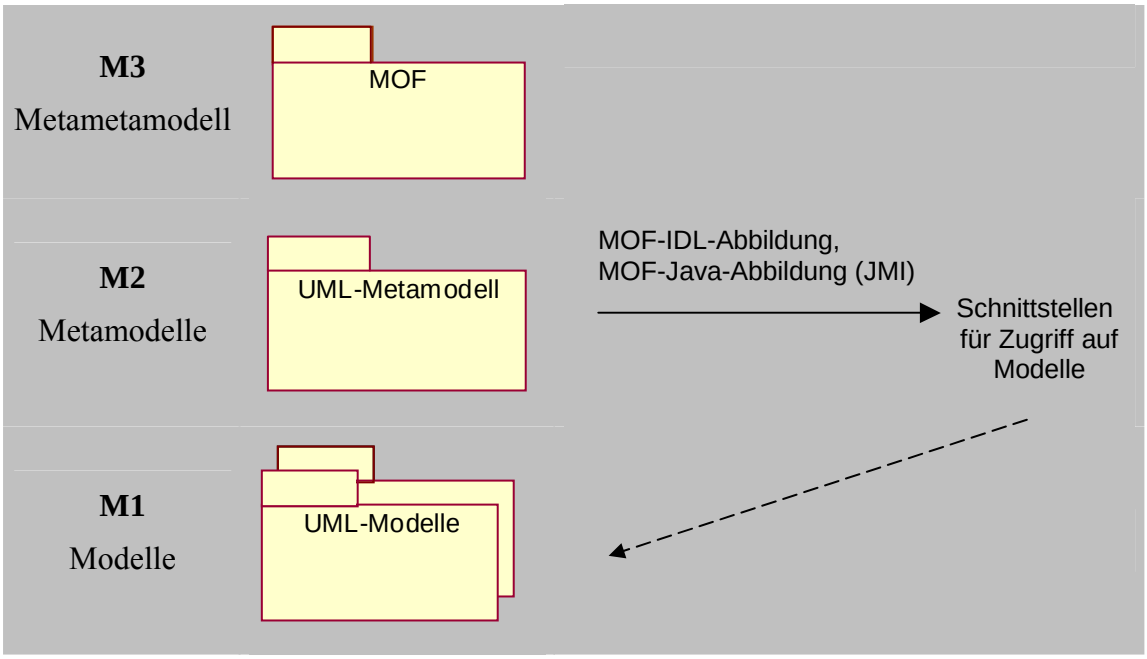


Abbildung 2-4: JMI und MOF-IDL-Abbildung

Die MOF-IDL-Abbildung ist Teil der MOF-Spezifikation [MOF14,S.5-1ff]. Bei JMI handelt es sich hingegen um eine eigenständige Spezifikation [JMI10]. JMI basiert auf der MOF Version 1.4.

Während die MOF-IDL-Abbildung CORBA-IDL Schnittstellen erzeugt, definiert JMI eine MOF-Java-Abbildung, die beschreibt, wie man aus einem Metamodell Java-Interfaces erzeugt. Im Folgenden soll anhand eines simplen Metamodells das Aussehen solcher Schnittstellen illustriert werden.

Beispiel für JMI Schnittstellen

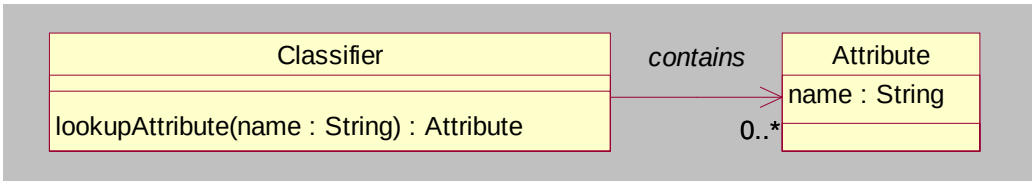


Abbildung 2-5: Ein simples Metamodell

Für das in Abbildung 2-5 dargestellte Metamodell liefert die MOF-Java-Abbildung hauptsächlich[1] folgende Schnittstellen:

Class Proxy Interfaces

- enthalten Fabrikmethoden zum Erzeugen von Instanzen der Klassen[2]

```
public interface ClassifierClass extends javax.jmi.reflect.RefClass {
    public Classifier createClassifier();
}
```

```
public interface AttributeClass extends javax.jmi.reflect.RefClass {
    public Attribute createAttribute();
    public Attribute createAttribute(String name);
}
```

Instance Interfaces

- enthalten Methoden zum Zugriff auf und zur Manipulation von Attributen und Assoziationen (sofern navigierbar)
- enthalten Methoden entsprechend den im Metamodell definierten Operationen wie `lookupAttribute()`

```
public interface Classifier extends javax.jmi.reflect.RefObject {
    public Attribute lookupAttribute(String name);
    public Collection getAttribute();[3]
}
```

```
public interface Attribute extends javax.jmi.reflect.RefObject {
    public String getName();
    public void setName(String newValue);
}
```

[1] Neben den dargestellten Schnittstellen für die beiden Klassen wird auch noch für die Assoziation selbst eine Schnittstelle generiert, auf die hier nicht näher eingegangen werden soll.
[2] Hier nicht dargestellt: Der Zugriff auf die Class Proxies selbst erfolgt über die für das umgebende Paket generierte Schnittstelle.
[3] Die Collection dient nicht nur zum Zugriff auf die Menge der assoziierten Attribute, sondern auch zur Manipulation dieser Menge.

Reflektion

Neben der Nutzung der metamodellspezifischen Schnittstellen ist es auch möglich, auf Modelle (M1) mittels Reflektion zuzugreifen. So erbt z.B. im Falle von JMI jedes Instance Interface von der Schnittstelle `RefObject`. Diese enthält u.a. die Methode `refGetValue()` zum Zugriff auf Attribute und Assoziationen und die Methode `refInvokeOperation()` zum Aufruf von Operationen. Zur Illustration stellt Tabelle 2-1 den metamodellspezifischen Methoden aus obigem Beispiel ihre reflektiven Äquivalente gegenüber:

	metamodellspezifisch	**reflektiv**
Classifier	`lookupAttribute(name)`	`refInvokeOperation("lookupAttribute", new Object[]{name})`
	`getAttribute()`	`refGetValue("Attribute")`
Attribute	`getName()`	`refGetValue("name")`
	`setName(newValue)`	`refSetValue("name", newValue)`

Tabelle 2-1: metamodellspezifische und reflektive Methoden

Semantik

Außer der Spezifikation der Abbildung von Metamodellen auf IDL- bzw. Java-Schnittstellen wird in der MOF- bzw. JMI-Spezifikation auch die Semantik für Modelle (M1) festgelegt [MOF14,S.5-6ff], [JMI10,S.40ff].

So wird zum Beispiel eine Instanz einer Assoziation des Metamodells als eine Menge von Links (Verbindungen) zwischen Instanzen von Klassen des Metamodells beschrieben. Die Semantik der entsprechenden (metamodellspezifischen oder reflektiven) Zugriffsmethoden für diese Assoziation wird basierend auf dieser Menge von Links spezifiziert.

Während die Semantik von Instanzen von Paketen, Klassen, Attributen und Assoziationen feststeht, kann eine solche für im Metamodell vorhandene Operationen wie `lookupAttribute()` im obigen Beispiel nicht im Rahmen der MOF- oder JMI-Spezifikation erfolgen. Vielmehr muss der Entwickler des Metamodells diese selbst festlegen und implementieren.

2.4 MOF-Repository

In den folgenden Kapiteln wird der Begriff **MOF-Repository** verwendet. Daher wird hier zunächst die Bedeutung des Begriffes im Rahmen dieser Arbeit festgelegt.

Ein **MOF-Repository** ist eine Softwarekomponente mit folgenden Funktionen und Eigenschaften:

(1) Laden und Speichern von Modellen und Metamodellen als XML-Dokumente entsprechend des XMI-Standards.

(2) Generierung von Java- und/oder IDL-Schnittstellen für Metamodelle entsprechend der MOF-IDL- bzw. MOF-Java-Abbildung.

(3) Generierung von Implementationen für diese Schnittstellen entsprechend der in der MOF- bzw. JMI-Spezifikation beschriebenen Semantik.

(4) Möglichkeit der Implementation von in Metamodellen definierten Operationen. Die Konventionen hierfür sind nicht standardisiert und demzufolge proprietär.

2.5 Das OCL-Metamodell in der MOF-Metadatenarchitektur

Abbildung 2-6 ordnet das OCL-Metamodell in die MOF-Metadatenarchitektur ein. Es ist aufgrund von Assoziationen und Vererbungsbeziehungen abhängig vom UML-Metamodell.

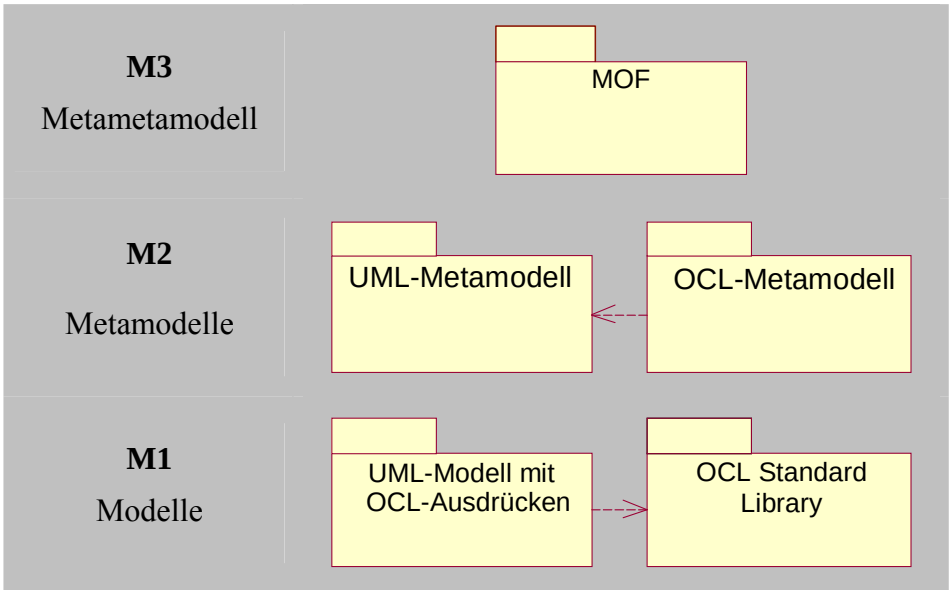


Abbildung 2-6: OCL-Metamodell in der MOF-Metadatenarchitektur

Neben dem OCL-Metamodell, welches sich auf Metaebene M2 befindet, wird in [OCL16,S.6-1ff] die OCL Standard Library beschrieben. Diese enthält u.a. vordefinierte primitive Datentypen wie `Integer`, `Boolean` oder `String` und die dafür zulässigen Operationen. Da diese Typen Instanzen der Klasse `Primitive` des UML-Metamodells sind, ist die OCL Standard Library der Metaebene M1 zuzuordnen.

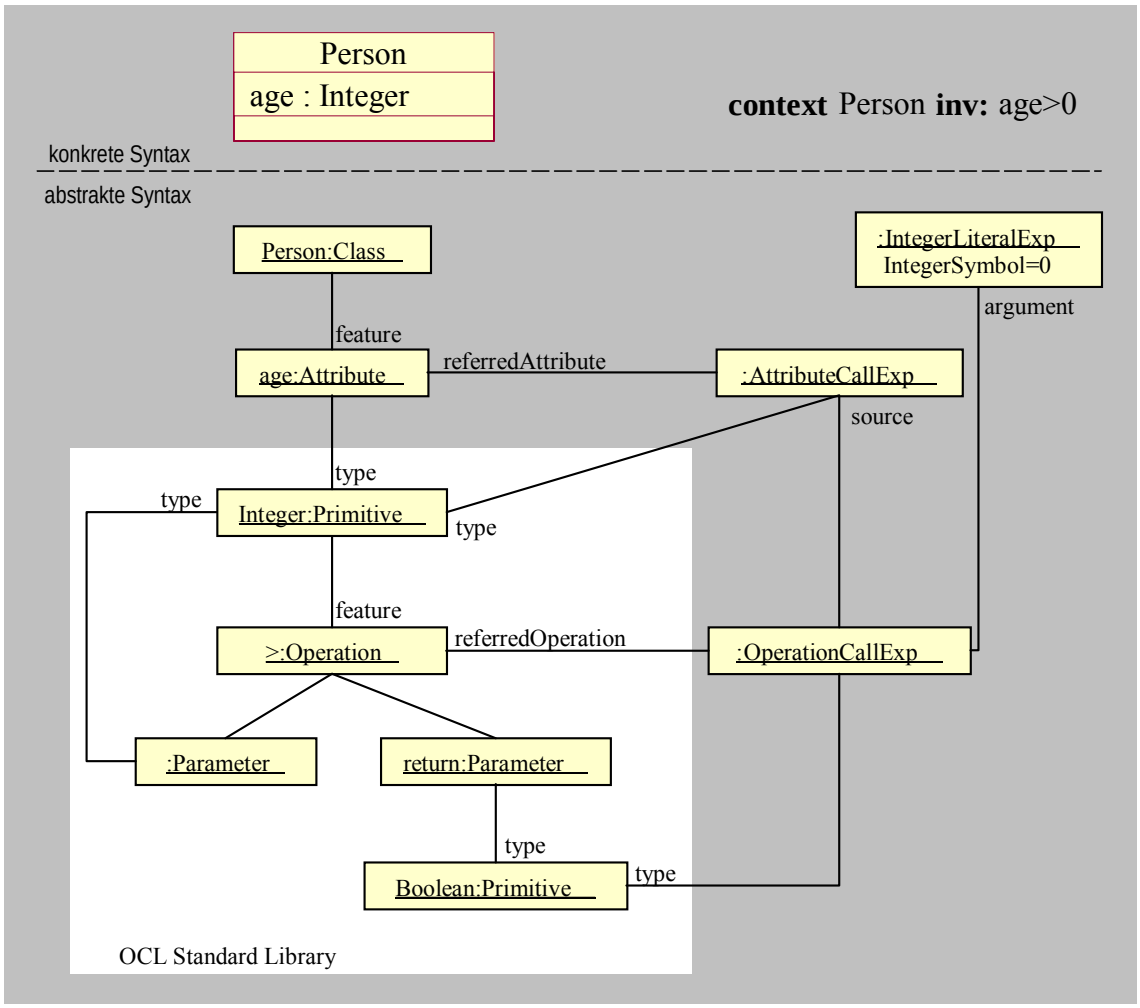


Abbildung 2-7: Ein OCL-Ausdruck als Instanz des OCL-Metamodells[1]

In Abbildung 2-7 wird eine OCL-Invariante für eine Klasse `Person` betrachtet. Während der obere Teil der Abbildung das UML-Modell in grafischer Notation und die OCL-Invariante in textueller Form – also in konkreter Syntax – darstellt, zeigt der untere Teil die Klasse `Person` als Instanz des UML-Metamodells und den Ausdruck `age>0` als Instanz des OCL-Metamodells.

[1] Es wurde eine verkürzte Notation für die Objekte angewandt: z.B. meint `Integer:Primitive` eine Instanz der Klasse `Primitive`, deren Attribut `name` den Wert „Integer“ hat. Das Attribut name erben alle Klassen des UML- und OCL-Metamodells von der Klasse `ModelElement`.

`Class`, `Attribute`, `Operation`, `Primitive` und `Parameter` sind Klassen des UML-Metamodells. Die Klassen `OperationCallExp`, `AttributeCallExp` und `IntegerLiteralExp` gehören zum OCL-Metamodell und haben die gemeinsame Oberklasse `OclExpression`. Der primitive Datentyp `Integer` und die Vergleichsoperation > sind Bestandteile der OCL Standard Library.

Nicht nur in UML-Modellen findet OCL Anwendung, sondern auch in Metamodellen. Beispiele hierfür sind die Wellformedness Rules (WFRs) in MOF, im UML-Metamodell und im OCL-Metamodell selbst [OCL16,S.3-17]. So wird beispielsweise für `AttributeCallExp` folgende Invariante formuliert:

```
context AttributeCallExp
inv: type = referredAttribute.type
```

Der Vergleich mit der Invariante in Abbildung 2-7 zeigt deutlich, dass dieser OCL-Ausdruck eine Metaebene höher einzuordnen ist. Folglich wird für seine Darstellung in abstrakter Syntax ein OCL-Metamodell für MOF benötigt. Aufgrund der Tatsache, dass der Kern (Core) von UML und MOF erst im Rahmen von UML 2.0 wirklich gleich sein wird, kann dieses OCL-Metamodell für MOF nicht einfach nur eine Teilmenge des OCL-Metamodells für UML sein.

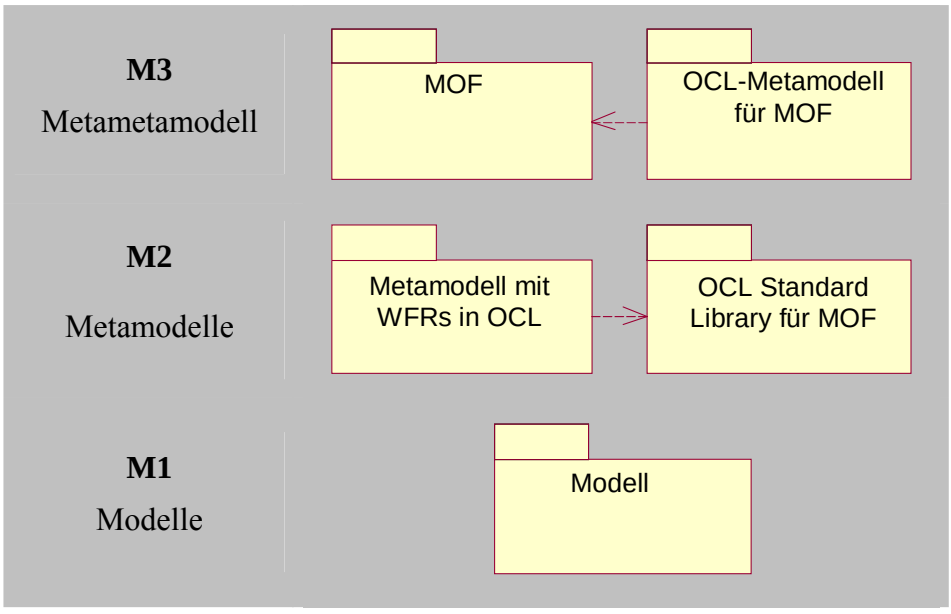


Abbildung 2-8: OCL-Metamodell für MOF

2.6 Anforderungen an den OCL-Compiler

Nach erfolgter Einordnung von OCL in die MOF-Metadatenarchitektur lassen sich nun die aus der Aufgabenstellung hervorgehenden Anforderungen an den OCL-Compiler präzisieren:

- OCL-Ausdrücke in konkreter Syntax müssen in Instanzen des OCL-Metamodells transformiert werden. Je nachdem, ob ein UML-Modell oder ein Metamodell zugrunde liegt, muss dabei das OCL-Metamodell für UML oder für MOF Verwendung finden.
- Der Code-Generator soll eine Transformation von OCL-Ausdrücken als Instanzen (M2) des OCL-Metamodells für MOF in Java-Code vornehmen. Dieser Java-Code soll die Auswertung der OCL-Ausdrücke für ein Modell (M1) durchführen.
- Der Code-Generator sollte möglichst gut wiederverwendbar sein für eine Transformation von OCL-Ausdrücken in UML-Modellen (M1) nach Java-Code, wie sie in [Fin00] beschrieben wird.

Schon alleine aus der ersten Anforderung folgt, dass für eine Implementierung des Compilers die Funktionalität eines MOF-Repository entsprechend Kapitel 2.4 benötigt wird. Um Instanzen des OCL-Metamodells erzeugen zu können, muss man es zunächst zusammen mit MOF bzw. mit dem UML-Metamodell per XMI laden. Anschließend werden unter Nutzung einer der beiden in Kapitel 2.3 vorgestellten Abbildungen Schnittstellen generiert.

Mit der Architektur des Compilers auf Basis eines MOF-Repository beschäftigt sich das folgende Kapitel.

3 Architektur des Compilers

Abbildung 3-1 zeigt das MOF-Repository als zentrale Komponente der Architektur des Compilers. Es enthält die Metamodelle für MOF, UML und OCL sowie deren Instanzen.

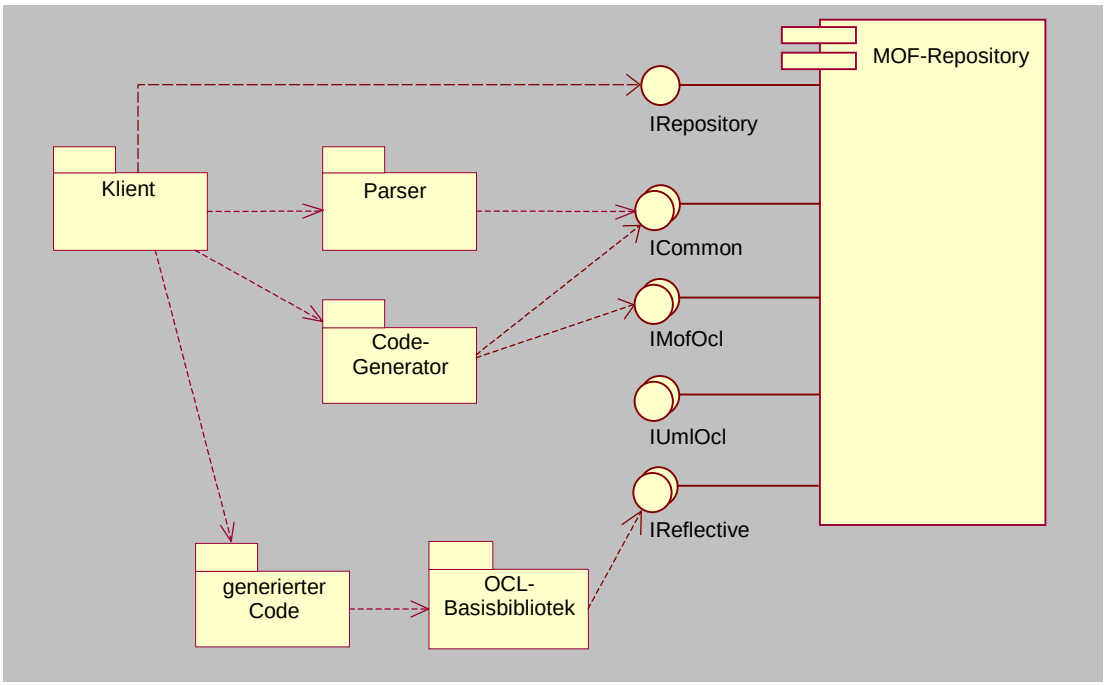


Abbildung 3-1: Architektur des Compilers

Die vom Repository angebotene Funktionalität lässt sich in verschiedene Gruppen von Schnittstellen unterteilen. In der Abbildung sind diese mit **`IRepository`**, **`ICommon`**, **`IMofOcl`**, **`IUmlOcl`** und **`IReflective`** bezeichnet.

- **`IRepository`** umfasst Funktionen zum Laden und Speichern von Modellen bzw. Metamodellen mittels XMI und Funktionen zur Generierung von IDL- oder Java-Schnittstellen aus Metamodellen. Die **`IRepository`** Schnittstelle ist nicht standardisiert. Dieses Problem soll im Rahmen von MOF 2.0 gelöst werden [MOF2FO].
- **`IMofOcl`** und **`IUmlOcl`** sind die aus MOF und dem UML-Metamodell – jeweils mit integriertem OCL-Metamodell – generierten IDL- oder Java-Schnittstellen. Details der Integration des OCL-Metamodells werden in Kapitel 5.1 diskutiert.
- Die mit **`ICommon`** bezeichnete Gruppe von Schnittstellen abstrahiert Gemeinsamkeiten von **`IMofOcl`** und **`IUmlOcl`**. Sie ermöglicht den Zugriff auf und die

Manipulation von OCL-Metamodellinstanzen in transparenter Art und Weise, d.h. unabhängig davon, ob es sich um OCL-Ausdrücke in Metamodellen oder in UML-Modellen handelt. Details zu **`ICommon`** werden in Kapitel 5.2 betrachtet.

- **`IReflective`** bezeichnet die in Kapitel 2.3 beschriebenen reflektiven Schnittstellen. Mit diesen ist es möglich, auf Modelle (M1) zuzugreifen, ohne für das entsprechende Metamodell Java- oder IDL-Schnittstellen generieren zu müssen.

Der **`Parser`** übersetzt für ein gegebenes Metamodell oder UML-Modell OCL-Ausdrücke in konkreter Syntax in OCL-Ausdrücke in abstrakter Syntax, erzeugt also im Repository Instanzen des OCL-Metamodells. Die Implementation des Parsers erfolgt nicht im Rahmen dieser Arbeit, sondern ist Gegenstand von [Kon03].

Der **`Code-Generator`** erwartet als Eingabe ein Metamodell **X** (M2) mit OCL-Ausdrücken, die als Instanz des OCL-Metamodells für MOF vorliegen. Er erzeugt Java-Code[1], der diese OCL-Ausdrücke für Modelle (M1) auswertet, die Instanzen des Metamodells **X** sind.

Hierbei nutzt der **`Code-Generator`** soweit wie möglich die Schnittstellen **`ICommon`**. Werden Informationen benötigt, die für OCL für MOF spezifisch sind, wird auch **`IMofOcl`** genutzt. Analog hierzu würde ein **`Code-Generator`** für OCL-Ausdrücke in UML-Modellen **`IUmlOcl`** verwenden. Die Details der Code-Generierung werden in Kapitel 5.6 beschrieben.

Der **`generierte Code`** nutzt die **`OCL-Basisbibliothek`**[2] aus [Fin99]. Diese greift mit Hilfe der Schnittstellen **`IReflective`** auf Modelle (M1) zu. Dies erfordert eine Anpassung der Bibliothek (siehe Kapitel 5.5).

Der **`Klient`** ist ein beliebiges System, das **`IRepository`** nutzt, um ein Metamodell **X** in das Repository zu laden, und den **`Parser`** und **`Code-Generator`** auf OCL-Ausdrücke in diesem Metamodell anwendet. Anschließend werden mittels **`IRepository`**

---

[1] Die Zielsprache Java folgt aus der Anforderung, die in Java implementierte Bibliothek aus [Fin99] zu nutzen.

[2] Der Begriff OCL-Basisbibliothek soll fortan für die in [Fin99] erstellte Klassenbibliothek in Java stehen. Diese stellt eine Implementation der in Kapitel 2.5 vorgestellten OCL Standard Library dar. Beide Begriffe müssen jedoch klar getrennt werden: Währen die OCL Standard Library als Teil eines Modells im Repository existiert und vordefinierte Typen und Operationen beschreibt, befindet sich die OCL-Basisbibliothek außerhalb des Repository.

Instanzen des Metamodells **X** in das Repository geladen. Für diese wird der generierte Code ausgeführt.

Für UML-Modelle und darin enthaltene OCL-Ausdrücke ist der letztgenannte Schritt nicht möglich, da die Metaebene für Instanzen (M0) von UML-Modellen nicht Teil des Repository ist. Vielmehr wird in diesem Fall der erzeugte Code in den Code eingefügt, den ein UML-CASE-Tool für das UML-Modell generiert hat (siehe [Wie00]). Eine Code-Generierung für OCL-Ausdrücke in UML-Modellen wird jedoch in dieser Arbeit nicht weiter betrachtet.

Vergleich mit der Architektur in [Fin00]

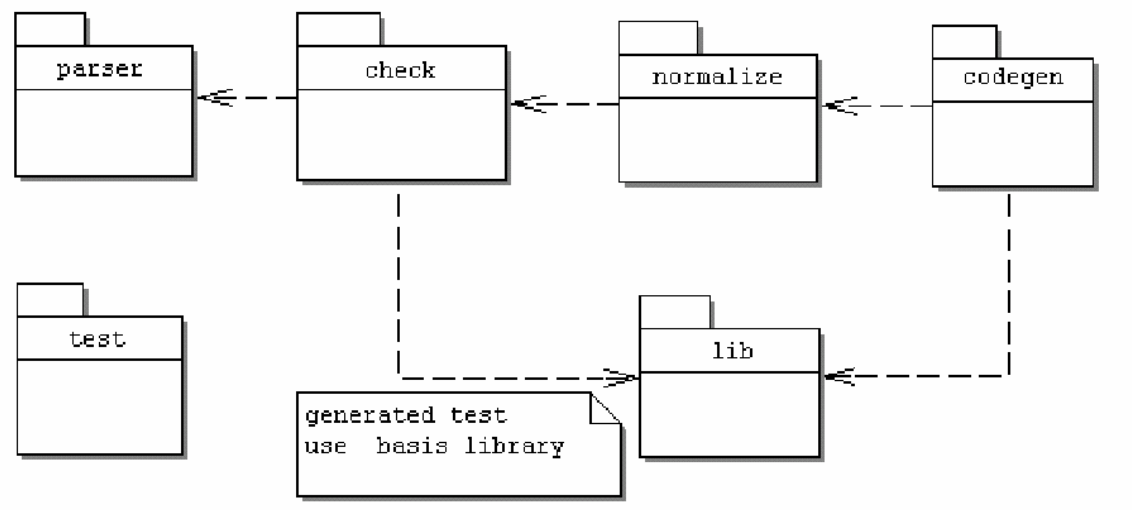


Abbildung 3-2: Architektur des OCL-Compilers in [Fin00]

Der OCL-Compiler in [Fin99] basiert auf einem abstrakten Syntaxbaum, der vom Parser erzeugt, anschließend vom so genannten Type-Checker mit Typinformationen versehen, dann normalisiert und schließlich als Eingabe des Code-Generators verwendet wird.

Die Rolle des abstrakten Syntaxbaumes übernimmt im metamodellbasierten Compiler das OCL-Metamodell. Eine Instanz des OCL-Metamodells gilt jedoch nur dann als wohlgeformt, wenn sie für jeden Teilausdruck Typinformation enthält:

Abbildung 3-3: Jeder OCL-Ausdruck hat einen Typ.

Folglich ist das Ermitteln der Typinformation in der hier vorgestellten Architektur Aufgabe des Parsers. In Kapitel 5.3 wird der **`TypeEvaluator`** vorgestellt, der den Typ einer `OclExpression` entsprechend den Wellformedness Rules ermittelt. Dieser kann vom Parser zur Erfüllung dieser Aufgabe genutzt werden.

Ebenfalls dem Parser zuzuordnen ist die Normalisierung. Ein Teilschritt der Normalisierung in [Fin99] ist z.B. das Hinzufügen eines expliziten `self` an Stellen des Ausdrucks, wo dies ausgelassen wurde. In den Abbildungsregeln von konkreter nach abstrakter Syntax in [OCL16] werden implizite Kontexte mittels entsprechender Fallunterscheidung separat behandelt. Die abstrakte Syntax enthält grundsätzlich nur explizite Referenzen auf Variablen, auch im Falle von `self`.

4 Wahl einer MOF-Repository Implementation

In diesem Kapitel wird für das MOF-Repository, welches die zentrale Komponente der im vorigen Kapitel vorgestellten Architektur des Compilers ist, eine Implementation ausgewählt. Zunächst werden für diese Auswahl Kriterien aufgestellt. Anschließend werden die betrachteten Produkte vorgestellt. Anhand der Kriterien wird schließlich die Entscheidung für eine Implementation getroffen..

4.1 Kriterien

Die Auswahl einer MOF-Repository Implementation erfolgte entsprechend folgender Kriterien:

1. Es werden nur Produkte betrachtet, welche die in Kapitel 2.4 geforderte Funktionalität besitzen.
2. Repositories, die **JMI** unterstützen, werden solchen vorgezogen, die nur die MOF-IDL-Abbildung beherrschen. Der Vorteil von CORBA IDL ist die Unabhängigkeit von der Programmiersprache. Der Vorteil von JMI hingegen ist die einfache Benutzung der Schnittstellen in Java ohne zusätzliche Kenntnis des entsprechenden CORBA Language Mapping. Insbesondere unter der Prämisse der Aufgabenstellung, dass die Transformation der OCL-Ausdrücke Java als Zielsprache hat, ist JMI die bessere Wahl. Die Abbildungsregeln beim Zugriff auf die jeweilige Metamodellinstanz müssen sich bei JMI nicht an der Konkatenation von zwei Transformationen (MOF-IDL-Abbildung und CORBA Language Mapping) orientieren, sondern können sich nach der direkten Transformation von MOF nach Java richten.
3. Ein weiteres Kriterium ist die unterstützte **MOF-Version**. Es sollte z.B. möglich sein, das UML-Metamodell zu laden, dass als XML-Dokument als Instanz von MOF 1.3 vorliegt (siehe Kapitel 2.2).
4. Das Produkt sollte **frei verfügbar** sein. Insbesondere im Falle der recht jungen JMI-Technologie ist Open-Source wünschenswert.
5. Auf gute **Dokumentation** wird Wert gelegt.

4.2 Evaluierte Implementationen

Im Folgenden sollen nun die wesentlichen Eigenschaften von vier MOF-Repository-Implementationen erläutert werden. Betrachtet wurden dMof 1.1 von DSTC [WWW4], das NetBeans Metadata Repository von Sun Microsystems [WWW5], CIM 1.3 von Unisys [WWW6] und das Novosoft Metadata Framework [WWW7].

DMof 1.1 von Distributed Systems Technology Centre (DSTC)

- Unterstützt CORBA IDL, jedoch kein JMI.
- MOF-Version 1.3, XMI-Version 1.1.
- Gute Dokumentation.
- Unterstützung (Support) und Weiterentwicklung wurden im Januar 2002 eingestellt aufgrund des geringen Interesses an dem Produkt von Seiten der Wirtschaft.
- Persistenz realisiert per JDBC-angebundener Datenbank.
- Lizenz: 30 Tage Evaluierung.

NetBeans Metadata Repository

- Integriert in die Entwicklungsumgebung NetBeans, autonomer Betrieb ist jedoch auch möglich. Künftige Versionen der NetBeans IDE werden wahrscheinlich sogar auf dem Metadata Repository basieren.
- Unterstützt JMI, nicht aber CORBA-IDL-Abbildung.
- Intern wird MOF 1.4 verwendet, es können jedoch auch Metamodelle geladen werden die als Instanz von MOF 1.3 vorliegen.
- XMI 1.1 (lesen) und 1.2 (lesen und schreiben).
- Mittelmäßige Dokumentation, aber sehr gute Unterstützung: Die Entwickler beantworten Fragen in der Mailing-Liste meist innerhalb eines Tages.
- Sun Public License, Open Source.
- Modelle persistent in integrierter Datenbank oder transient im Hauptspeicher.

CIM 1.3 von Unisys

- JMI-Referenzimplementation.
- Unterstützt nur MOF 1.4. und XMI 1.2. Das UML-Metamodell, welches als Instanz von MOF 1.3 vorliegt, kann somit nicht geladen werden.
- Grafische Workbench zum Laden und Speichern von Metamodellen und Modellen.
- Verwaltung des Repository über JNDI.
- Persistenz ausschließlich in Form von XMI Files.
- Mittelmäßige Dokumentation, keine Unterstützung.
- Open Source.
- Kommerzielle Version erhältlich: Adaptive Repository Enterprise Edition mit CORBA IDL und JDBC Support. Evaluierung jedoch nicht möglich.

Novosoft Metadata Framework

- Unterstützt JMI , MOF 1.4 und XMI 1.1.
- Mangelhafte Dokumentation.

4.3 Zusammenfassung und Entscheidung

	NetBeans Metadata Repository	Unisys CIM1.3	DSTC dMof 1.1	Novosoft Metadata Framework
MOF	1.3, 1.4	1.4	1.3	1.4
XMI	1.1	1.2	1.1	1.1
JMI	ja	ja	nein	ja
CORBA IDL	nein	nein	ja	nein
Dokumentation und Unterstützung	++	+-	+	--
Lizenz	Open Source	Open Source	30 Tage	Open Source

Tabelle 4-1: Ergebnisse der Evaluierung von MOF-Repositories

Entscheidung:

Es wird das **NetBeans Metadata Repository** verwendet. Ausschlaggebend waren die Unterstützung von JMI, MOF 1.3 und 1.4 und die hervorragende Unterstützung.

Sämtliche Schnittstellen des Repository sind also Java-Schnittstellen. Die OCL-Basisbibliothek muss derart angepasst werden, dass sie die im JMI-Standard spezifizierten reflektiven Schnittstellen benutzt.

5 Detailentwurf

Dieses Kapitel beschreibt die Entwurfsdetails des metamodellbasierten OCL-Compilers. Zunächst setzt sich Unterkapitel 5.1 mit der Integration des OCL-Metamodells mit MOF und mit dem UML-Metamodell auseinander. Anschließend wird in Unterkapitel 5.2 die Struktur eines gemeinsamen OCL-Metamodells für MOF und UML erläutert. Unterkapitel 5.3 beschreibt Änderungen an den Wellformedness Rules des OCL-Metamodells und die Implementierung einer Typinferenz. In Unterkapitel 5.4 wird die in der OCL-Spezifikation beschriebene OCL Standard Library für UML präzisiert und die an MOF angepasste Variante vorgestellt. Unterkapitel 5.5 erläutert die Änderungen an der OCL-Basisbibliothek, die vorgenommen wurden, um einen Zugriff auf das Modell per JMI-Reflektion zu unterstützen. Die auf dieser angepassten Basisbibliothek aufbauende Code-Generierung ist das Thema von Unterkapitel 5.6. In Unterkapitel 5.7 wird eine grafische Oberfläche für die Auswertung von OCL-Ausdrücken in Metamodellen vorgestellt.

5.1 Integration der Metamodelle

Aufgrund von Abhängigkeiten zwischen OCL- und UML-Metamodell ist es nötig, beide Metamodelle zu integrieren und als ein einziges im Repository zu halten. Im Folgenden werden zunächst diese Abhängigkeiten näher erläutert. Anschließend werden Techniken zur Integration der Metamodelle beschrieben und Besonderheiten bei der Integration des OCL-Metamodells mit MOF erläutert.

5.1.1 Abhängigkeiten zwischen UML- und OCL-Metamodell

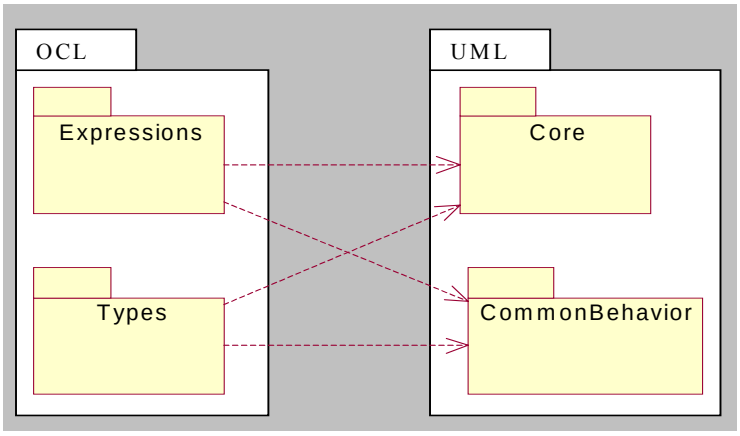


Abbildung 5-1: Abhängigkeiten zwischen OCL- und UML-Metamodell

Das OCL-Metamodell setzt sich aus den Paketen `Expressions` und `Types` zusammen. Während `Expressions` die Klasse `OclExpression` und ihre Unterklassen enthält, werden in `Types` Klassen definiert, deren Instanzen OCL-Typen wie z.B. Tupeltypen oder Kollektionstypen sind.

Die Abhängigkeiten vom UML-Metamodell betreffen hauptsächlich dessen `Core`-Paket aber auch das Paket `CommonBehavior`.[1]

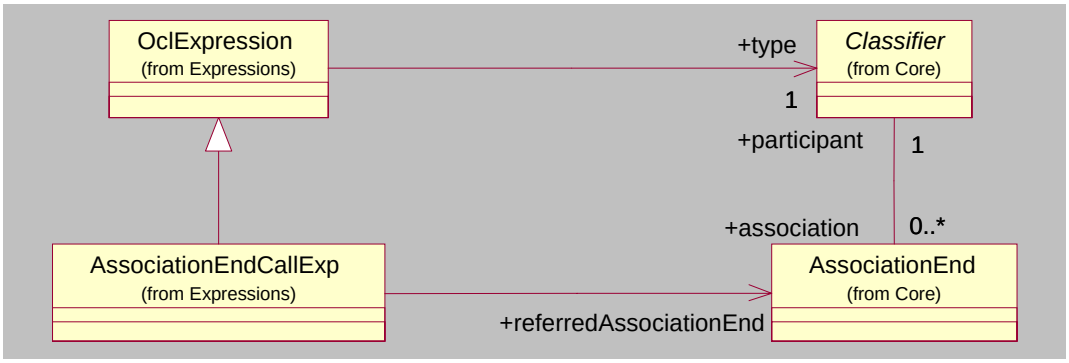


Abbildung 5-2: Abhängigkeiten durch Assoziationen

In `Expressions` besteht diese Abhängigkeit vor allem aufgrund von Assoziationen, wie z.B. zwischen `AssociationEndCallExp` und `AssociationEnd` (Abbildung 5-2), in `Types` dagegen vor allem aufgrund von Vererbungsbeziehungen zu Klassen wie `Classifier` und `DataType` des UML-Metamodells (Abbildung 5-3).

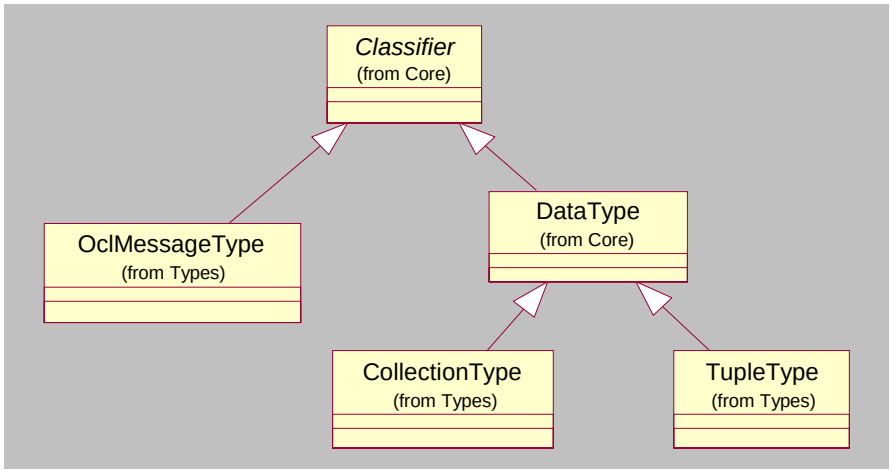


Abbildung 5-3: Abhängigkeiten durch Vererbung

[1] Für eine Übersicht über alle Pakete des UML-Metamodells siehe z.B. [UML15, S.5-3]

5.1.2 Integrationstechniken

Integration der Metamodelle in Rational Rose

Wie in Kapitel 2.2 beschrieben, ist es mit dem Unisys XMI Add-in möglich, Metamodelle, die als XML-Dokument vorliegen, in Rose zu importieren. Es ist jedoch nicht möglich, mittels dieses Add-ins zu einem bereits bestehenden Modell ein weiteres – als XML-Dokument gegebenes – hinzuzufügen. Für einen solchen Zweck existiert aber in Rose ein Werkzeug namens **Model Integrator**, welches es ermöglicht, mehrere Rose-Modelle (*.mdl-Dateien) zu einem einzigen zusammenzuführen.

Somit ergibt sich folgender möglicher Ablauf für eine Integration für OCL- und UML-Metamodell bzw. MOF:

1. Erstellen und Speichern des OCL Metamodells in Rose.
2. Import von UML-Metamodell/MOF als XML-Dokument [WWW1][WWW2].
3. Speichern von UML-Metamodell/MOF als Rose-Modell.
4. Zusammenführen der Modelle mit Hilfe des Model Integrator.
5. Künftige Änderungen am OCL-Metamodell werden am Ergebnis der Integration durchgeführt, um diese nicht wiederholen zu müssen.

Dieses Vorgehen weist jedoch einen entscheidenden Nachteil auf:

MOF 1.4 kann auf diese Weise nicht mit dem OCL-Metamodell zusammengeführt werden, da das Unisys XMI Add-in nur Metamodelle unterstützt, die Instanzen von MOF 1.3 sind. MOF 1.4 liegt jedoch als Instanz von MOF 1.4 vor [WWW2].

Da in Kapitel 4.3 entschieden wurde, ein JMI-Repository einzusetzen und da JMI auf der MOF-Version 1.4 basiert, ist es erforderlich, dass das OCL-Metamodell und MOF 1.4 integriert werden. Es wurde daher die Entscheidung getroffen, die Integration der Metamodelle erst im Repository selbst durchzuführen. Diese Vorgehensweise wird im nächsten Abschnitt erläutert.

Integration der Metamodelle im Repository

Der grundlegende Ablauf für eine Integration im Repository lässt sich wie folgt zusammenfassen:

- Laden von UML-Metamodell/MOF in das Repository.

- Laden des OCL-Metamodells in das Repository.
- Herstellen der Verbindungen (Generalisierungsbeziehungen und Assoziationen) zwischen den Klassen beider Metamodelle.

Der letzte Punkt wirft die Frage auf, woher die Informationen zu beziehen sind, welche Klasse mit welcher zu verbinden ist. Anstatt nun ein zusätzliches Artefakt einzuführen, dass diese Informationen enthält, werden diese in das OCL-Metamodell eingebracht. Dies wird durch die Einführung sogenannter **Metaproxies** realisiert.

Metaproxies sind Klassen im OCL-Metamodell, die als Stellvertreter für die Klassen von MOF bzw. des UML-Metamodells fungieren. Alle Assoziationen und Vererbungsbeziehungen zwischen OCL-Metamodell und UML-Metamodell bzw. MOF werden zunächst zwischen den Klassen des OCL-Metamodells und den Metaproxies modelliert. Während der Integration im Repository werden die Metaproxies gelöscht. Die an ihnen endenden Assoziationen und Vererbungsbeziehungen werden nun zu den adäquaten Klassen im UML-Metamodell bzw. in MOF hinzugefügt.

Die Wahl der Bezeichnung Metaproxies rührt daher, dass es sich nicht um Stellvertreter auf Instanzebene (hier M1), sondern auf Ebene der Metadaten (hier M2) handelt.

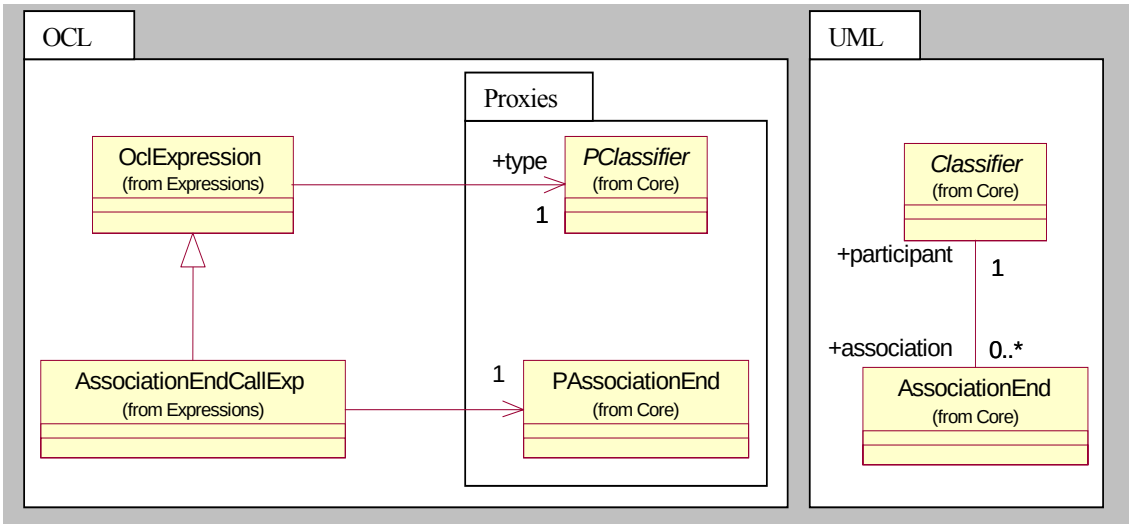


Abbildung 5-4: Metaproxies

Das Konzept der Metaproxies wird in Abbildung 5-4 illustriert. Die Metaproxies residieren in einem separaten Paket des OCL-Metamodells namens `Proxies`. Sie

tragen den Namen der zu stellvertretenden Klasse[1]. Im Rahmen der Metamodellintegration geschieht nun Folgendes:

- Laden von OCL-Metamodell und UML-Metamodell bzw. MOF in das Repository.
- Auffinden des Proxy-Paketes und Ersetzung aller darin enthaltenen Klassen durch die „originalen" Klassen aus dem UML-Metamodell bzw. aus MOF.

Das Ergebnis der Integration für das dargestellte Beispiel ist in Abbildung 5-2 zu sehen.

5.1.3 Integration von MOF und OCL-Metamodell

Im Grundlagenkapitel wurde in Abbildung 2-8 zunächst vereinfachend angenommen, dass MOF und OCL-Metamodell auf Metaebene M3 integriert werde könnten. Dies ist jedoch nicht möglich. Das MOF-Metametamodell auf Ebene M3 ist fester Bestandteil des Repository. Auf ihm basiert z.B. die Implementation von XMI und JMI. Eine Manipulation dieses „nativen" MOF ist nicht möglich.

Daher kann die Integration von MOF und OCL-Metamodell nur wie folgt durchgeführt werden:

- MOF wird explizit als Instanz des „nativen" MOF in das Repository geladen (nun: Metaebene **M2**).
- Das OCL-Metamodell wird hinzugeladen und die Integration wird durchgeführt.

Die Integration auf Metaebene M2 hat folgende Konsequenzen (Abbildung 5-5):

- Ein Metamodell **X**, für das OCL-Ausdrücke kompiliert werden sollen, muss nun auf Metaebene M1 geladen werden.
- Soll der für die OCL-Ausdrücke generierte Code nun für eine Instanz von **X** ausgeführt werden, so muss X zusätzlich auf Ebene M2 als Instanz des „nativen" MOF ins Repository geladen werden. (Denn nur so kann es instantiiert werden, da es in einem MOF-Repository keine Metaebene M0 gibt.)

[1] Zur besseren Unterscheidung mit dem Präfix „P" versehen.

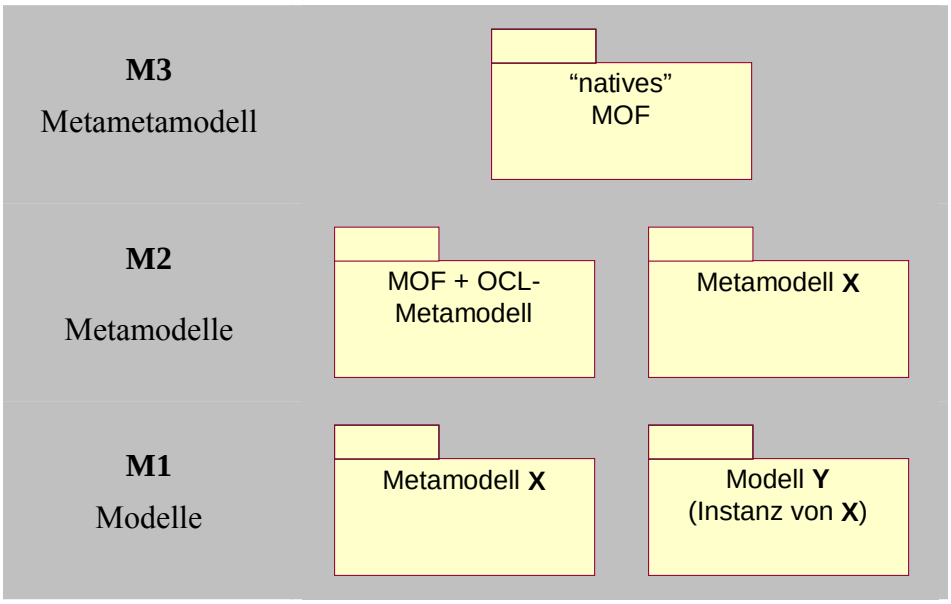


Abbildung 5-5: Integration von MOF und OCL-Metamodell

5.1.4 Einordnung der Metamodellintegration in die Architektur

Die Integration der Metamodelle im Repository ist eine Manipulation von Metamodellen (M2): Klassen des OCL-Metamodells – nämlich die Metaproxies – werden durch andere ersetzt. Für die Manipulation von Metamodellen M2 müssen die Schnittstellen genutzt werden, die für MOF (Ebene M3) selbst generiert wurden. Jedes MOF-Repository enthält von Beginn an MOF und verfügt somit über diese Schnittstellen. In Abbildung 5-6 werden sie mit `IMof` bezeichnet. Diese sind nicht zu verwechseln mit den in Kapitel 3 geforderten Schnittstellen `IMofOcl`, die durch Integration von OCL-Metamodell und MOF auf Metaebene M2 entstehen. Neben `IMof` wird auch die Schnittstelle `IRepository` genutzt. Diese wird für das Laden der Metamodelle und für das Speichern des Ergebnisses der Integration benötigt.

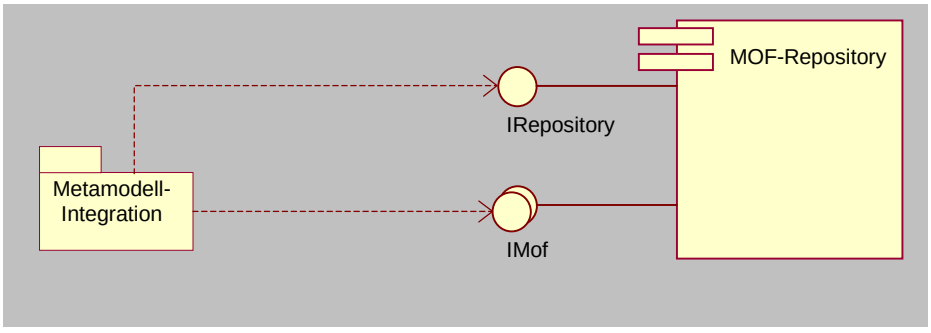


Abbildung 5-6: Metamodell-Integration in der Architektur des Compilers

5.2 Gemeinsames OCL-Metamodell für UML und MOF

Dieses Unterkapitel beschreibt den Entwurf eines gemeinsamen OCL-Metamodells **Common-OCL** für UML und MOF. Aus diesem gemeinsamen Metamodell werden per MOF-Java-Mapping die Schnittstellen `ICommon` (siehe Kapitel 3) generiert, die dem Parser ermöglichen, in gleicher Art und Weise auf Metamodelle und UML-Modelle zuzugreifen und über diesen OCL-Ausdrücke in abstrakter Syntax zu erzeugen.

In Abschnitt 5.2.1 werden die Unterschiede zwischen dem Core des UML-Metamodells und MOF beschrieben, die die Ursache dafür sind, dass sich das in [OCL16] spezifizierte Metamodell nicht ohne Änderungen auf MOF anwenden lässt. Abschnitt 5.2.2 beschreibt das Paket `CommonModel`, welches eine Abstraktion für die von OCL benötigten Klassen des MOF- bzw. UML-Metamodells darstellt. Abschnitt 5.2.3 befasst sich mit Adaptern für einige Klassen in MOF. Außerdem wird, basierend auf einem Adapter für die Klasse `DataType`, die Abbildung von MOF-Datentypen auf OCL-Typen beschrieben. Abschnitt 5.2.5 erläutert, wie sich Teile des gemeinsamen OCL-Metamodells generieren lassen. In Abschnitt 5.2.6 werden Hilfsklassen beschrieben, die dem Metamodell hinzugefügt wurden. Abschnitt 5.2.7 stellt abschließend noch einmal die Gesamtstruktur des gemeinsamen OCL-Metamodells vor.

5.2.1 Unterschiede zwischen UML-Kern und MOF

Für die meisten Klassen des UML-Metamodells, die im OCL-Metamodell referenziert werden, gibt es zwar gleichnamige Klassen in MOF, jedoch können diese nicht in jedem Falle in gleicher Weise verwendet werden. So gilt in MOF z.B., dass nur Instanzen der Metaklasse `Class` Operationen haben dürfen (siehe [MOF14,S.3-15]), während dies in UML für `Classifier` und alle Unterklassen möglich ist, insbesondere auch für `DataType` und `PrimitiveType`. Im Falle von MOF sollte also im OCL-Metamodell `Class` anstatt `Classifier` Verwendung finden.

Für die UML-Metaklasse `EnumerationLiteral`, die von der OCL-Metaklasse `EnumLiteralExp` referenziert wird [OCL16,S.3-16], findet sich keine Entsprechung in MOF. Wohl aber wird dort das Konzept der `Enumeration` an sich durch die Metaklasse `EnumerationType` unterstützt, die die Literale als Liste von Zeichenketten enthält [MOF14,S.3-35]. Für einen solchen Fall ist es nötig, für `EnumerationLiteral` einen entsprechenden Adapter einzuführen. Aber auch die Klasse `EnumerationType`

selbst kann nicht verwendet werden, da ihre Instanzen keine Operationen besitzen dürfen. Da es aber in OCL z.B. durchaus möglich ist, Literale einer `Enumeration` auf Gleichheit zu prüfen, muss für eine `Enumeration` u.a. die Operation „=“ definiert sein.

Während für `EnumerationLiteral` noch ein adäquates Konzept in MOF existiert, gibt es auch Klassen, für die das nicht der Fall ist, z.B. `AssociationClass`.

Die anhand von Beispielen aufgezeigten Unterschiede und dadurch implizierten Änderungen für MOF lassen sich wie folgt zusammenfassen: Für einige Klassen des UML-Metamodells, die im OCL-Metamodell referenziert werden ...

(1) gibt es in MOF adäquate, aber ungleichnamige Klassen

(2) wird in MOF ein Adapter benötigt

(3) gibt es in MOF keine gleichwertigen Konzepte.

Die Fälle (1) und (3) werden in Abschnitt 5.2.2 behandelt, Fall (2) wird in Abschnitt 5.2.3 näher betrachtet.

Die gravierendsten Differenzen zwischen UML-Metamodell und MOF bestehen jedoch bezüglich der Assoziationen zwischen den Klassen. Während einige Klassen zumindest noch namentlich und bezüglich mancher Attribute in MOF und im UML-Metamodell übereinstimmen, so sind doch ihre Beziehungen untereinander sehr verschieden modelliert.

Vorrangig in den Wellformedness Rules des OCL-Metamodells, aber auch in den Regeln für die Abbildung konkreter auf abstrakte Syntax werden OCL-Ausdrücke verwendet, die entlang von Assoziationen des UML-Metamodells navigieren. Solche Regeln sind folglich nicht ohne Änderung für MOF anwendbar. Im nächsten Abschnitt wird u.a. auch beschrieben, wie vom Zugriff auf Assoziationen des UML-Metamodells mittels Einführung zusätzlicher Operationen in den Klassen des Paketes `CommonModel` abstrahiert werden kann. Darauf aufbauend wird in Unterkapitel 5.3 beschrieben, wie die Wellformedness Rules des OCL-Metamodells zu ändern sind, um sie auf UML und MOF gleichermaßen anwenden zu können.

5.2.2 Das Paket `CommonModel`

Um einen einheitlichen Zugriff auf Metamodelle und UML-Modelle zu ermöglichen, ist es nötig, die im vorigen Abschnitt erläuterten Unterschiede zwischen UML-Metamodell

und MOF mittels einer Abstraktion zu verbergen. Diese Abstraktion wird durch das Paket `CommonModel` realisiert. Wie Abbildung 5-7 zeigt, sind die Pakete `Expressions` und `Types` des gemeinsamen OCL-Metamodells `Common-OCL` nicht mehr direkt vom UML-Metamodell abhängig, sondern von `CommonModel`.

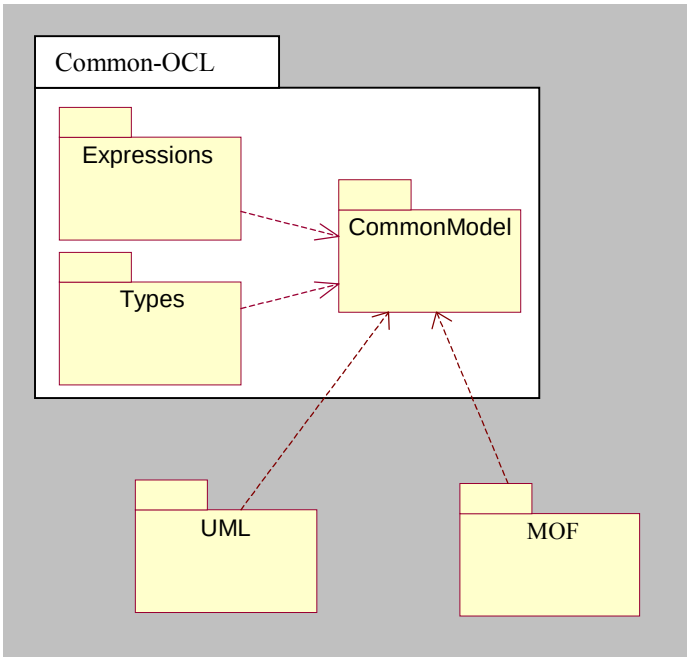


Abbildung 5-7: Das Paket CommonModel

Die Abhängigkeiten zwischen `MOF` bzw. `UML` und `CommonModel` stehen für Vererbungsbeziehungen. Dies wird in Abbildung 5-8 illustriert, welche erneut die Assoziation zwischen `OclExpression` und `Classifier` aufgreift.

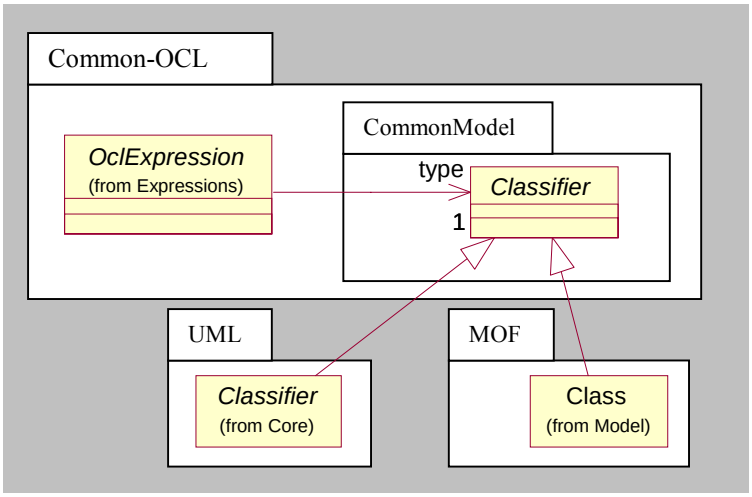


Abbildung 5-8: OclExpression und Classifier

Während in [OCL16] die Klasse `Classifier` des UML-Metamodells direkt von `OclExpression` referenziert wird, tritt an ihre Stelle nun die abstrakte Oberklasse `Classifier` in `CommonModel`. Im vorigen Abschnitt wurde erläutert, warum in `MOF` nicht `Classifier`, sondern `Class` die adäquate Klasse zu `Classifier` in `UML` ist. Dementsprechend ist die Vererbungsbeziehung zwischen `CommonModel` und `MOF` gestaltet.

Für Klassen wie `AssociationClass`, die keine Entsprechung in MOF haben, gibt es auch keine Vererbungsbeziehung zwischen `CommonModel` und `MOF`.

Während der obige Entwurf genügt, um Assoziationen zwischen dem OCL- und dem UML-Metamodell zu behandeln, ist er für Vererbungsbeziehungen unzureichend. Um dies näher zu erläutern, wird die Klasse `CollectionType` im Paket `Types` betrachtet. Diese ist laut [OCL16,S.3-2] Unterklasse von `UML::Core::Classifier`.[1] Dadurch wird z.B. die Eigenschaft von `Classifier`, Operationen besitzen zu können, an alle OCL-Kollektionstypen vererbt.

Modelliert man nun die Vererbungsbeziehung zwischen `CollectionType` und `Classifier` analog zu Abbildung 5-8, so erhält man die in Abbildung 5-9 dargestellte Struktur[2].

Es zeigt sich, das auf diese Weise nicht erreicht wird, dass eine Instanz von `CollectionType` Operationen besitzen kann.

Eine mögliche Lösung des Problems wäre, zwischen `Classifier` und `Operation` in `CommonModel` eine Assoziation einzuführen. Dies würde jedoch auch nicht – wie eigentlich beabsichtigt – zu einer Wiederverwendung der bereits in `UML` vorhandenen Assoziation führen.

---

[1] Eigentlich ist `CollectionType` Unterklasse von `DataType` und somit nur indirekt Unterklasse von `Classifier.` Dies ändert aber nichts am erläuterten Sachverhalt.

[2] `MOF` wurde hier aus Gründen der Übersichtlichkeit ausgelassen, gestaltet sich jedoch analog zu `UML`.

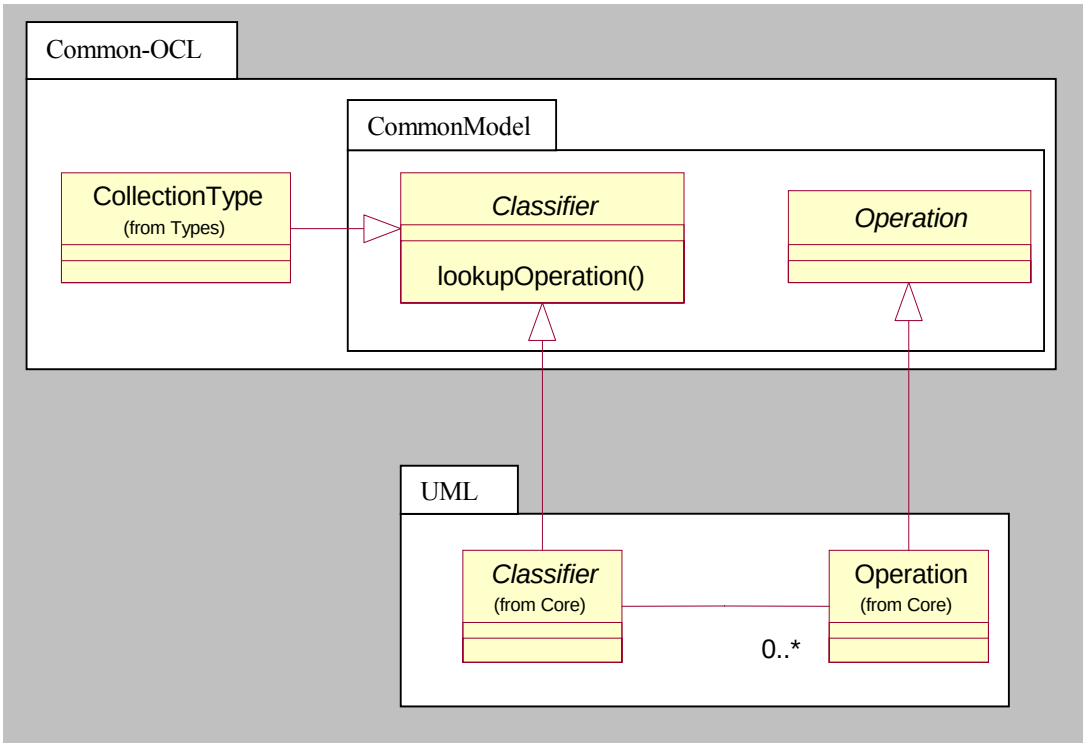


Abbildung 5-9: CollectionType und Classifier

Unter der Prämisse, dass `CollectionType` im Falle von UML die Eigenschaften von `UML::Core:Classifier` und im Falle von MOF die Eigenschaften von `MOF::Model::Class` erben soll, ergibt sich zwangsläufig, dass auch für `CollectionType` selbst Spezialisierungen für MOF und UML existieren müssen. Dies führt zu der in Abbildung 5-10 dargestellten Struktur.

`CollectionType` in `Common-OCL` wird nun nie direkt instantiiert, sondern dient nur als Beschreibung einer gemeinsamen Schnittstelle. Instanzen von `CollectionType` in UML-OCL und MOF-OCL haben die geforderte Eigenschaft, Operationen besitzen zu können.

Die hier an `CollectionType` exemplarisch erläuterte Vererbungshierarchie gilt nicht nur für Klassen im Paket `Types`, sondern auch für die des Paketes `Expressions`, da diese ebenfalls von einer Klasse des UML-Metamodells erben, und zwar von `ModelElement`.

Die Integration von UML- Metamodell bzw. MOF mit dem OCL-Metamodell erfolgt weiterhin, wie in Kapitel 5.1 beschrieben, durch die Verwendung von Metaproxies. Abbildung 5-10 stellt also das Ergebnis der Metamodellintegration dar, bei der die Metaproxies `PClassifier` und `POperation` durch die entsprechenden Klassen des

UML-Metamodells ersetzt wurden. Das Paket, das die Metaproxies enthält, ist für UML nun `UML-OCL::Proxies` und für MOF `MOF-OCL::Proxies`.

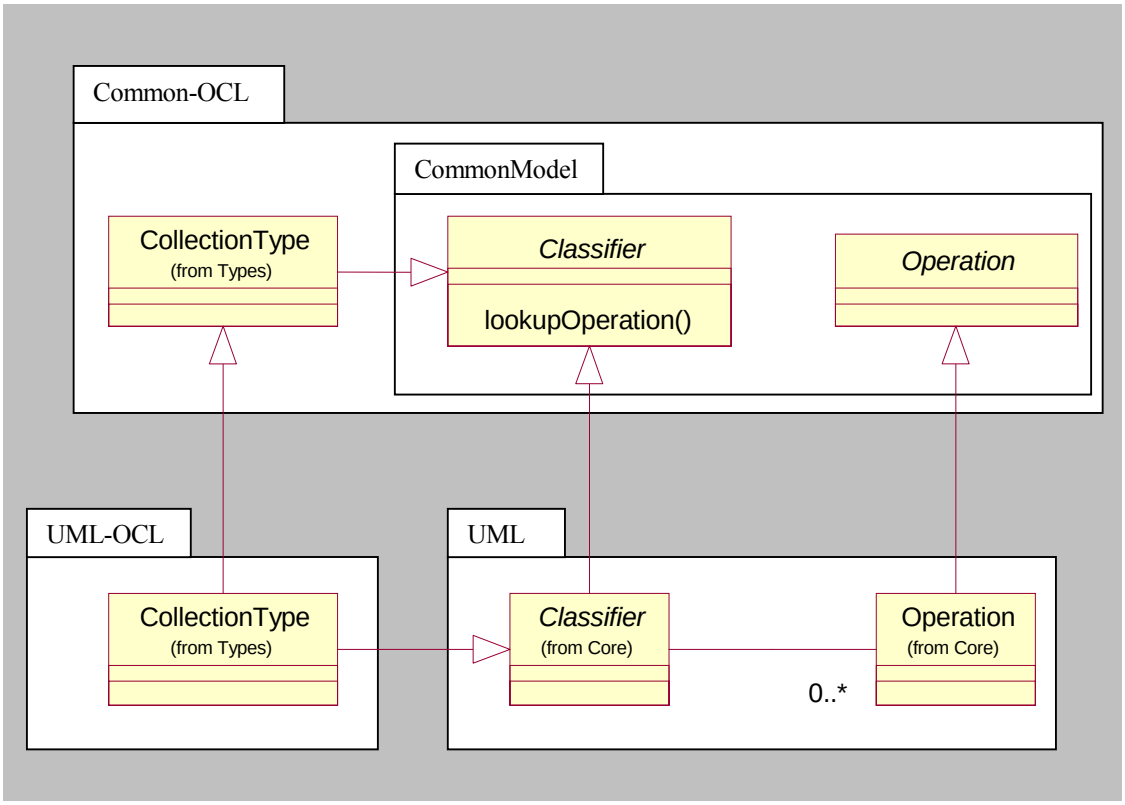


Abbildung 5-10: Spezialisierung der Common-OCL-Klassen

Abstraktion der Assoziationen, Attribute und Operationen des UML-Metamodells

Dieser Abschnitt erläutert, wie mit Assoziationen, Attributen und Operationen des UML-Metamodells umgegangen wird, die in der Spezifikation des OCL-Metamodells, insbesondere jedoch bei der Spezifikation der Abbildung von konkreter nach abstrakter Syntax (Abstract Syntax Mapping) [OCL16,S.4-1] Verwendung finden. Es ist nötig, dass auf diese Assoziationen, Attribute und Operationen auf `Common-OCL`-Ebene zugegriffen werden kann, damit die Implementation des Parsers möglich ist.

Auf Ebene von `Common-OCL` ist es nicht möglich, entlang interner Assoziationen des UML-Metamodells zu navigieren. Da solche Navigationen jedoch in den Wellformedness Rules des OCL-Metamodells in großer Zahl vorkommen, können diese nicht in der Spezifikation angegebenen Form auf `Common-OCL`-Ebene angewandt werden (siehe Kapitel 5.3).

Die Regeln des Abstract Syntax Mapping hingegen benutzen – bis auf wenige Ausnahmen – keine internen Assoziationen des UML-Metamodells. Stattdessen machen sie

Gebrauch von zusätzlich definierten Operationen auf Klassen des UML-Metamodells [OCL16,S.3-22ff]. So wird z.B. für `Classifier` die Operation `lookupOperation()` definiert, die es ermöglicht, eine Operation, die zu einer bestimmten Instanz von `Classifier` gehört, anhand ihres Namens und ihrer Signatur aufzufinden. Wie Abbildung 5-10 zeigt, wird `lookupOperation()`in `Common-OCL` definiert und steht somit für den Parser zur Verfügung.

Die Implementierung von `lookupOperation()`erfolgt in den entsprechenden Unterklassen von `Classifier`, also `Class` in MOF und `Classifier` in UML.[1]

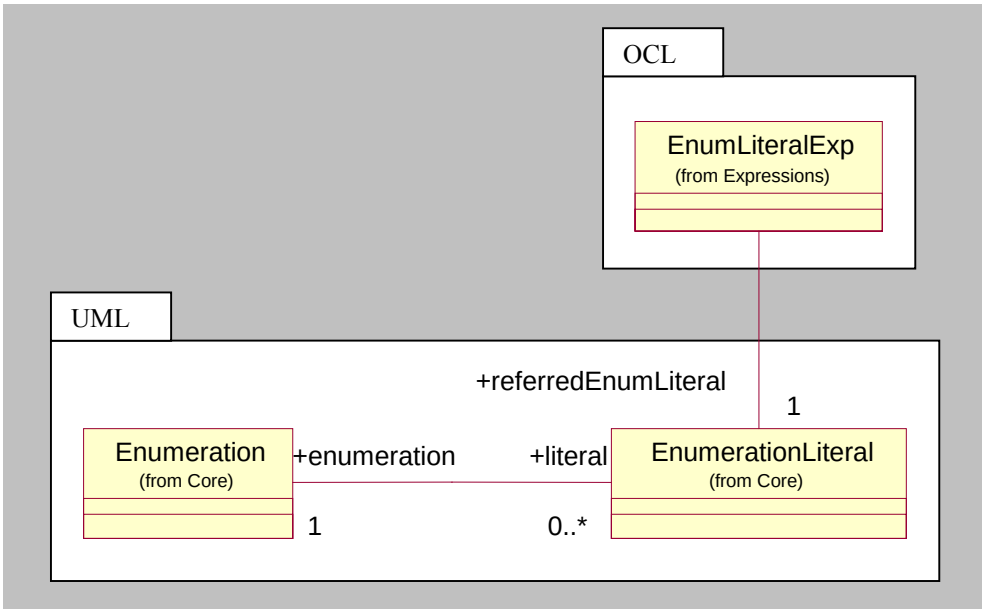


Abbildung 5-11: Navigation entlang Assoziationen des UML-Metamodells

Bei den Regeln des Abstract Syntax Mapping für `EnumLiteralExp` wird entlang einer internen Assoziation des UML-Metamodells navigiert. Dies wird in Abbildung 5-11 anhand des entsprechenden Metamodellausschnittes aus der OCL-Spezifikation und anhand folgender, aus der Spezifikation übernommener Regel für `EnumLiteralCS` [OCL16,S.4-6] deutlich:

[1] Die Operation `lookupOperation()` wurde im Diagramm nicht als abstrakt dargestellt, da MOF 1.4 nicht die Möglichkeit bietet, Operationen als abstrakt zu definieren. Auch macht MOF 1.4 keinerlei Aussagen über das Überschreiben von Operationen in Unterklassen. Ob sich also für `lookupOperation()` in den Unterklassen unterschiedliche Implementierungen angeben lassen, hängt von den Fähigkeiten der MOF-Repository-Implementation ab. Das gewählte Produkt Netbeans Metadata Repository unterstützt dies.

EnumLiteralExpCS

Synthesized attributes

```
EnumLiteralExpCS.ast.referredEnumLiteral =
   EnumLiteralExpCS.ast.type.oclAsType (Enumeration).literal->
      select (l |l.name=simpleNameCS.ast )->any(true)
```

Um diese Regel nun trotzdem auf `Common-OCL` Ebene anwenden zu können, wird für `Enumeration` die zusätzliche Operation `getLiteralA()`[1] definiert, deren Implementierung wiederum erst in den entsprechenden Unterklassen erfolgt. Abbildung 5-12 zeigt `Enumeration` und `EnumerationLiteral` in `Common-OCL` und die Beziehungen zu den entsprechenden Klassen des UML-Metamodells. Die entsprechend angepasste Regel lautet:

EnumLiteralExpCS

…

Synthesized attributes

```
EnumLiteralExpCS.ast.referredEnumLiteral =
   EnumLiteralExpCS.ast.type.oclAsType (Enumeration).getLiteralA()->
      select (l |l.name=simpleNameCS.ast )->any(true)
```

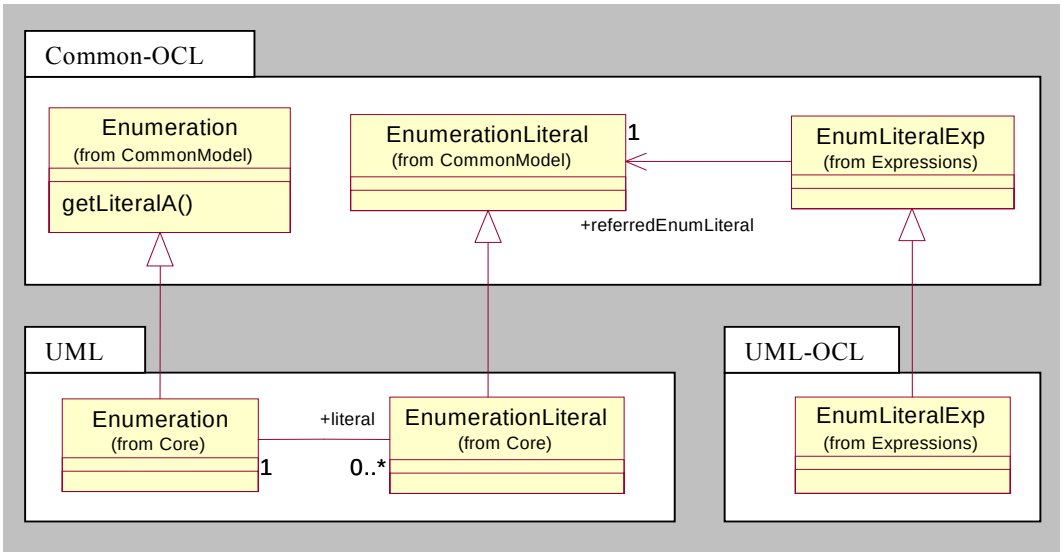


Abbildung 5-12: Operationen in Common-OCL für Assoziationen in UML

[1] In MOF-Modellen ist es möglich, für Parameter und Rückgabewerte nicht nur Typen, sondern auch Multiplizitäten anzugeben. Folglich ist der Rückgabetyp von `getLiteralA()` `EnumerationLiteral` und die Multiplitzität ist 0..*. Der Suffix „A“ im Namen der Operation steht für „abstrakt“ und wurde gewählt, um nicht mit der für die Klasse `Enumeration` des UML-Metamodells (JMI-)generierten Methode `getLiteral()`in Konflikt zu geraten. Die Implementierung von `getLiteralA()` für das UML-Metamodell besteht einfach aus einer Delegation an `getLiteral()`.

Weiterhin zeigt Abbildung 5-12 am Beispiel der Assoziation zwischen `EnumLiteralExp` und `EnumerationLiteral`, dass die in der OCL-Spezifikation angegebenen Assoziationen zwischen OCL- und UML-Metamodell auf `Common-OCL`-Ebene modelliert werden und somit sowohl von `MOF-OCL`, als auch von `UML-OCL` geerbt werden. Auf die gleiche Weise wird mit internen Assoziationen des OCL-Metamodells verfahren.

Neben der Navigation entlang der Assoziationen des UML-Metamodells greifen einige Regeln des Abstract Syntax Mapping auch auf Attribute von UML-Metaklassen zu. Konkret betrifft dies das Attribut `name` der Klasse `ModelElement`. So wird z.B. in obiger Regel mit `l.name=simpleNameCS.ast` der Name eines Literales zum Vergleich herangezogen. `EnumerationLiteral` erbt hierbei das Attribut `name` von `ModelElement`. Ähnlich wie bei internen Assoziationen des UML-Metamodells wird der Zugriff auf Attribute durch Definition zusätzlicher Operationen auf `Common-OCL`-Ebene ermöglicht. Im Falle von `name` sind dies die Operationen `getNameA()` und `setNameA()`[1] für `ModelElement` (Abbildung 5-13).

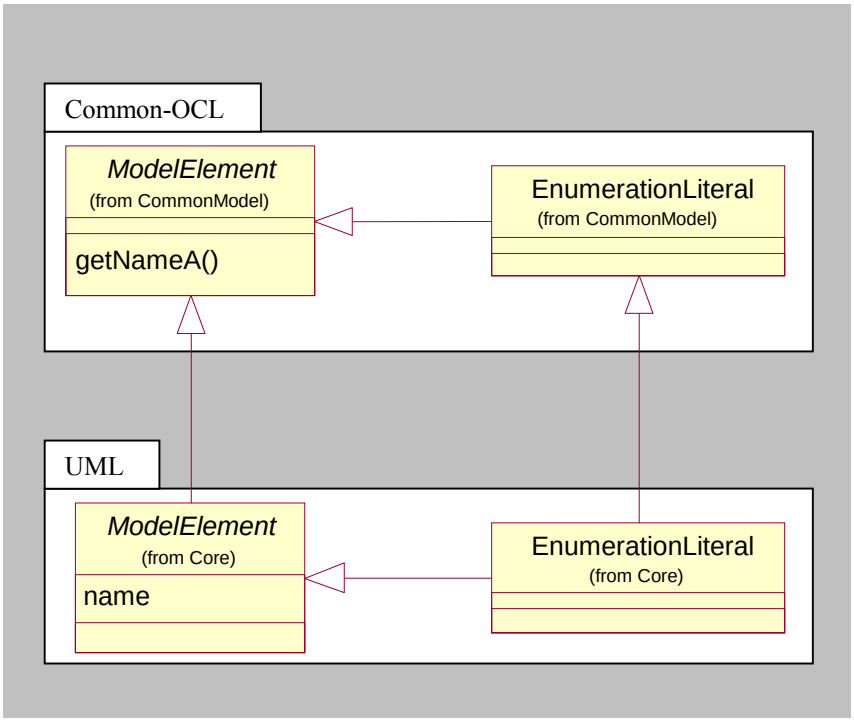


Abbildung 5-13: Operationen in Common-OCL für Attribute in UML

[1] Es ist zwar nicht erforderlich, den Namen von Modellelementen des zugrundeliegenden UML- oder MOF-Modells zu ändern, aber es besteht z.B. von Seiten des Parsers der Bedarf, Instanzen von `OclExpression` (einer Unterklasse von ModelElement) Namen zuzuweisen.

Attribute von OCL-Metaklassen werden hingegen auf `Common-OCL`-Ebene definiert und somit – wie die internen Assoziationen des OCL-Metamodells – an `MOF-OCL` und `UML-OCL` vererbt.

Klassen und Vererbungsbeziehungen im Paket `CommonModel`

Abbildung 5-13 wirft weiterhin die Frage auf, wie mit den Verberbungsbeziehungen zwischen Klassen des UML-Metamodells umzugehen ist. Da `getNameA()` auch für `EnumerationLiteral` verfügbar sein soll, muss die Vererbungsbeziehung zwischen `ModelElement` und `EnumerationLiteral` auch im Paket `CommonModel` modelliert werden. Dies legt nahe, einfach die gesamte Vererbungshierarchie des UML-Metamodells in `CommonModel` zu duplizieren. Wie Abbildung 5-14 illustriert, ist dies jedoch nicht nötig. Es werden die Vererbungshierarchien für `Operation` und `Attribute` im UML-Metamodell und in `Common-OCL` gegenübergestellt. Exemplarisch werden einige Attribute von Klassen des UML-Metamodells dargestellt.

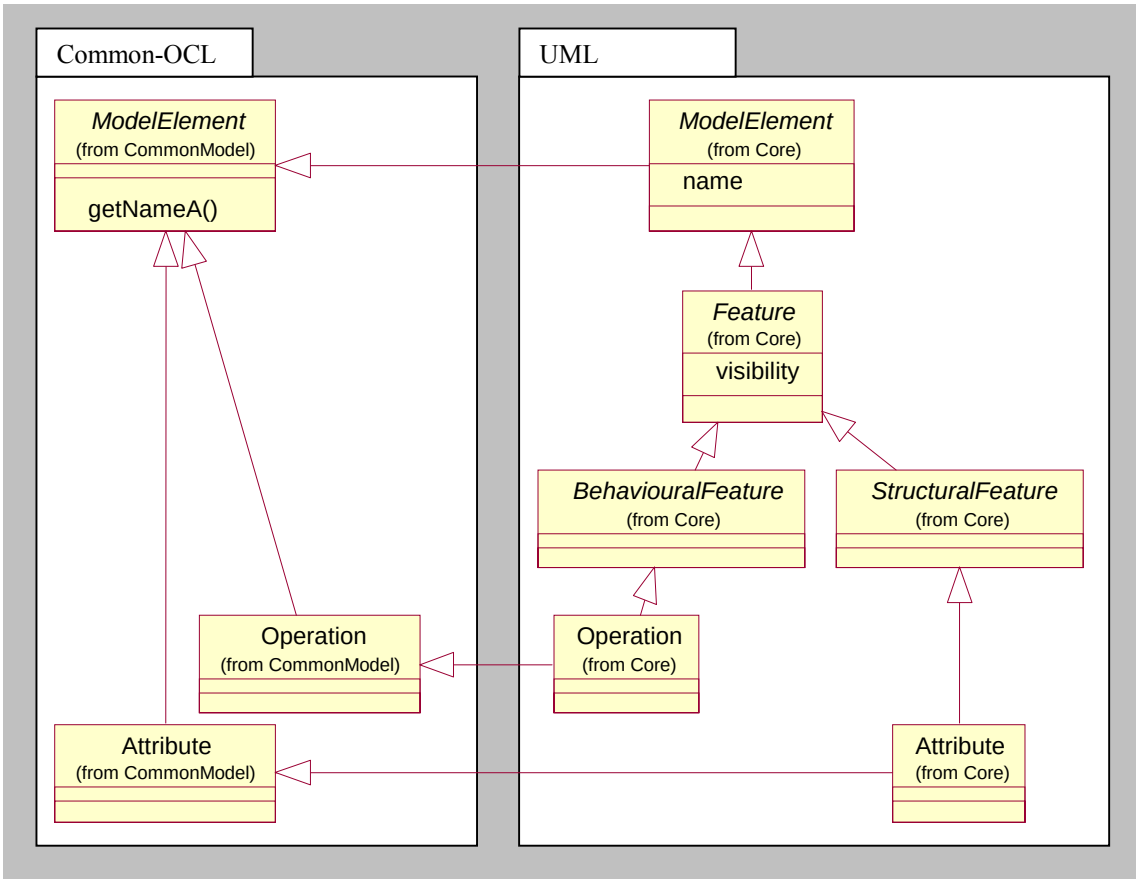


Abbildung 5-14: Vererbungshierarchien in Common-OCL und UML

Die in Common-OCL gewählte Vererbungshierarchie ist flacher, z.B. existieren die Klassen Feature, BehaviouralFeature und StructuralFeature im Paket CommonModel nicht.

Die Entscheidung, ob eine Klasse des UML-Metamodells in das Paket CommonModel übernommen wird, hängt von folgenden Kriterien ab:

- Ist in der OCL-Spezifikation eine Assoziation oder Vererbungsbeziehung zwischen der UML-Metaklasse und einer OCL-Metaklasse angegeben?
- Benutzen Regeln des Abstract Syntax Mapping Attribute oder Operationen der UML-Metaklasse, oder navigieren sie entlang von Assoziationen zu oder von dieser Klasse?

Aus dem ersten Kriterium folgt z.B., dass ModelElement, Attribute und Operation in CommonModel existieren. Aus dem zweiten folgt z.B., dass Enumeration in CommonModel übernommen wird, obwohl keine Assoziationen oder Vererbungsbeziehungen mit OCL-Metaklassen bestehen. Feature hingegen wird in CommonModel nicht benötigt, da z.B. in keiner Regel des Abstract Syntax Mapping das Attribut visibility verwendet wird.

Im Wesentlichen entspricht die Liste der Klassen des UML-Metamodells, für die gleichnamige Klassen in CommonModel existieren, den in Kapitel 8.2 der OCL-Spezifikation angegebenen [OCL16,S.8-2]. Folgende Klassen wurden jedoch nicht berücksichtigt:

- StructuralFeature, da nur eine konkrete Unterklasse existiert, nämlich Attribute.
- CallAction und SendAction. Bezüglich dieser Klassen bestehen in [OCL16] Inkosistenzen. Im Metamodell werden sie von OclMessageExp referenziert, im Abstract Syntax Mapping wird dagegen davon ausgegangen, dass OclMessageExp direkt die Klassen Operation bzw. Signal referenziert. Da letzteres plausibler erscheint, wurden CallAction und SendAction entfernt.

5.2.3 Adapter für MOF und MOF-OCL-Typabbildung

Im Folgenden wird gezeigt, wie bereits in Abschnitt 5.2.1 beschriebene Probleme aufgrund von Unterschieden zwischen UML und MOF mit Hilfe zusätzlicher Adapter im

Paket `MOF-OCL` gelöst werden. Auf Basis eines Adapters für die Metaklasse `DataType` wird eine Abbildung von MOF-Datentypen auf OCL-Datentypen spezifiziert.

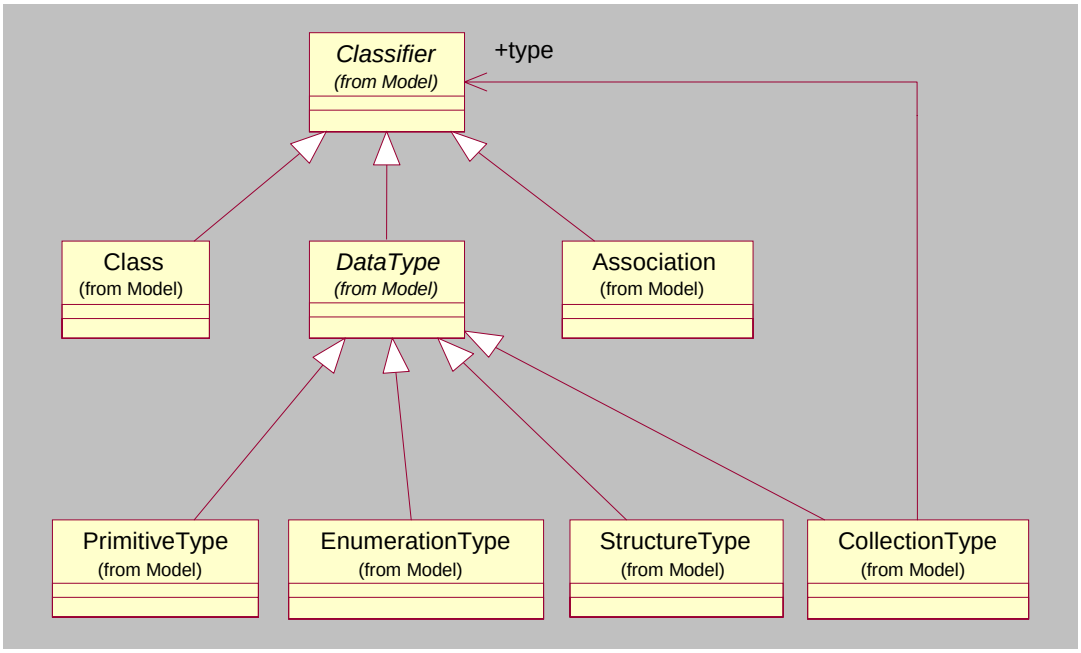


Abbildung 5-15: Metaklassen für Datentypen in MOF

Abbildung 5-15 zeigt einen Ausschnitt aus MOF. Dargestellt sind die Metaklasse `Classifier` und ihre Unterklassen. Für `PrimitiveType` werden in der MOF-Spezifikation Instanzen vordefiniert, und zwar `Boolean`, `Integer`, `Long`, `Float`, `Double` und `String` [MOF14,S.3-114]. Alle weiteren Datentypen in MOF-Modellen (also Instanzen von `CollectionType`, `EnumerationType`, `StructureType`) sind benutzerdefiniert.

Während in UML jede Instanz von `Classifier` (bzw. von Unterklassen von `Classifier`) Attribute, Operationen und Assoziationen mit anderen Instanzen von `Classifier` besitzen darf, gilt dies in MOF nur für Instanzen von `Class` [MOF14,S.3-15]. `DataType` und `Association` können somit nicht als OCL-Typen betrachtet werden, da sie keine Eigenschaften haben, die mittels eines OCL-Ausdrucks erfragt werden könnten (Navigieren entlang von Assoziationen, Abfrage von Attributen oder Aufruf von Operationen). Nicht einmal die Gleichheit zweier Instanzen eines `PrimitiveType` wie `Integer` ließe sich feststellen, da hierfür die entsprechende Operation mit dem Namen „=“ für `Integer` definiert sein müsste. Diese Vergleichsoperation ist in `OclAny` definiert, und alle OCL-Typen erben von `OclAny`. `Integer`

kann jedoch nicht von `OclAny` erben oder kann zumindest dessen Operationen nicht erben, da `Integer` als Instanz von `PrimitiveType` keine Operationen besitzen darf.

Ohne, dass dies nun in der MOF-Spezifikation beschrieben ist, wird aber bei der Formulierung von OCL-Constraints über MOF-Modellen offensichtlich angenommen, dass die im Modell verwendeten MOF-Datentypen als OCL-Datentypen „interpretiert“ werden.

Hierzu ein Beispiel aus den Wellformedness Rules von MOF[1] [MOF14,S.3-95]:

```
context Reference
inv: self.scope = ScopeKind::instance_level
```

Die Metaklasse `Reference` besitzt ein Attribut `scope`. Dieses hat den Typ `ScopeKind`. `ScopeKind` wiederum ist ein Enumerationstyp, oder genauer: eine Instanz von `MOF::Model::EnumerationType`. Die Teilausdrücke `self.scope` und `ScopeKind::instance_level` können aber nicht als Typ einen MOF-Enumerationstyp haben, da für einen solchen keine Operationen definiert sein können, insbesondere auch nicht „=“. Dieses Problem wird folgendermaßen gelöst:

- Modellierung einer Typabbildung auf Metamodellebene durch Verwendung eines Adapters für `DataType`. Für obiges Beispiel heißt das, dass für die MOF-Enumeration `ScopeKind` eine entsprechende OCL-Enumeration existiert, die Instanz des Adapters für `DataType` (bzw. einer Unterklasse) ist und die Operationen besitzen darf.

- Anpassung der WFRs des OCL-Metamodells unter Berücksichtigung dieser Typabbildung (siehe Kapitel 5.3). Für obiges Beispiel betrifft dies z.B. die Wellformedness Rule für `AttributeCallExp`. Diese muss nun zum Ausdruck bringen, dass der Typ des Ausdrucks `self.scope` nicht mehr die MOF-Enumeration `ScopeKind` ist, sondern die entsprechende OCL-Enumeration.

- Anpassung der Spezifikation zusätzlicher Operationen für UML-Metaklassen [OCL16,S.3-22ff]. So muss z.B. die Operation `hasMatchingSignature()` der Metaklasse Operation für `MOF-OCL` derart implementiert werden, dass sie die Typen der Parameter unter Berücksichtigung der Typabbildung miteinander vergleicht. Ein weiteres Beispiel ist die Operation `findClassifier()` der Meta-

[1] Da MOF selbst eine Instanz von MOF ist, sind die WFRs folglich OCL-Constraints über einem MOF-Modell.

klasse `Package`. Findet diese beispielsweise eine MOF-Enumeration im Modell, so muss die entsprechende OCL-Enumeration zurückgegeben werden, welche im Falle ihres Nichtvorhandenseins zunächst erzeugt werden muss (s.a. Anlage C – Spezifikation der Operationen).

Der Adapter für `DataType` ist in dem Paket `Adapters` enthalten, welches zu `MOF-OCL` hinzugefügt wurde. Es handelt sich hierbei um eine Mischform von **Klassen**- und **Objektadapter**. Einerseits erbt der Adapter von `MOF::Model::Class` die Fähigkeit, Operationen besitzen zu können, andererseits ist der abzubildende MOF-Datentyp per Assoziation mit dem Adapter verbunden (Abbildung 5-16).

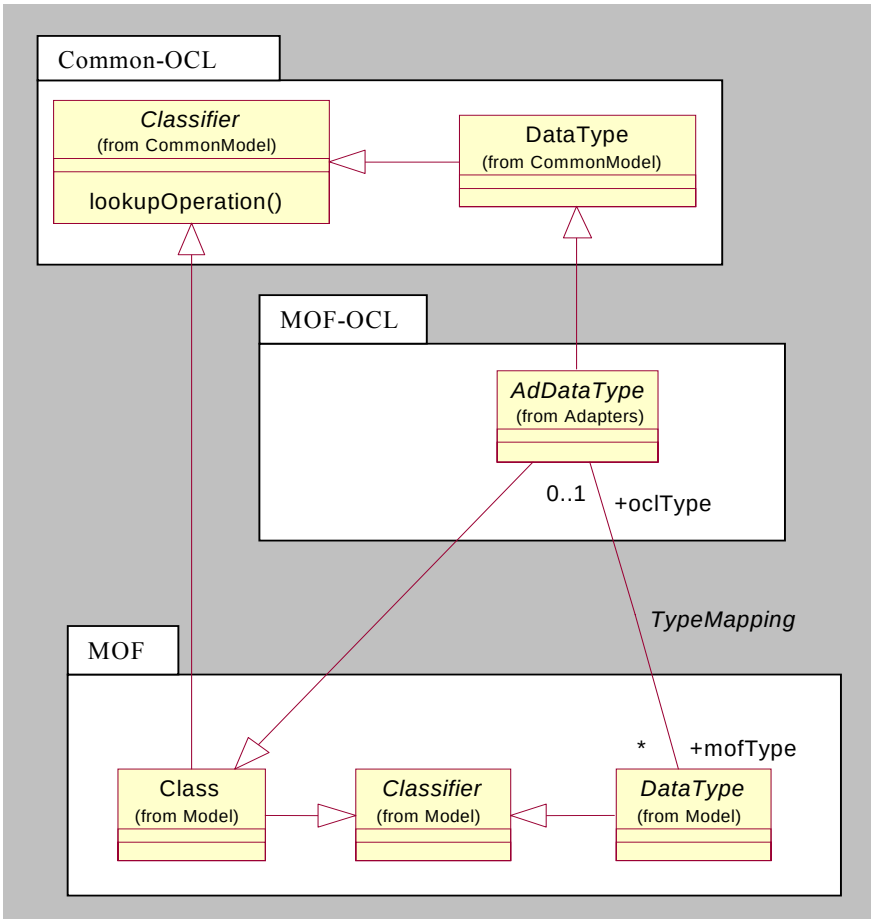


Abbildung 5-16: Adapter für DataType in MOF

Zuordnung zu den Rollen des Entwurfsmusters **Adapter** [Gam95]:

- **Target**: `DataType` in `Common-OCL`
- **Adapter**: `AdDataType` in `MOF-OCL`
- **Adaptee**: `DataType` und `Class` in `MOF`

Im Folgenden wird nun für jede Unterklasse von `DataType` mittels OCL-Constraints beschrieben, wie der jeweilige MOF-Datentyp auf einen OCL-Datentyp abgebildet wird. Dabei wird von folgender zusätzlich definierter Operation Gebrauch gemacht:

```
context MOF::Model::Classifier
def: toOclType() : MOF::Model::Class
     = if self.oclIsKindOf(MOF::Model::Class) then self
       else if self.oclIsKindOf(MOF::Model::DataType) then
          self.oclAsType(MOF::Model::DataType).oclType
          else OclUndefined endif
       endif
```

Die Operation liefert für einen `Classifier` den entsprechenden OCL-Typ. Im Falle einer Instanz von `Class` wird diese selbst zurückgegeben, im Falle von `DataType` die entsprechende Instanz der Adapterklasse für `DataType`. Für Instanzen der Klasse `Association`, die ebenfalls eine Unterklasse von Classifier ist, liefert die Operation `OclUndefined`. Da Assoziationen in MOF keine Attribute oder Operationen besitzen dürfen, macht es keinen Sinn, Instanzen von `Association` z.B. auf Instanzen von `AssociationClass` in Common-OCL abzubilden.

In [Ric01,S.82ff] wird vorgeschlagen, Assoziationen generell – also auch ohne Assoziationsklasse – als OCL-Typen zu behandeln, jedoch wurde dies einerseits in der OCL-Spezifikation nicht umgesetzt, und andererseits ist einziger Sinn dieses Vorschlages, dass die Navigationsmöglichkeiten im Falle von n-ären (n>2) Assoziationen verbessert werden. In MOF sind jedoch nur binäre Assoziationen erlaubt [MOF14,S.3-50].

Tabelle 5-1 zeigt eine Übersicht über die im Folgenden spezifizierte Typabbildung[1].

MOF-Datentyp	OCL-Datentyp
PrimitiveType	Primitive
EnumerationType	Enumeration
CollectionType	CollectionType
StructureType	TupleType

Tabelle 5-1: Abbildung von MOF-Datentypen auf OCL-Datentypen

[1] Dargestellt ist die Korrespondenz zwischen den Metaklassen, deren Instanzen die eigentlichen Datentypen sind.

Da `Primitive` und `Enumeration` wiederverwendete Klassen des UML-Metamodells sind, werden für diese zwei Klassen Adapter in `MOF-OCL` benötigt. Für die Kollektionstypen und für `TupleType` ist dies nicht nötig, da sich Spezialisierungen bereits im Paket `Types` in `MOF-OCL` befinden.

PrimitiveType

Wie Abbildung 5-17 zeigt, wird für die Abbildung der primitiven Typen der Adapter `MOF-OCL::Adapters::AdPrimitive` verwendet. Instanzen dieser Klasse sind die vier primitiven Typen der OCL Standard Library, also `Boolean`, `Integer`, `Real` und `String` (s.a. Kapitel 5.4). Diese vier Typen stehen den sechs primitiven MOF-Typen `Boolean`, `Integer, Long`, `Float`, `Double` und `String` gegenüber. Die folgenden OCL-Invarianten beschreiben demzufolge keine injektive Abbildung.

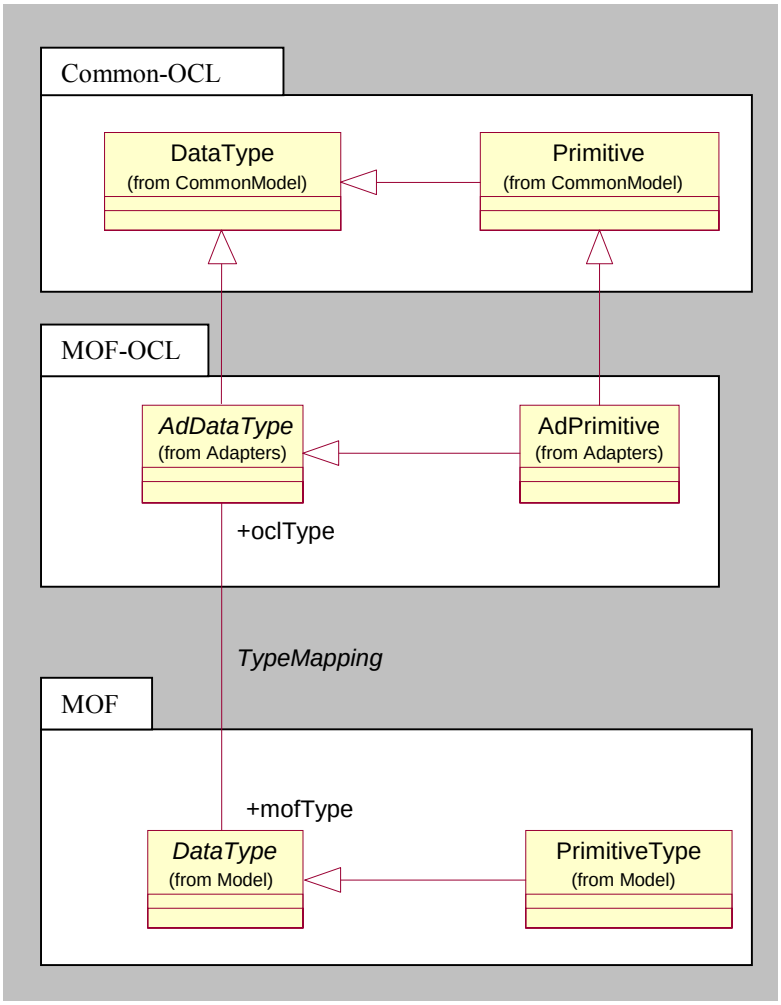


Abbildung 5-17: PrimitiveType in MOF und Primitive in OCL

```
context MOF::Model::PrimitiveType
inv: self.oclType.oclIsTypeOf(MOF-OCL::Adapters::AdPrimitive)
inv: self.name = 'Boolean' implies oclType.name = 'Boolean'
inv: self.name = 'Integer' implies oclType.name = 'Integer'
inv: self.name = 'Long' implies oclType.name = 'Integer'
inv: self.name = 'Float' implies oclType.name = 'Real'
inv: self.name = 'Double' implies oclType.name = 'Real'
inv: self.name = 'String' implies oclType.name = 'String'
```

EnumerationType

Während im UML-Metamodell – und somit auch in `Common-OCL` – die Literale einer `Enumeration` als Instanzen einer separaten Metaklasse `EnumerationLiteral` modelliert werden, geschieht dies in MOF durch das Attribut `labels` in der Metaklasse `EnumerationType`.

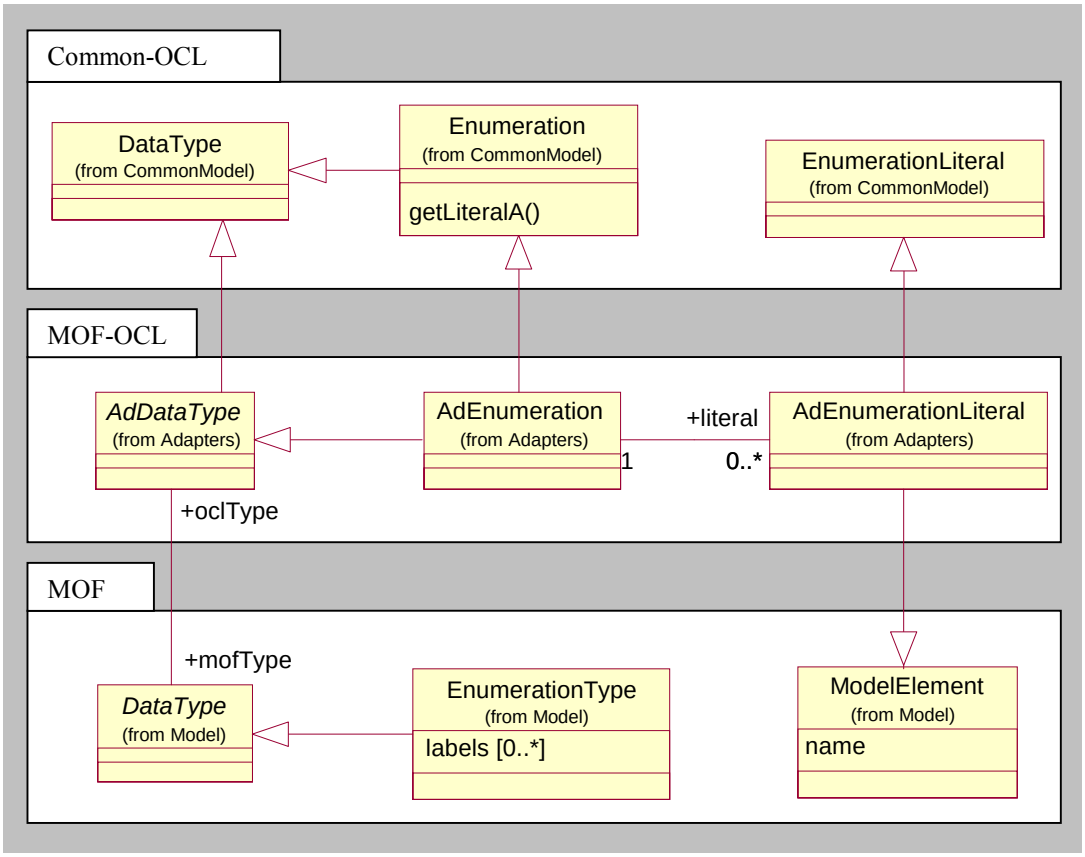


Abbildung 5-18: EnumerationType in MOF und Enumeration in OCL

Um nun `EnumerationType` an die geforderten Schnittstellen `Enumeration` und `EnumerationLiteral` anzupassen, werden in `MOF-OCL` zwei Adapterklassen definiert, die die Namen `AdEnumeration` bzw. `AdEnumerationLiteral` tragen

(Abbildung 5-18). Zu einer bestimmten Instanz von `AdEnumeration` gehört dabei eine Menge von Instanzen von `AdEnumerationLiteral`, wobei diese jeweils mit den Werten des Attributes `labels` der betreffenden Instanz von `EnumerationType` korrespondieren:

```
context MOF::Model::EnumerationType
inv: oclType.oclIsTypeOf(MOF::Adapters::AdEnumeration) and
   oclType.name = name and
   oclType.oclAsType(MOF::Adapters::AdEnumeration).literal.name
       = labels->asBag()
```

CollectionType

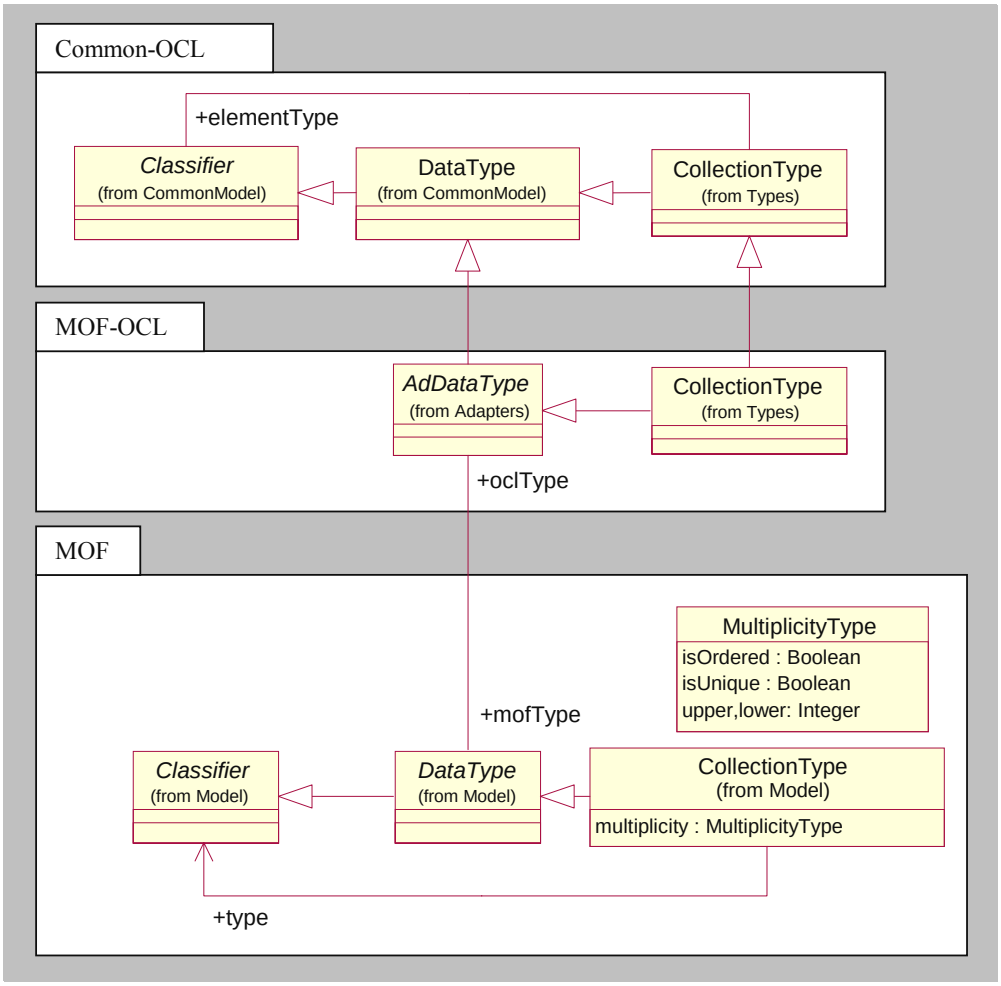


Abbildung 5-19: CollectionType in MOF und OCL[1]

[1] Nicht dargestellt sind die Unterklassen von `CollectionType`. Diese sind sowohl in `Common-OCL` als auch in MOF-OCL vorhanden und unterliegen dem gleichen Vererbungsschema wie `CollectionType`. So erbt z.B. `SetType` in `MOF-OCL` sowohl von `CollectionType` in `MOF-OCL`, als auch von `SetType` in `Common-OCL`.

In Common-OCL hat die Metaklasse CollectionType die vier Unterklassen SetType, BagType, SequenceType und OrderedSet. Je nachdem, welche dieser Unterklassen instantiiert wird, erhält man Kollektionstypen für Mengen, Multimengen, Listen oder geordnete Mengen. In MOF existiert ebenfalls eine Metaklasse CollectionType [MOF14,S.3-34]. Die Art der Kollektion wird jedoch durch das Attribut multiplicity näher beschrieben. Dieses hat den Typ MultiplicityType, welcher die Attribute lower, upper, isOrdered, isUnique definiert. Während die ersten beiden die minimale und maximale Anzahl von Elementen in der Kollektion festlegen und für die Abbildung auf OCL-Kollektionstypen nicht relevant sind[1], werden isOrdered und isUnique dazu verwendet, die entsprechende Unterklasse von CollectionType in Common-OCL zu wählen:

isOrdered	isUnique	OCL-Kollektionstyp
false	false	BagType
false	true	SetType
true	false	SequenceType
true	true	OrderedSetType

Tabelle 5-2: Abbildung von MOF- auf OCL-Kollektionstypen

Der einem MOF-Kollektionstyp zugeordnete Elementtyp ist eine Instanz von MOF::Model::Classifier. Folglich muss bei der Abbildung auf OCL-Kollektionstypen auch dieser Elementtyp abgebildet werden. Dies setzt sich bei verschachtelten Kollektionstypen entsprechend rekursiv fort. Für die Spezifikation dieser Abbildung wird von der weiter oben definierten Operation toOclType() Gebrauch gemacht.

Im Gegensatz zu den Abbildungsregeln für EnumerationType wird der Name eines Kollektionstypes nicht aus MOF übernommen. Vielmehr wird dieser Name durch die entsprechenden Wellformedness Rules des OCL-Metamodells bestimmt [OCL16,S.3-6f]. So lautet dieser beispielsweise für eine Instanz von SetType, welche als Elementtyp den primitiven Typ Integer hat, „Set(Integer)“.

[1] Das heißt, Kollektionen in OCL haben immer Multiplizität „0..*“

```
context MOF::Model::CollectionType
inv: oclType.oclIsKindOf(MOF-OCL::Types::CollectionType) and
     if multiplicity.isOrdered then
        if multiplicity.isUnique then
           oclType.oclIsTypeOf(MOF-OCL::Types::OrderedSetType)
        else oclType.oclIsTypeOf(MOF-OCL::Types::SequenceType)
        endif
     else
        if multiplicity.isUnique then
           oclType.oclIsTypeOf(MOF-OCL::Types::SetType)
        else oclType.oclIsTypeOf(MOF-OCL::Types::BagType)
        endif
     endif and
     oclType.oclAsType(MOF-OCL::Types::CollectionType).elementType
        = type.toOclType()
```

StructureType

Sowohl bei `StructureType` in `MOF`, als auch bei `TupleType` in `Common-OCL` handelt es sich um eine Metaklasse für strukturierte Datentypen. Während bei `TupleType` die Felder des Datentyps als Instanzen von Attribute modelliert werden[1], existiert hierfür in MOF die Metaklasse `StructureField`. Demzufolge ist jede Instanz von `StructureField` auf eine Instanz von `Attribute` mit gleichem Namen und gleichem Typ abzubilden.

Der Name eines `TupleType` ist nicht frei wählbar, sondern wird entsprechend [OCL16,S.3-7] aus den Namen und Typnamen seiner Felder gebildet. So bedeutet der Name „Tuple(a:Integer,b:Real)“, dass der `TupleType` zwei Attribute mit den Namen „a“ und „b“ des Typs Integer bzw. Real besitzt.

Folgende OCL-Invariante beschreibt die Abbildung von `StructureType` auf `TupleType` mittels Abbildung von `StructureField` auf `Attribute`. Für die Abbildung der Typen der einzelnen Felder wird dabei wiederum `toOclType()` verwendet.

```
package MOF::Model
context StructureType
inv: oclType.oclIsTypeOf(MOF-OCL::Types::TupleType) and
   oclType.containedElement -> select(me | me.oclIsTypeOf(Attribute))
```

[1] Dies ermöglicht – wie bei Klassen – den Zugriff auf die Felder mittels `AttributeCallExp`.

```
      ->forAll(me | me.oclAsType(Attribute).type.oclIsKindOf(Class)) and
  containedElement -> select(me | me.oclIsTypeOf(StructureField))
      -> collect(me | Tuple {
            name : String = me.name,
            type : Class = me.oclAsType(StructureField).type.toOclType()
         })
= oclType.containedElement -> select(me | me.oclIsTypeOf(Attribute))
      -> collect(me | Tuple {
            name : String = me.name,
            type : Class = me.oclAsType(Attribute).type.oclAsType(Class)
         })
```

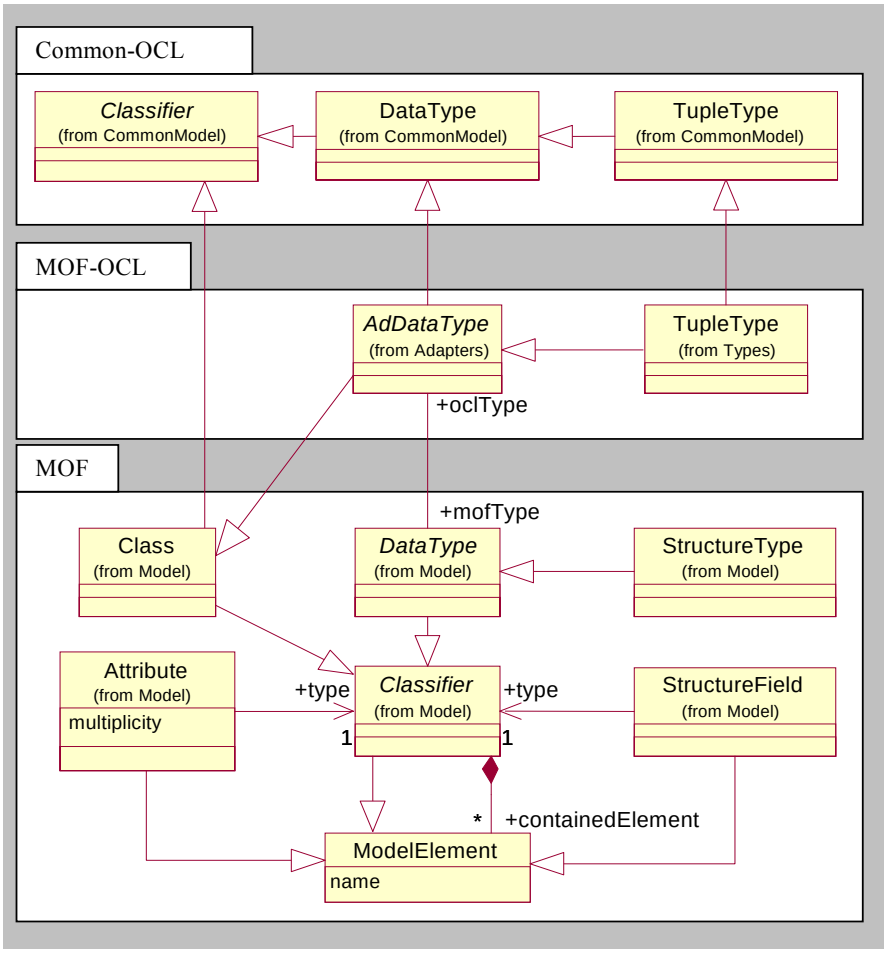


Abbildung 5-20: StructureType in MOF und TupleType in OCL[1]

[1] Aus Gründen der Übersichtlichkeit wurde die Vererbungshierarchie in `MOF` vereinfacht dargestellt. So fehlen die Metaklassen `TypedElement` und `Namespace`. Erstere ist gemeinsame Oberklasse von `Attribute` und `StructureField`. Letztere befindet sich in der Vererbungshierarchie zwischen `Classifier` und `ModelElement`.

5.2.4 Typabbildung für UML

Die im vorigen Abschnitt erläuterten Gründe für die Notwendigkeit einer Typabbildung treffen nicht nur für Metamodelle zu, sondern auch für UML-Modelle. Auch in diesen liegen zunächst nicht die primitiven Datentypen vor, die in der OCL Standard Library definiert sind. Vielmehr ist es sogar der Fall, dass jedes UML-CASE-Tool seine eigenen primitiven Datentypen definieren kann. Die Ursache hierfür ist, dass das UML-Metamodell im Gegensatz zu MOF keine Typen vordefiniert[1].

Die im vorigen Abschnitt für `MOF::Model::Classifier` definierte Operation `toOclType()` wird also auch für `UML::Core::Classifier` benötigt. In der Implementation bildet diese Methode Datentypen basierend auf ihrem Namen ab. So ist z.B. das Ergebnis der Typabbildung für eine Instanz von `UML::Core::DataType`, die den Namen „Integer“ oder „int“ trägt, der Datentyp `Integer` der OCL Standard Library.

5.2.5 Generierung der spezifischen OCL-Pakete

In Abbildung 5-24 ist erneut die Paketstruktur dargestellt. Das Metamodell scheint gegenüber dem in der Spezifikation angegebenen auf die dreifache Größe angewachsen zu sein. Dies bedeutet einen erhöhten Wartungsaufwand. Werden z.B. strukturelle Änderungen in den Paketen `Expressions` oder `Types` in `Common-OCL` vorgenommen, so sind diese auch in den entsprechenden Paketen in `MOF-OCL` und `UML-OCL` vorzunehmen. Die spezifischen `Expressions`- und `Types`-Pakete lassen sich jedoch komplett aus denen in `Common-OCL` generieren.

[1] Hier muss klar unterschieden werden zwischen den Datentypen, die in UML-Modellen verwendet werden und solchen, die zur Definition des UML-Metamodells verwendet werden. Letztere sind im Paket `Data Types` des UML-Metamodells definiert. Sie sind Instanzen der MOF-Klassen `PrimitiveType` und `EnumerationType`, auch wenn diese Tatsache in der UML-Spezifikation recht unpräzise beschrieben wird.

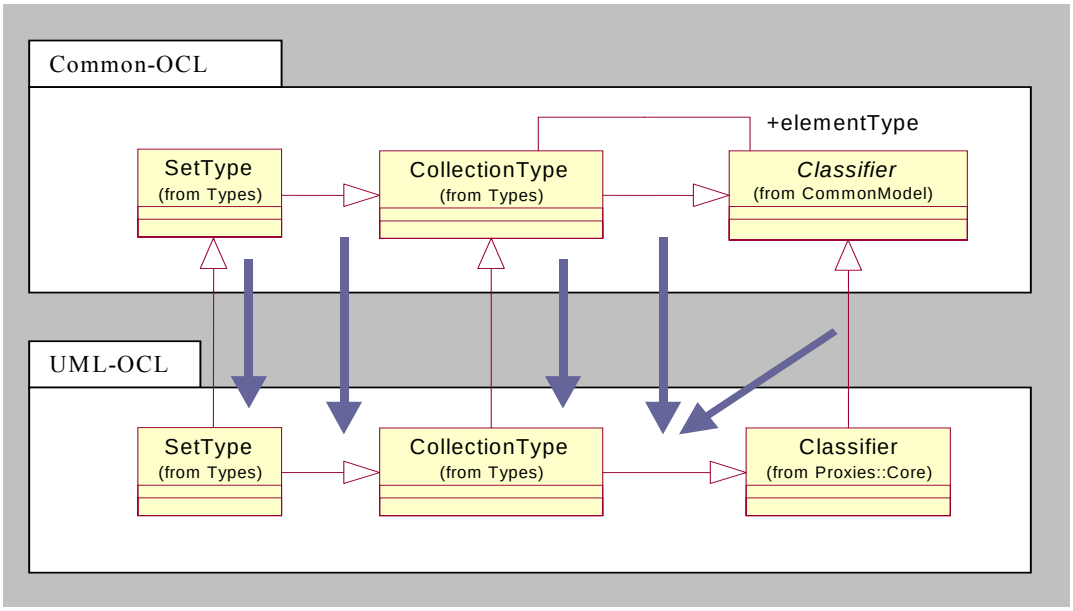


Abbildung 5-21: Generierung der spezifischen OCL-Pakete

Abbildung 5-21 zeigt dies am Beispiel von `CollectionType` und `SetType`. Die einzelnen Teilschritte der Generierung lauten wie folgt:

Teilschritt	Ergebnis am Beispiel
Erzeuge leeres Paket `Expressions` bzw. `Types` (im Folgenden als spezifische OCL-Pakete bezeichnet).	`UML-OCL::Types`
Erzeuge für jede Klasse in `Common-OCL::Expressions` bzw. `Common-OCL::Types` eine Klasse gleichen Namens im entsprechenden spezifischen OCL-Paket[30].	`CollectionType` und `SetType` in `UML-OCL::Types`
Erstelle für jede der erzeugten Klassen eine Vererbungsbeziehung zwischen ihr und ihrem in `Common-OCL` befindlichen Original.	`CollectionType` bzw. `SetType` in `UML-OCL::Types` erben von `CollectionType` bzw. `SetType` in `Common-OCL::Types`

[30] Hierbei ist die erzeugte Klasse genau dann abstrakt, wenn das auch für ihr Original in `Common-OCL` gilt. In dieser Vorgehensweise liegt auch die Ursache dafür, dass die Klassen in Common-OCL nicht alle als abstrakt modelliert wurden, obwohl sie nie instantiiert werden sollen.

Betrachte die Oberklassen der originalen Klassen.		Im Falle von `CollectionType` ist das `Classifier` und im Falle von `SetType` `CollectionType`.
Fall 1:	Oberklasse befindet sich in `CommonModel`. Betrachte die Unterklasse dieser Oberklasse, die sich im jeweiligen Paket für Metaproxies oder Adapter befindet[31].Erstelle Vererbungsbeziehung zwischen generierter Klasse und diesem Adapter bzw. Metaproxy	UML-OCL::Types::CollectionType erbt vom (UML-)Metaproxy für Classifier.
Fall 2:	Oberklasse befindet sich in `Expressions` bzw. `Types`. Erstelle Vererbungsbeziehung zwischen betrachteter generierter Klasse und der für die Oberklasse generierten Klasse.	In `UML-OCL::Types` erbt `SetType` von `CollectionType`.

Tabelle 5-3: Generierung der spezifischen OCL-Pakete

Man sieht, dass für diese Generierung die Vererbungsbeziehungen zwischen den Klassen in `Common-OCL` und denen in den Metaproxy- bzw. Adapterpaketen von entscheidender Bedeutung sind. Diese Vererbungsbeziehungen beschreiben, wie die spezifischen OCL-Pakete für MOF und UML aus `Common-OCL` herzuleiten sind.

Die Generierung lässt sich auch als ein Teilschritt der Metamodellintegration auffassen, womit diese nun folgendermaßen abläuft:

1. Lade das gemeinsame OCL-Metamodell und MOF bzw. das UML-Metamodell in das Repository.

[31] Dies ergibt für `MOF-OCL` bzw. `UML-OCL` jeweils genau eine Klasse, entweder einen Metaproxy oder einen Adapter. Bei Klassen wie AssociationClass, von denen keine OCL-Metaklassen erben, ist es auch zulässig, dass kein Metaproxy und kein Adapter existiert.

2. Generiere die spezifischen Pakete `Types` und `Expressions`.
3. Ersetze die Metaproxies durch die entsprechenden MOF- bzw. UML-Metamodell-Klassen.

5.2.6 Abstrakte Fabriken und Hilfsklassen

Da das Erzeugen von OCL-Ausdrücken als Instanzen der Metaklassen des Paketes `Expressions` auf transparente Art möglich sein soll, also unabhängig davon, ob `UML-OCL` oder `MOF-OCL` zu Grunde liegt, wird dem Paket `Expressions` die Klasse `OCLExpressionFactory` hinzugefügt. Diese enthält für jede konkrete Klasse im Paket Expressions eine Operation der Form

`create<Name_der_Klasse>():<Name_der_Klasse>.`

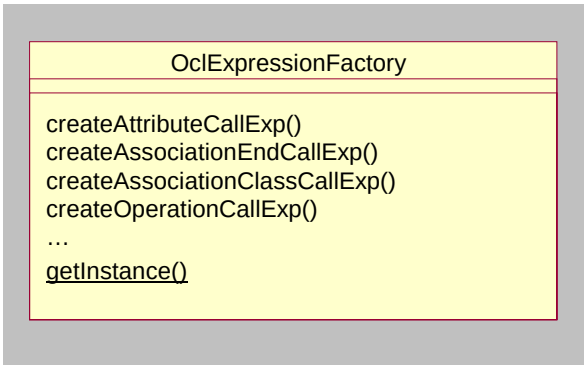


Abbildung 5-22: OclExpressionFactory

Zuordnung zu den Rollen des Entwurfsmusters **Abstract Factory** [Gam95]:

- **Abstract Factory:** `OclExpressionFactory` in `Common-OCL::Expressions`
- **ConcreteFactory:** `OclExpressionFactory` in den für `MOF-OCL` und `UML-OCL` generierten `Expressions`-Paketen[32]
- **AbstractProduct:** `OperationCallExp`, `AttributeCallExp` usw. in `Common-OCL::Expressions`
- **ConcreteProduct:** `OperationCallExp`, `AttributeCallExp` usw. in den für `MOF-OCL` und `UML-OCL` generierten `Expressions`-Paketen

[32] Aus den Metaklassen `MOF-OCL::Expressions::OclExpression` und `UML-OCL::Expressions::OclExpression` werden durch das JMI-Mapping bzw. MOF-IDL-Mapping zunächst nur Schnittstellen erzeugt, die von der Schnittstelle für `Common-OCL::Expressions::OclExpression` erben. Die eigentlichen konkreten Fabriken sind die Implementationsklassen für diese Schnittstellen.

Neben Abstract Factory findet auch das Entwurfsmuster Singleton [Gam95] Anwendung. Daher wird noch eine Methode `getInstance()` definiert, die jeweils die einzige Instanz von `OclExpressionFactory` für `MOF-OCL` bzw. `UML-OCL` liefert[33].

Für das Paket `Types` ist es nicht nötig, dass der Parser jede der enthaltenen Klassen explizit instantiieren kann. Vielmehr sind die Instanzen dieser Klassen bereits als Datentypen im betrachteten UML- oder MOF-Modell enthalten oder werden im Rahmen der Typabbildung erzeugt (s. Kapitel 5.2.3). Einige Typen müssen jedoch explizit vom Parser erzeugt werden, z.B. Instanzen von `TupleType`. Im Paket `Types` wird daher zusätzlich die Klasse `OclLibrary` definiert, die einerseits den Zugriff auf vordefinierte Typen in der OCL Standard Library erleichtern soll, andererseits aber auch Fabrik-Methoden zum expliziten Erzeugen von Typen enthält. Die vollständige Spezifikation von `OclLibrary` befindet sich in Anlage C – Spezifikation der Operationen, Seite 128.

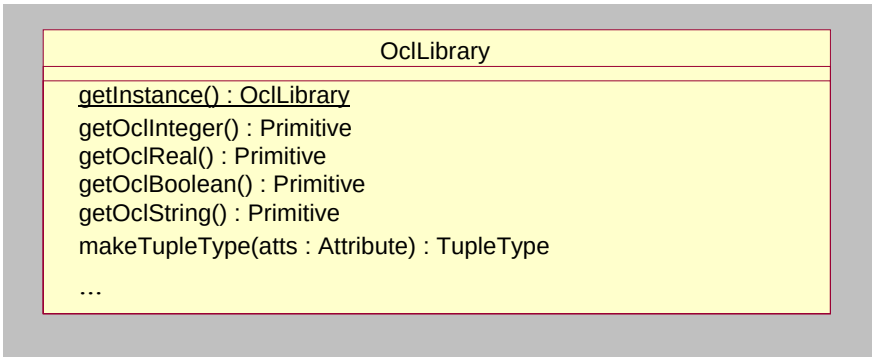


Abbildung 5-23: OclLibrary

[33] Genauer: Die einzige Instanz in Bezug auf die Metamodellinstanz. Wird das OCL-Metamodell im Repository mehrfach instantiiert, so existiert innerhalb jeder dieser Instanzen auch eine Instanz von `OclExpressionFactory`.

5.2.7 Zusammenfassung

In folgender Tabelle und Abbildung werden zusammenfassend die Pakete des gemeinsamen OCL-Metamodells noch einmal genannt und erläutert.

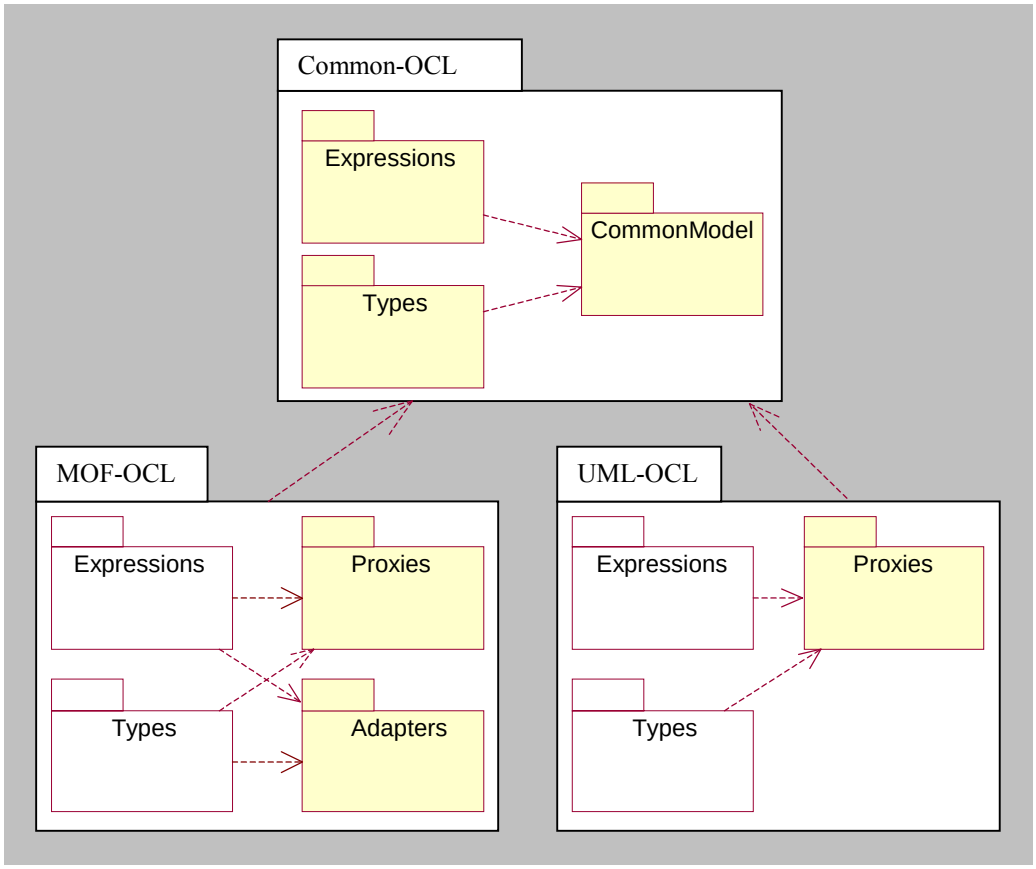


Abbildung 5-24: Paketstruktur, detailliert

Common-OCL	Expressions, Types	• Enthalten die Metaklassen für OCL-Ausdrücke und OCL-Typen wie in der OCL-Spezifikation beschrieben. • Assoziationen und Vererbungsbeziehungen jedoch nicht direkt mit UML-Metaklassen, sondern mit Klassen in `CommonModel`. • Assoziationen und Operationen für OCL-Metaklassen werden nie in den spezifischen `Expressions`- und `Types`-Paketen modelliert, sondern ausschließlich hier. • Zusätzliche Definition von abstrakten Fabriken bzw. Hilfsklassen (`OclExpressionFactory` und `OclLibrary`)

	CommonModel	• Enthält gemeinsame Oberklassen für MOF- und UML-Metamodell entsprechend Kapitel 8 der OCL-Spezifikation. • Keine Attribute oder Assoziationen. Stattdessen Definiton zusätzlicher Operationen für Zugriff auf benötigte MOF- bzw. UML-Metaattribute und Meta-assoziationen.
MOF-OCL, UML-OCL	Expressions, Types	• Werden aus `Common-OCL` generiert. • Enthalten die spezifischen Metaklassen für OCL-Ausdrücke und OCL-Typen für MOF bzw. OCL. • Klassen erben von den entsprechenden Klassen in `Types` und `Expressions` in `Common-OCL`. • Klassen erben nicht von Klassen in `CommonModel`, sondern von den entsprechenden Unterklassen in `Proxies` bzw. `Adapters`.
	Proxies	• Enthält die Metaproxies für die Klassen des UML- bzw. MOF-Metamodells.
	Adapters	• Enthält Adapter für die Klassen in `CommonModel`, für die es keine hinreichend adäquaten Klassen im MOF-Metamodell gibt.

Tabelle 5-4: Pakete des gemeinsamen OCL-Metamodells

5.3 Wellformedness Rules und Typinferenz

Die in [OCL16] angegebenen Wellformedness Rules für das OCL-Metamodell navigieren entlang interner Assoziationen des UML-Metamodells und greifen auf Attribute zu, die in Klassen des UML-Metamodells definiert sind. Somit sind sie nicht auf `MOF-OCL` oder `Common-OCL` anwendbar. Abschnitt 5.3.1 zeigt, wie die WFRs des OCL-Metamodells vom UML-Metamodell entkoppelt und auf `Common-OCL`-Ebene definiert werden. Abschnitt 5.3.2 gibt einen Überblick über an den WFRs vorgenommene Ergänzungen und Vervollständigungen .

In Anlage B – Wellformedness Rules findet sich die Spezifikation aller WFRs für das gemeinsame OCL-Metamodell.

Auf Basis der WFRs für das Paket `Expressions` wird in Abschnitt 5.3.3 die Implementierung der Typinferenz beschrieben, welche für einen OCL-Ausdruck den Typ bestimmt.

5.3.1 Anpassung der WFRs an das gemeinsame OCL-Metamodell

Die Anpassung der WFRs an das gemeinsame OCL-Metamodell lässt sich in folgende Schritte unterteilen:

1. Entkopplung der WFRs vom UML-Metamodell durch Definition zusätzlicher Operationen und Nutzung dieser, anstatt direkt auf Attribute oder Assoziationen des UML-Metamodells zuzugreifen.
2. Spezifikation dieser Operationen für UML und für MOF. (Für eine vollständige Auflistung siehe Anlage C – Spezifikation der Operationen)

Diese Schritte werden im Folgenden anhand der WFR für `AttributeCallExp` erläutert. In ihrer ursprünglichen Form hat diese folgendes Aussehen [OCL16,S.3-18]:

```
context AttributeCallExp
inv: self.type = self.referredAttribute.type
```

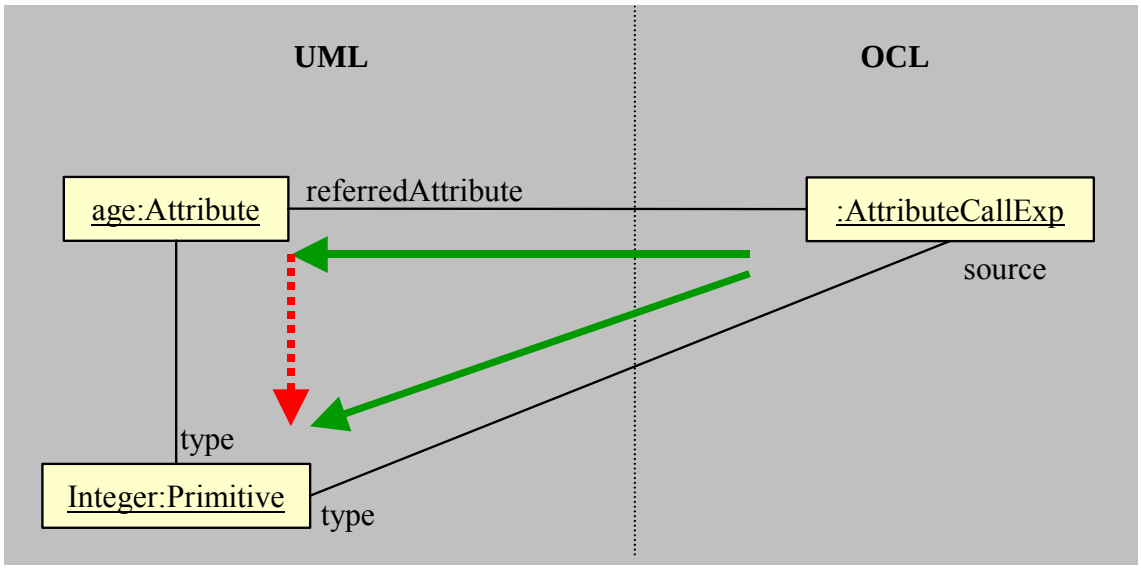


Abbildung 5-25: Wellformedness Rule für AttributeCallExp

Abbildung 5-25 illustriert diese WFR anhand eines Ausschnittes aus dem bereits im Grundlagenkapitel vorgestellten Objektdiagramm.

Wie zu erkennen ist, handelt es sich bei `self.type` und `self.referredAttribute` um Navigationen entlang Assoziationen vom OCL-Metamodell zum UML-Metamodell. Da derartige Assoziationen auf `Common-OCL`-Ebene modelliert sind (siehe Kapitel 5.2.2), können sie in der WFR unverändert bleiben.

Dagegen ist `referredAttribute.type` eine Navigation entlang einer Assoziation innerhalb des UML-Metamodells, und zwar von `UML::Core::Attribute` zu `UML::Core::Classifier`. Eine solche Navigation ist jedoch auf `Common-OCL-Ebene` nicht möglich.

Daher wird für `Attribute` im Paket `CommonModel` die Methode `getTypeA()`[34] definiert und die WFR wie folgt angepasst:

```
context Common-OCL::CommonModel::AttributeCallExp
inv: self.type = self.referredAttribute.getTypeA()
```

Die Spezifikationen für `getTypeA()` lauten wie folgt:

Für UML:

```
context UML::Core::Attribute
def: getTypeA() : OCL::CommonModel::Classifier = self.type.toOclType()
```

Für MOF:

```
context MOF::Model::Attribute
def: getTypeA() : OCL::CommonModel::Classifier = self.type.toOclType()
```

Es ist zu bemerken, dass zwar beide Definitionen gleich zu sein scheinen, es sich jedoch um Navigationen entlang zweier verschiedener Assoziationen handelt, einerseits von `UML::Core::Attribute` nach `UML::Core::Classifier` und andererseits von `MOF::Model::Attribute` nach `MOF::Model::Classifier`.

Durch den Aufruf von `toOclType()` wird in beiden Fällen die Abbildung in einen OCL-Typ entsprechend Kapitel 5.2.3 bzw. 5.2.4 durchgeführt.

5.3.2 Vervollständigung der WFRs

Die WFR für `AttributeCallExp` ist zugleich auch ein Beispiel für die Unvollständigkeit der WFRs in der OCL-Spezifikation:

---

[34] Die verwendete Namenskonvention wurde bereits im Kapitel 5.2.2 eingeführt. Der Suffix „A“ steht für „abstrakt“ und dient der Vermeidung von Namenskonflikten.

UML erlaubt für Attribute die Angabe einer Multiplizität und einer Ordnung. Dies wird jedoch in der WFR nicht berücksichtigt.

Ebenso ist es in MOF möglich, für Attribute Multiplizität und Ordnung anzugeben. Zusätzlich kann man noch spezifizieren, ob im Falle eines mehrwertigen Attributs die Werte eindeutig sein sollen.

Um dies zu berücksichtigen, wurden für `Attribute` zusätzlich folgende Operationen definiert (siehe auch Anlage C – Spezifikation der Operationen):

- `isMultipleA()`, `isUniqueA()`, `isOrderedA()`

Die WFR wurde wie folgt erweitert:

```
context Common-OCL::CommonModel::AttributeCallExp
inv: let attrType : Classifier = self.referredAttribute.getTypeA() in
    type = if attr.isMultipleA() then
               if attr.isUniqueA() then
                  if attr.isOrderedA() then attrType.getOrderedSetType()
                  else attrType.getSetType() endif
               else
                  if attr.isOrderedA() then attrType.getSequenceType()
                  else attrType.getBagType() endif
               endif
           else attrType  endif
```

Während die WFR für `AttributeCallExp` in der Spezifikation unvollständig ist, fehlen andere essentielle WFRs völlig. So ist z.B. keine WFR formuliert, die den Typ einer `AssociationEndCallExp` beschreibt.

Folgende Tabelle gibt eine Übersicht über vorgenommene Ergänzungen:

Klasse des OCL-Metamodells	**vorgenommene Ergänzung**
`AssociationClassCallExp`	WFR zur Beschreibung des Typs
`AssociationEndCallExp`	WFR zur Beschreibung des Typs
`AttributeCallExp`	Berücksichtigung von Multiplizität, Ordnung und Eindeutigkeit
`CollectionLiteralExp`	`OrderedSetType` ergänzt

`IteratorExp`	WFRs ergänzt für die Iteratoroperationen `one`, `any`, `collectNested` und `sortedBy`
`OclMessageExp`	WFR zur Beschreibung des Typs
`OclExpressionWithTypeArgExp`	WFR zur Beschreibung des Typs (s.a. Kapitel 5.4)
`OperationCallExp`	WFR zur Beschreibung des Typs

Abbildung 5-26: Vervollständigung der Wellformedness Rules

5.3.3 Typinferenz

Wie z.B. in Abbildung 5-8 ersichtlich, wird jedem OCL-Ausdruck ein Typ zugeordnet. Der größte Teil der in der OCL-Spezifikation angegebenen Wellformedness Rules für die Klassen des Paketes `Expressions` [OCL16,S.3-17ff] beschreibt für die Unterklassen von `OclExpression`, welchen Typ diese Ausdrücke haben. Der Typ hängt dabei entweder von den Typen der Teilausdrücke ab oder von Typinformationen aus dem UML-Modell.

Ersteres trifft z.B. für `IfExp` zu, deren Typ anhand der Typen der Teilausdrücke im Then- und Else-Zweig beschrieben wird:

```
context IfExp
inv: type = thenExpression.type.commonSuperType(elseExpression.type)
```

Letzteres ist der Fall für die Unterklassen von `ModelPropertyCallExp`, wie z.B. `AttributeCallExp` (siehe voriger Abschnitt).

Festzuhalten ist, dass in einer wohlgeformten Instanz des OCL-Metamodells die Typinformationen vorhanden sein müssen. Folglich müssen diese im Rahmen des Abstract Syntax Mapping – also vom Parser – ermittelt werden. Ob dies nun sukzessive während des Parsens geschieht[35], oder ob der Parser in einem ersten Pass zunächst einen abstrak-

[35] Die in der OCL-Spezifikation angegebenen Regeln für das Abstract Syntax Mapping legen diese Vorgehensweise nahe. So wird z.B. für eine `OperationCallExp` [OCL16, S.4-16] zunächst der Typ des Teilausdrucks links vom Operationsaufruf ermittelt, um für diesen dann `lookupOperation()` aufzurufen und somit die aufzurufende Operation im UML-Modell zu finden.

ten Syntaxbaum ohne Typinformationen erstellt und diese in einem zweiten Pass hinzugefügt werden[36], soll hier nicht Gegenstand tiefergehender Erörterung sein.

Der Prozess des Ermittelns von Typinformationen durch einen Compiler wird als Typinferenz bezeichnet.

Zwar wird im Rahmen der vorliegenden Arbeit der Parser selbst nicht betrachtet, jedoch ist es auch zweckmäßig, für „manuell“ erstellte Instanzen des OCL-Metamodells zumindest die Typinformation automatisch zu ermitteln. Daher wurde der Visitor [Gam95] **TypeEvaluator** für das OCL-Metamodell implementiert. Diese Klasse basiert auf einem naiven Typinferenz-Algorithmus, der den Typ einer `OclExpression` bottom-up[37] bestimmt.

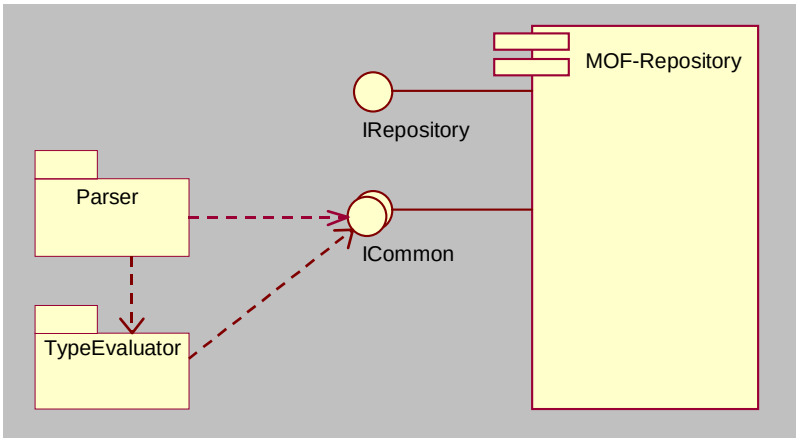


Abbildung 5-27: Typinferenz in der Compilerarchitektur

Typinferenz und Mehrfachvererbung

Während die meisten WFRs des `Expression`-Paketes den Typ eines OCL-Audrucks eindeutig beschreiben (z.B. obige WFR für `AttributeCallExp`), ist dies für einige nicht der Fall. Dies betrifft konkret `IfExp` und `CollectionLiteralExp`. Beide nutzen die Operation `commonSuperType()`, die den kleinsten[38] gemeinsamen Supertyp von zwei Typen ermittelt. Wie Abbildung 5-28 zeigt, kann dieser jedoch im Falle von Mehrfachvererbung nicht immer eindeutig bestimmt werden.

---

[36] Dies ist z.B. das Vorgehen in [Fin00].
[37] also indem er zunächst rekursiv die Typen der Teilausdrücke ermittelt.
[38] d.h. den in der Vererbungshierarchie am tiefsten gelegenen.

Die Spezifikation von `commonSuperType()` [OCL16,S.3-22] trifft in diesem Fall nur die Aussage: Eine von den beiden Klassen `S1` und `S2` ist der kleinste gemeinsame Supertyp von `A` und `B`.

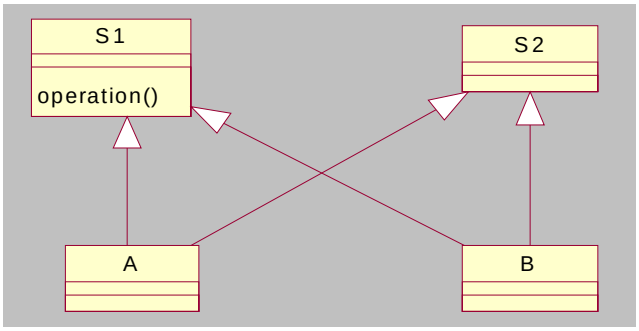


Abbildung 5-28: Multiple kleinste gemeinsamer Supertypen

Solche Uneindeutigkeiten können nur durch elaboriertere Typinferenz-Algorithmen korrekt behandelt werden. Betrachtet man z.B. den Ausdruck

```
(if true then A else B endif).operation(),
```

so kann zwar der Typ der `IfExp` nur `S1` sein, da `operation()` für `S1` definiert ist. Um dies jedoch automatisch zu ermitteln, reicht es nicht aus, nur bottom-up die Typen der Teilausdrücke der `IfExp` zu betrachten. Vielmehr muss auch der Kontext[39] des Ausdrucks mit einbezogen werden.

Die Beschränkung des hier implementierten Typinferenz-Algorithmus bringt jedoch keine wesentlichen Nachteile mit sich. Durch Nutzung von `oclAsType()` können Ausdrücke wie der obige so formuliert werden, dass keine Uneindeutigkeiten bei der Ermittlung des Typs auftauchen:

```
(if true then A.oclAsType(S1) else B.oclAsType(S1) endif).operation()
```

5.4 OCL Standard Library

Die OCL Standard Library enthält die vordefinierten OCL-Typen und ihre Operationen. Im Folgenden soll zunächst ein Überblick über diese Typen und deren Modellierung in `UML-OCL` und `MOF-OCL` gegeben werden. Anschließend wird in Abschnitt 5.4.2 die Beziehung zwischen der OCL Standard Library und dem UML-Modell bzw. dem

[39] Hier ist nicht der Kontext eines OCL-Constraints gemeint, also z.B. eine Klasse oder eine Operation, sondern vielmehr der übergeordnete Knoten im abstrakten Syntaxbaum.

Metamodell beschrieben. Abschnitt 5.4.3 beschäftigt sich speziell mit den Operationen `oclAsType`, `oclIsTypeOf` und `oclIsKindOf`.

5.4.1 Die Typen der OCL Standard Library

Tabelle 5-5 zeigt die Typen der OCL Standard Library. Diese Typen sind Instanzen von Klassen des OCL-Metamodells. Da Klassen in `Common-OCL` – wie in Kapitel 5.2.2 erläutert – nicht instantiiert werden, sind die Metaklassen der OCL-Typen für UML und für MOF unterschiedlich.

Für UML sind die OCL-Typen Instanzen von Klassen aus den Paketen `UML::Core` und `UML-OCL::Types`.

Für MOF hingegen entstammen die Metaklassen den Paketen `MOF-OCL::Types` und `MOF::Model`. Der Adapter `MOF-OCL::Adapters:AdPrimitive` ist die Metaklasse für primitive Typen. Siehe hierzu auch Kapitel 5.2.3.

OclTyp	UML-OCL Metaklasse	MOF-OCL Metaklasse
OclAny	UML::Core::Class[40]	MOF::Model::Class
OclVoid	UML-OCL::Types::VoidType	MOF-OCL::Types::VoidType
Real, Integer, Boolean, String	UML::Core::Primitive	MOF-OCL::Adapters:AdPrimitive
Collection	UML-OCL::Types::CollectionType	MOF-OCL::Types::CollectionType
Bag	UML-OCL::Types::BagType	MOF-OCL::Types::BagType
OrderedSet	UML-OCL::Types::OrderedSetType	MOF-OCL::Types::OrderedSetType
Sequence	UML-OCL::Types::SequenceType	MOF-OCL::Types:: SequenceType
Set	UML-OCL::Types::SetType	MOF-OCL::Types::SetType
OclMessage	UML-OCL::Types::OclMessageType	MOF-OCL::Types::OclMessageType
Tuple[41]	UML-OCL::Types::TupleType	MOF-OCL::Types::TupleType

Tabelle 5-5: Die Typen der OCL Standard Library und ihre Metaklassen

[40] [OCL16, S.6-1] gibt `Classifier` als Metaklasse von `OclAny` an. Doch `Classifier` ist abstrakt.
[41] Fehlt in [OCL16, S.6-2 ff].

5.4.2 Einbindung der OCL Standard Library in das Modell

Die Typen der OCL Standard Library residieren in einem separaten Paket, das mangels anderer Angaben in [OCL16] den Namen „OclLib“ trägt.

Dieses Paket wird nach dem Laden eines Modells in das Repository erstellt und dann dem Modell hinzugefügt. Anschließend werden `OclAny`, `OclVoid` und die primitiven Typen erzeugt.

Da `OclAny` Supertyp aller Typen, also auch aller Klassen des Modells ist, werden nun noch Vererbungsbeziehungen zwischen `OclAny` und allen Klassen des Modells, die keine Oberklasse haben, hergestellt. Analog hierzu wird allen Klassen, die keine Unterklassen besitzen, `OclVoid` als solche zugeordnet. Somit wird `OclVoid` zum gemeinsamen Subtyp aller anderen Typen.

Die Kollektions- und Tupeltypen sowie die `OclMessage`-Typen werden nur bei Bedarf erzeugt, z.B. durch den `Parser` oder den `TypeEvaluator`. Eine andere Vorgehensweise ist nicht möglich, da es unendlich viele solcher Typen gibt. So lassen sich beispielsweise Kollektionen beliebig tief verschachteln.

Kollektionstypen können mit Hilfe der für `Classifier` definierten Operationen `getSetType, getBagType, getOrderedSetType, getSequenceType` ermittelt werden. Diese liefern für einen `Classifier` c die entsprechenden Kollektionstypen, die c als Elementtyp haben. Sind diese Typen noch nicht im `OclLib`-Paket vorhanden, werden sie erstellt.

Tupeltypen und `OclMessage`-Typen können durch Nutzung der entsprechenden Operationen der Hilfsklasse `OclLibrary` erzeugt werden (siehe auch Anlage C – Spezifikation der Operationen, Seite 128).

5.4.3 Operationen mit Typargument

Eine Besonderheit stellen die drei Operationen `oclAsType`, `oclIsTypeOf` und `oclIsKindOf` dar, da sie als Argument einen Typ erwarten. [OCL16] enthält widersprüchliche Angaben darüber, wie diese Operationen zu modellieren sind:

(1) Laut [OCL16,S.6-3f] werden sie als Operationen von `OclAny` definiert und können somit von einer `OperationCallExp` referenziert werden. Das Problem des Typarguments wird dadurch gelöst, dass eine Enumeration **`OclType`**

definiert wird, die für jeden Typ des Modells (also für jeden `Classifier`) ein Literal enthält. Das Argument der Operationen hat diese Enumeration `OclType` als Typ.

(2) Laut dem Semantik-Kapitel [OCL16,S.A-25] wird eine neue Unterklasse von `OclExpression` namens `OclOperationWithTypeArgument` eingeführt. Jeder Aufruf von `oclAsType`, `oclIsTypeOf` und `oclIsKindOf` entspricht einer Instanz von `OclOperationWithTypeArgument`.

Im Folgenden werden Gründe genannt, warum hier Variante (2) umgesetzt wurde. Anschließend wird die entsprechende Änderung des OCL-Metamodells beschrieben:

- Die Literale der `OclType`-Enumeration entsprechen im Prinzip dem Konzept der Fremdschlüssel aus der Welt der relationalen Datenbanken. In der Welt der objektorientierten Modellierung wird eine Beziehung zwischen zwei Klassen jedoch durch Assoziationen und nicht durch Fremdschlüssel ausgedrückt.
- Betrachtet man z.B. den Ausdruck `self.oclIsKindOf(Integer)`, so sieht man, dass das Typargument durchaus nicht der konkreten Syntax einer `EnumerationLiteralExp` entspricht. Laut Variante (1) müsste es eigentlich heißen: `self.oclIsKindOf(OclType::Integer)`.
- Variante (2) ist analog zu dem Vorgehen, dass im Falle von `IteratorExp` und `IterateExp` angewandt wurde: Da einer Operation als Argument keine `OclExpression` übergeben werden kann, wurden `iterate`, `forAll` u.s.w. nicht als Operationen auf den Kollektionstypen definiert, sondern es wurden `IteratorExp` und `IterateExp` als Unterklassen von `OclExpression` eingeführt.

Das OCL-Metamodell wurde um die Klasse `OclOperationWithTypeArgExp` erweitert. Um das Typargument zu modellieren, besitzt diese eine Assoziation zu `Classifier`.

Für `OclOperationWithTypeArgExp` wurden außerdem WFRs hinzugefügt (siehe Anlage B – Wellformedness Rules).

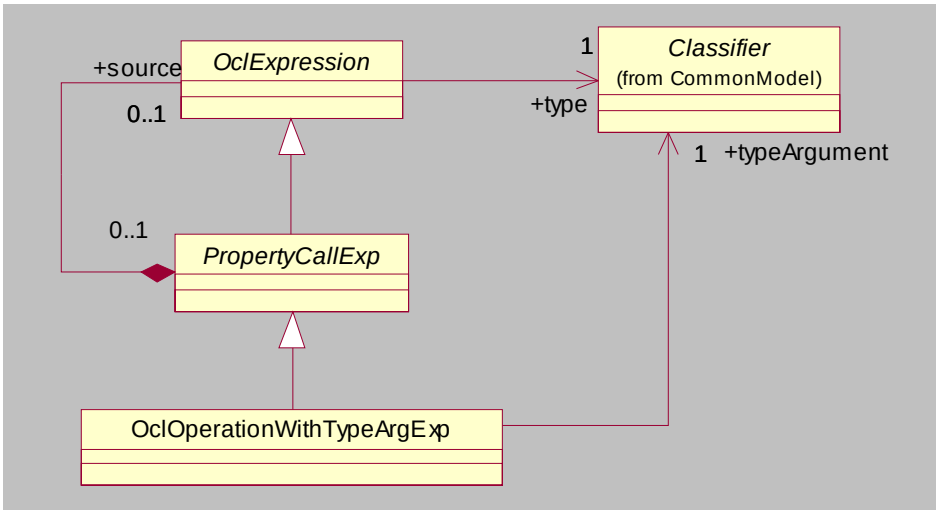


Abbildung 5-29: OclOperationWithTypeArgExp

5.5 Anpassung der OCL-Basisbibliothek

Die im Rahmen von [Fin99] entstandene OCL-Basisbibliothek ist eine Java-Implementation der Typen der OCL Standard Library und ihrer Operationen.

Bereits bei der in [Fin00] vorgestellten Code-Generierung für OCL-Constraints in UML-Modellen wird von der OCL-Basisbibliothek Gebrauch gemacht. Die dort verwendete Implementation nutzt Java-Reflektion, um auf Modellinformation zuzugreifen.

Für die Code-Generierung OCL-Ausdrücke über Metamodellen wird eine Implementation der Bibliothek benötigt, die auf Modelle (M1) im Repository per reflektiven JMI-Schnittstellen zugreift.

Die eigentlichen Implementationen der OCL-Typen, also Klassen wie `OclInteger`, `OclBoolean` oder `OclSet`, konnten weitestgehend unverändert verwendet werden. Doch es waren einige strukturelle Änderungen an der Bibliothek notwendig, da sie teilweise zuwenig Abstraktion verwendet, um z.B. den Modellzugriff per JMI einfach als Alternative zum Modellzugriff per Java-Reflektion implementieren zu können.[42] So bestehen beispielsweise über die Klassen der Bibliothek gestreut Abhängigkeiten von der Klasse `OclAnyImpl`, die den Modellzugriff per Java-Reflektion implementiert.

[42] Im Folgenden wird oft von der „Implementation der OCL-Basisbibliothek“ gesprochen. Gemeint ist damit immer die Implementation des Modellzugriffs.

Ein weiteres Problem bestand bezüglich der Klasse `OclType`. Diese ist als Wrapper um die Java-Klasse `Class` implementiert. Dies reicht jedoch nicht aus, um z.B. verschachtelte Kollektionstypen oder Tupeltypen zu beschreiben.

Die strukturellen Änderungen zur Lösung dieser Probleme sowie vorgenommene Erweiterungen sollen im Folgenden beschrieben werden.

5.5.1 Repräsentationen der Ocl-Typen

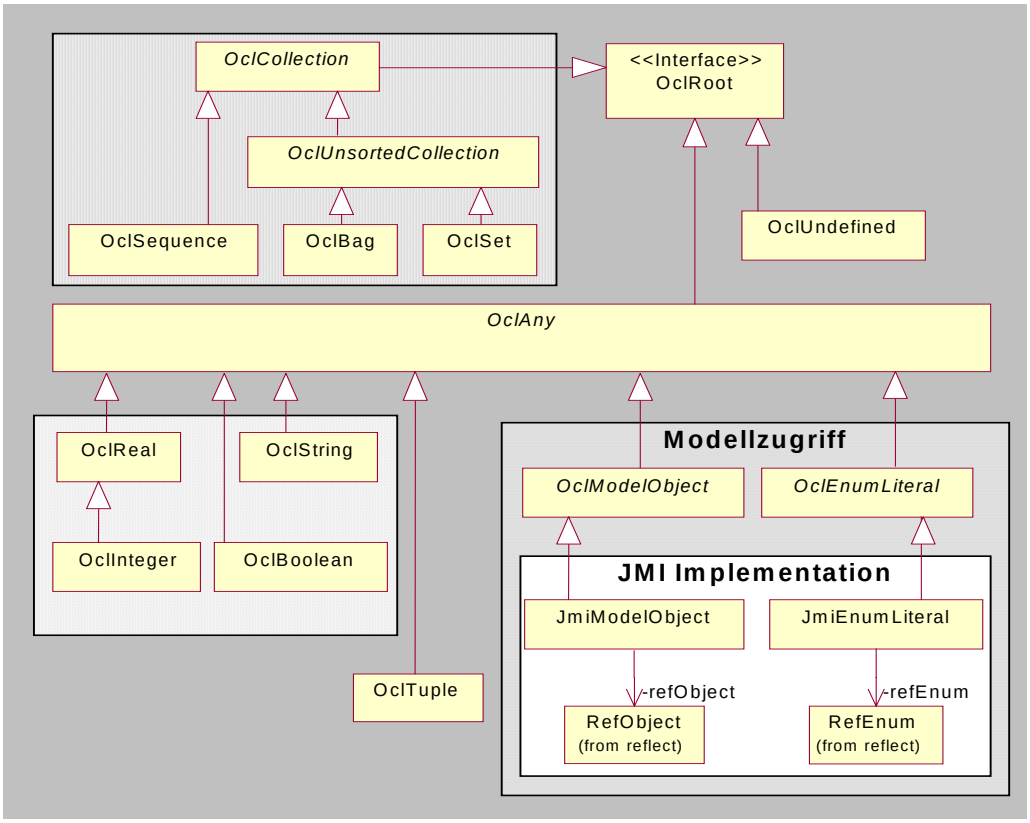


Abbildung 5-30: Repräsentationen der OCL-Typen und JMI Implementation

`OclAnyImpl` wurde durch die Klasse `OclModelObject` ersetzt. Diese enthält Template-Methoden für den Modellzugriff. Die eigentliche Implementation erfolgt in Unterklassen. Im Falle von JMI ist dies die Klasse `JmiModelObject`, die die reflektive JMI-Schnittstelle `RefObject` (s.a. Kapitel 2.3) nutzt, um für ein Objekt des Modells Attributwerte abzufragen, zu anderen Objekten zu navigieren oder Operationen aufzurufen.[43]

[43] Analog hierzu muss die Implementation eines Modellzugriffs per Java-Reflektion ebenfalls in einer Unterklasse von `OclModelObject` geschehen. Dies wurde jedoch hier noch nicht verwirklicht.

Neben dieser strukturellen Änderung wurden noch `OclTuple` und `OclEnumLiteral` neu hinzugefügt.

5.5.2 Typschema

Die Klasse `OclType` erbt nun nicht mehr von `OclAny`. Stattdessen ist sie Oberklasse im Typschema der Basisbibliothek:

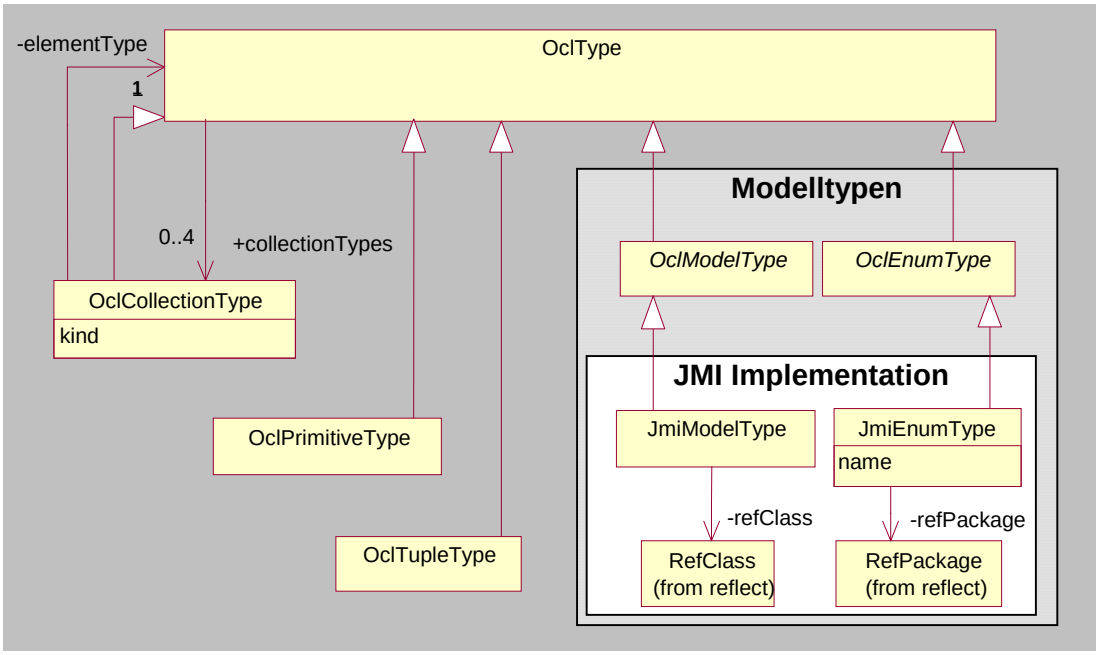


Abbildung 5-31: Typschema und JMI Implemetation

Dies entspricht in etwa dem `Types`-Paket des OCL-Metamodells, mit dem Unterschied, dass hier Typen aus dem Modell leichter von Typen aus der OCL Standard Library unterschieden werden können: Für Typen aus dem Modell existieren die separaten Unterklassen `OclModelType` und `OclEnumType`.

`OclType` und Unterklassen werden in der OCL-Basisbibliothek benutzt...

- für den Zugriff auf Klassenoperationen und Klassenattribute
- als Parameter für die Methoden `oclAsType()`, `oclIsTypeOf()` und `oclIsKindOf()` in `OclAny`.
- für die explizite Angabe des Types bei den Abbildungen:
    - Modellobjekt → Repräsentation in der OCL-Basisbibliothek (`OclRoot`)
    - Repräsentation in der OCL-Basisbibliothek (`OclRoot`) → Modellobjekt.

Diese Abbildungen sollen im Folgenden näher erläutert werden.

5.5.3 Abbildung von Modellobjekten

Objekte, die bei einem Zugriff auf das Modell zurückgegeben werden, müssen in die entsprechenden Repräsentationen in der OCL-Basisbibliothek konvertiert werden. So muss z.B. für eine Instanz von `RefObject` ein entsprechendes `JmiModelObject` erzeugt werden (s.a. Abbildung 5-30).

Dieses Beispiel zeigt, dass die erforderlichen Konvertierungen von der jeweiligen Implementation der OCL-Basisbibliothek (Modellzugriff per JMI-Reflektion oder per Java-Reflektion) abhängen. Um von der Implementation zu abstrahieren, wird wie folgt verfahren:

- Die Template-Methoden für den Modellzugriff in `OclModelObject` nutzen die Methode `getOclRepresentation()` der Schnittstelle `OclFactory`.
- Für jede Art des Modellzugriffs ist eine eigene Implementation dieser Fabrik bereitzustellen. Im Falle von JMI ist dies `JmiOclFactory`.

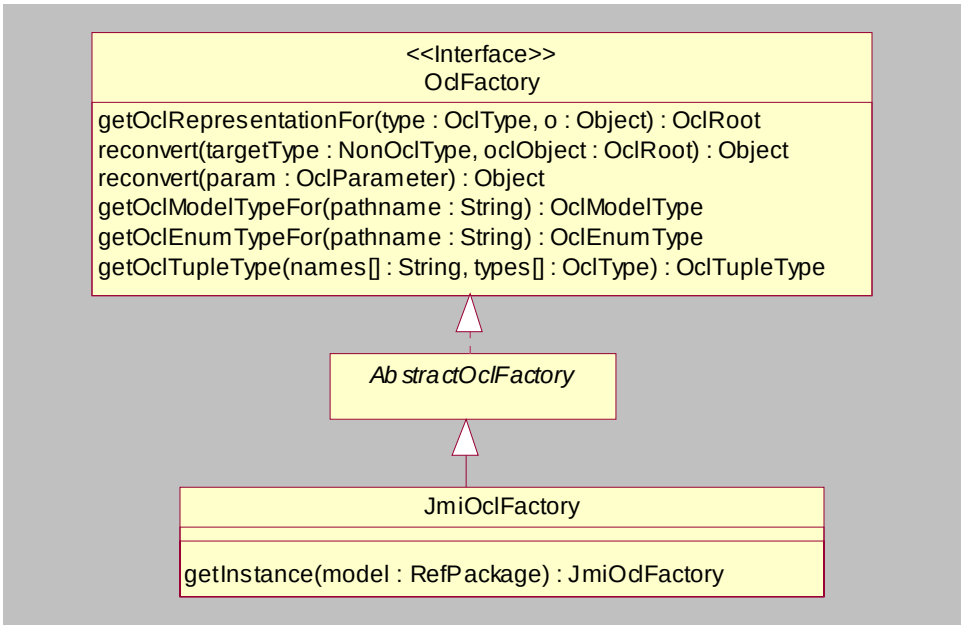


Abbildung 5-32: JmiOclFactory

Im Gegensatz zur ursprünglichen OCL-Basisbibliothek wurde jedoch die Entscheidung getroffen, dass der Methode `getOclRepresentation()` Typinformation übergeben werden muss.

Dies soll anhand des folgenden Beispiels erläutert werden:

- Im Modell existieren die Klassen `A` und `B`. Die Klasse `A` hat ein Attribut `b` vom Typ `B`. Dieses Attribut `b` hat die Multiplizität 0..*, ist ungeordnet und es sind Duplikate erlaubt.

Gemäß der WFR für `AttributeCallExp` ist der Typ eines Zugriffs auf dieses Attribut `Bag(B)`.

Der reflektive Zugriff auf das Attribut `b` per JMI würde einfach eine Instanz von `java.util.Collection` ergeben. Folglich wird Typinformation gebraucht, um die Entscheidung zu treffen, ob diese Collection nun in ein `OclSet` oder in ein `OclBag` zu wandeln ist.

Der entsprechende Code für den Attributzugriff hat folgendes Aussehen:

```
OclCollectionType bagOfBType= factory.getOclModelTypeFor("B").getOclBagType();
OclBag bagOfB = Ocl.toOclBag(a.getFeature(bagOfBType, "b"));
```

Hierbei ist `a` eine Instanz von `OclModelObject`, die eine Instanz der Modellklasse `A` repräsentiert. Die Methode `getFeature()` ruft für die Konvertierung des Ergebnisses des Modellzugriffs die Methode `getOclRepresentation()` der `OclFactory` auf.

Die Notwendigkeit der expliziten Typangabe besteht umgekehrt auch dann, wenn eine Instanz von `OclRoot` wieder in ein Modellobjekt konvertiert werden soll. Dies ist nötig, wenn beim Aufruf von Operationen des Modells Argumente übergeben werden.

Liegt z.B. eine Instanz der Klasse `OclInteger` vor, so wäre im Falle von JMI zu entscheiden, ob eine Konvertierung nach `java.lang.Integer` oder `java.lang.Long` erfolgen soll[44].

Für derartige Konvertierungen ist in der `OclFactory` die Methode `reconvert()` definiert. Die Typinformation wird dabei als Instanz von `NonOclType` übergeben. Wie diese Schnittstelle zu implementieren ist, steht völlig frei. Lediglich die entsprechende Implementation der `OclFactory` muss das so definierte Typschema „verstehen“.

Abbildung 5-33 zeigt die Implementation von `NonOclType`, die für die `JmiOclFactory` Anwendung findet.

---

[44] oder gar nach Double oder Float, denn aufgrund der Typkonformanz von Integer zu Real wäre dies ebenfalls möglich.

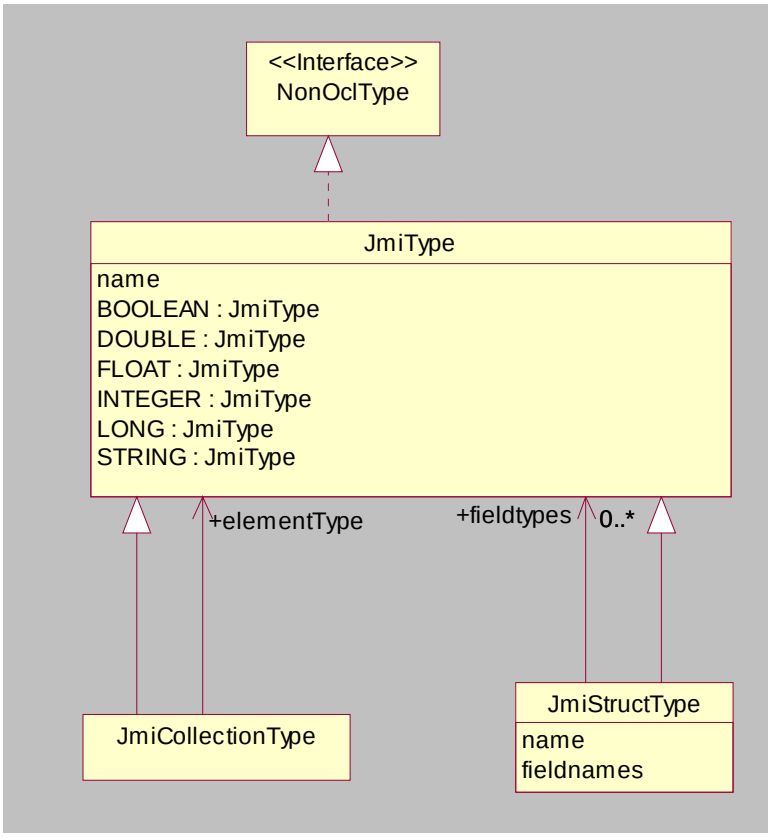


Abbildung 5-33: JmiType

5.6 Code-Generierung

Aufgabe der Code-Generierung ist es, eine OCL-Metamodellinstanz (im Folgenden auch OCL-Model genannt) in Code zu transformieren, der die im vorigen Abschnitt vorgestellte JMI-Implementation der OCL-Basisbibliothek nutzt, um OCL-Ausdrücke für ein gegebenes Modell (M1) auszuwerten.

Entsprechend Kapitel 2.6 besteht die zusätzliche Anforderung, dass die Transformation wiederverwendbar ist für die Code-Generierung für OCL-Ausdrücke in UML-Modellen. Dies wird vor allem dadurch erreicht, dass der eigentliche Modellzugriff in der Implementation der OCL-Basisbibliothek gekapselt ist. Um also wie in [Fin00] Code zu generieren, der auf Objekte (M0) mittels Java-Reflektion zugreift, muss es hauptsächlich möglich sein, die vom generierten Code benutzte Implementation der OCL-Basisbibliothek konfigurieren zu können.

Im Folgenden werden zunächst zwei mögliche Ansätze zur Transformation eines OCL-Modells in Java-Code vorgestellt. Danach wird die Spezifikation der Codegenerierung

erläutert. Abschließend werden die Artefakte der Implementation vorgestellt, die die Code-Generierung entsprechend dieser Spezifikation umsetzen.

5.6.1 Ansätze

Es werden folgende Ansätze diskutiert:

(1) Transformation des OCL-Modells in ein Zwischenmodell. Anschließende Transformation des Zwischenmodells in Java-Code.

(2) Direkte Transformation des OCL-Modells in Java-Code.

Hauptmotivation für Ansatz (1) ist, dass aus dem Zwischenmodell auch Code für OCL-Basisbibliotheken in einer anderen Programmiersprache als Java generiert werden könnte. Durch das Aufbrechen der Transformation ist es außerdem möglich, statt einer potentiell sehr komplexen und unverständlichen Transformationsspezifikation, wie sie im Fall von Ansatz (2) wahrscheinlich ist, zwei einfachere anzugeben.

Kernfrage ist jedoch, welches Metamodell für das Zwischenmodell geeignet ist. Einerseits sollte die Transformation des Zwischenmodells in Java-Code möglichst einfach sein, andererseits muss das Zwischenmodell noch allgemein genug sein, um daraus z.B. Code für eine in C++ implementierte OCL-Basisbibliothek zu generieren.

Um also ein Metamodell geeigneter Granularität entwerfen zu können, wäre es zumindest nötig, potentielle Unterschiede zwischen OCL-Basisbibliotheken in verschiedenen Programmiersprachen zu eruieren. Der Entwurf eines solchen Metamodells erfordert also einen hohen zusätzlichen Aufwand. Daher wurde in der vorliegenden Arbeit Ansatz (2) umgesetzt.

5.6.2 Spezifikation

Die Spezifikation von Transformationen von Modellen im Rahmen der MOF-Metadatenarchitektur ist Gegenstand einer sich gegenwärtig in der Entstehung befindlichen OMG-Spezifikation namens *MOF 2.0 Query / Views / Transformations* [QVT]. Die Bedeutung der Begrifflichkeiten ist hierbei wie folgt definiert:

- **Queries**: Anfragen an Modelle, um bestimmte Modellelemente auszuwählen. Als Anfragesprache wird in QVT mit hoher Wahrscheinlichkeit OCL selbst Anwendung finden.

- **Views**: Modelle, die aus anderen Modellen abgeleitet sind (z.B. sind die Ergebnisse von Transformationen Views).
- **Transformations**: Spezifikationen, die ein Modell mit einem anderen in Relation setzen oder aus einem Modell ein anderes erzeugen.

In diesem Sinne ist die im vorigen Abschnitt unter Ansatz (1) postulierte Transformation in ein Zwischenmodell eine Transformation im Sinne von QVT.

Das direkte Generieren von Code aus einem Modell kann ebenfalls als eine Transformation angesehen werden. Diese hat ein Modell als Ergebnis, das nur aus Zeichenketten besteht.

Um Instanzen von `String` zu erzeugen, bedarf es jedoch keiner speziellen Transformationssprache. Dies ist auch mit Hilfe der Anfragesprache OCL möglich. Die Code-Generierung lässt sich also im weitesten Sinne als eine Anfrage (Query) an das Modell auffassen, die als Ergebnis eine Zeichenkette hat.

Hier wurde daher der Weg eingeschlagen, die gesamte Code-Generierung in OCL zu spezifizieren. Ein Vorteil dieser Vorgehensweise ist das mögliche Bootstrapping des Code-Generators: Bei späteren Änderungen an der Spezifikation kann der OCL-Compiler benutzt werden, um den Code-Generator aus der Spezifikation zu generieren.

Ein Nachteil ist die relativ schlechte Verständlichkeit der Regeln z.B. im Vergleich zur Verwendung von Templates. Um hierfür Abhilfe zu schaffen, wird eine Hilfsklasse namens `Template` verwendet. Diese ermöglicht es z.B. anstatt

```
'final '.concat(typename).concat(' ').concat(id).concat(' = ')
```

einfach zu schreiben:

```
Template::fill('final $1 $2 =', Sequence{typename, id})
```

Die komplette Spezifikation findet sich in Anlage D – Spezifikation der Codegenerierung. Ausgangspunkt für eine Codegenerierung ist dabei immer eine einzelne `OclExpression`.

Im Folgenden soll die Spezifikation anhand der Regel für `IfExp` näher erläutert werden. Zunächst die Regel und der entsprechende Ausschnitt aus dem OCL-Metamodell im Überblick:

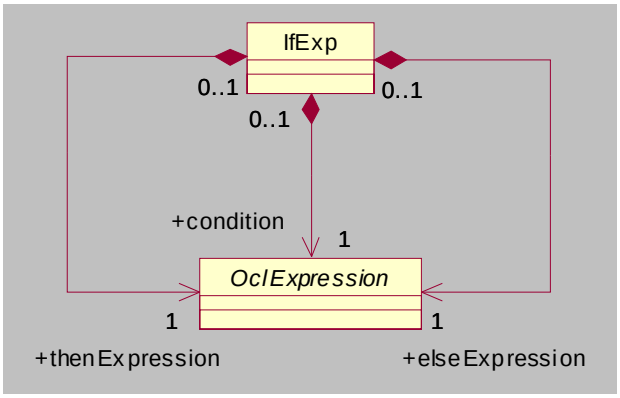


Abbildung 5-34: IfExp

```
context IfExp
def: appendJavaCode(env: Env) : Env =
let javaOclType : String = self.type.mapToJavaOcl(env, true) in
let withCond : Env = self.condition.appendJavaCode(env) in
let withThen : Env = self.thenExpression.appendJavaCode(withCond) in
let withElse : Env = self.elseExpression.appendJavaCode(withThen) in
let withId : Env = withElse.createId(self) in
withId.appendLine('final $1 $2 = $3;',
    Sequence{javaOclType, withId.getId(self),
        Cast::withCast(javaOclType,'$1.ifThenElse($2,$3)',
                                    Sequence{withId.getId(condition),
                                             withId.getId(thenExpression),
                                             withId.getId(elseExpression)})})
```

Nun die Regel im Einzelnen:

```
context IfExp
def: appendJavaCode(env: Env) : Env =
```

Die Code-Generierung für eine `IfExp` (und für alle anderen Unterklassen von `OclExpression` auch) wird in Form der Operation `appendJavaCode()` definiert. Diese erwartet einen Parameter vom Typ `Env` (Umgebung) und hat auch den Ergebnistyp `Env`.

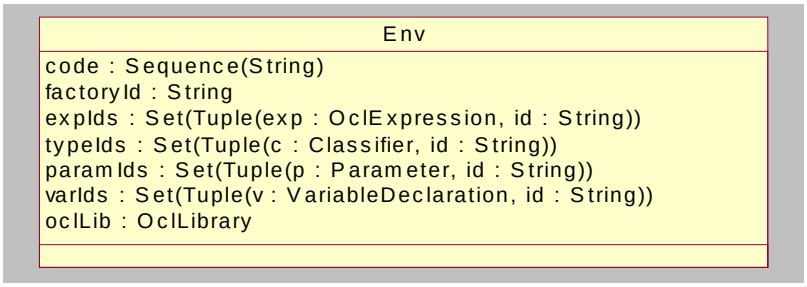


Abbildung 5-35: Umgebung für die Code-Generierung

Zweck der Umgebung ist es, den Zustand der Codegenerierung mitzuführen. Das ist einerseits der bisher erzeugte Code. Andererseits sind das alle Bezeichner (Identifier), die bereits für Variablen oder die Ergebnisse anderer Teilausdrücke erzeugt wurden.

```
let javaOclType : String = self.type.mapToJavaOcl(env, true) in
```

Hier wird der Name der Klasse der OCL-Basisbibliothek ermittelt, welche den Typ der `IfExp` repräsentiert. Hat z.B. die `IfExp` den Typ `Set(Integer)`, so wäre das Ergebnis "OclSet".

Zur Ermittlung dieses Namens wird die für `Classifier` definierte Operation `mapToJavaOcl()` verwendet (s.a. Anlage D – Spezifikation der Codegenerierung, S.145).

```
let withCond : Env = self.condition.appendJavaCode(env) in
let withThen : Env = self.thenExpression.appendJavaCode(withCond) in
let withElse : Env = self.elseExpression.appendJavaCode(withThen) in
```

Dies erzeugt den Code für die drei Teilausdrücke der `IfExp`.

```
let withId : Env = withElse.createId(self) in
```

Der Umgebung wird ein Bezeichner für die `IfExp` hinzugefügt. Man beachte, dass die Operation `createId()` und auch alle anderen Operationen für `Env` seiteneffektfrei spezifiziert sind (s. Anlage D – Spezifikation der Codegenerierung, S.132).

```
withId.appendLine('final $1 $2 = $3;',
    Sequence{javaOclType, withId.getId(self),
        Cast::withCast(javaOclType,'$1.ifThenElse($2,$3)',
                                    Sequence{withId.getId(condition),
                                             withId.getId(thenExpression),
                                             withId.getId(elseExpression)})})
```

Die Operation `appendLine()` fügt der Umgebung eine Code-Zeile hinzu. Da sie die Methode `Template::fill()` benutzt, erhält sie als erstes Argument ein Code-Template und als zweites die Liste der Template-Argumente.

Da der Code für eine `IfExp` die Methode `ifThenElse()` benutzt, die als Rückgabetyp `OclRoot` hat, ist es nötig, einen Type-Cast einzufügen. Dies wird durch die Hilfsoperation `Cast::withCast` realisiert.

Die Operation `getId()` liefert den in der Umgebung vorhandenen Bezeichner für einen Teilausdruck, für den schon der Code erzeugt und `createId()` aufgerufen wurde.

Initialisierung der Umgebung für JMI

Damit vom generierten Code die JMI-Implementierung der OCL-Basisbibliothek benutzt wird, ist die Codegenerierung mit folgender Umgebung zu beginnen:

```
Env::init(
    Sequence{'JmiOclFactory tudOcl20Fact = JmiOclFactory.getInstance(model)'},
    'tudOcl20Fact',
    metamodel
)
```

Dies liefert eine Umgebung, die bereits den Code für die Instantiierung der `JmiOclFactory` enthält. Außerdem wird der Bezeichner für diese Factory festgelegt. An allen Stellen im Code, wo eine `OclFactory` benötigt wird, wird die Zeichenkette "tudOcl20Fact" eingefügt.

Für die Verwendung einer anderen Implementation der `OclLibrary` muss eine entsprechend geänderte Anfangsumgebung verwendet werden.

5.6.3 Implementation

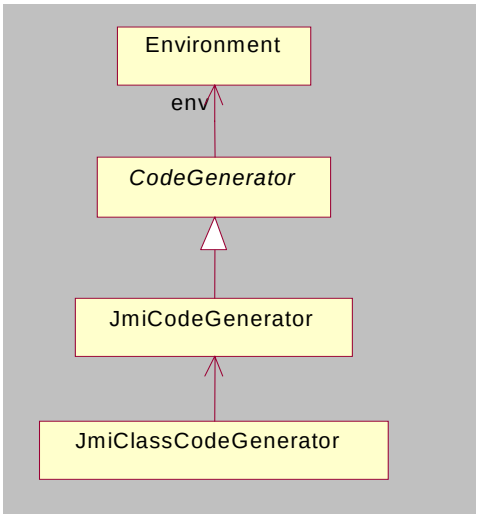


Abbildung 5-36: Implementation der Code-Generierung

Die spezifizierten Regeln für die Code-Generierung wurden in der Klasse `CodeGenerator` umgesetzt. Diese ist als Visitor [Gam95] des OCL-Metamodells

implementiert. Die konkrete Unterklasse `JmiCodeGenerator` startet die Code-Generierung unter Verwendung der `JmiOclFactory`.

`JmiClassCodeGenerator` generiert Code für alle Invarianten einer Klasse und fasst diese in einer `OclEval`-Klasse zusammen. Dabei wird für jede Invariante eine eigene Methode erzeugt. Folgender Codeausschnitt illustriert dies am Beispiel einer Klasse `ClassA` mit den Invarianten `WFR1` und `WFR2`:

```
public class OclEvalClassA{
   private RefPackage model;

   public OclEvalClassA(RefPackage model){ this.model=model; }

   //invariant WFR1 : ...
   public boolean evaluateWFR1(){
      ...//von JmiCodeGenerator erzeugter Code
   };

   //invariant WFR2 : ...
   public boolean evaluateWFR2(){
      ...//von JmiCodeGenerator erzeugter Code
   };
}
```

Wie man sieht, enthält die Klasse `OclEvalClassA` keine Referenz auf eine Instanz von `ClassA`. `JmiClassCodeGenerator` kann also grundsätzlich noch nicht mit freien Variablen wie `self` umgehen. Vielmehr wird davon ausgegangen, dass Invarianten von `ClassA` bei der Abbildung von konkreter in abstrakte Syntax eingebettet werden in:

```
ClassA::allInstances()->forAll(self : ClassA |...)
```

Die Methoden `evaluateWFR1()` und `evaluateWFR2()` durchlaufen also alle Instanzen von `ClassA`.

In Kapitel 7.4 von [OCL16] sind noch keine näheren Angaben darüber vorhanden, wie Invarianten in abstrakter Syntax dargestellt werden. Im Semantik-Kapitel [OCL16,S.A-29] wird jedoch die Äquivalenz zwischen einer Invariante und einem entsprechenden OCL-Ausdruck ohne freie Variablen unter Verwendung von `allInstances()` beschrieben.

Andere Arten der Verwendung von OCL-Ausdrücken, wie z.B. Vor- und Nachbedingungen für Operationen oder Definitionen von Operationen, werden von `JmiClassCodeGenerator` noch nicht unterstützt.

5.7 OCL Workbench

Um die Funktionalität des Compilers und insbesondere des Code-Generators zu demonstrieren, wurde ein GUI[45] namens OCL Workbench implementiert, welches das in Abbildung 5-5 dargestellte Szenario unterstützt. Konkret bietet die Workbench folgende Funktionen an:

(1) Laden von Metamodellen in das Repository per XMI.

(2) Erzeugen von OCL-Invarianten für diese Metamodelle.

(3) Generieren von Code für die Invarianten.

(4) Laden von Modellen und Ausführen des generierten Codes, um so die Einhaltung der Invarianten in den Modellen zu prüfen.

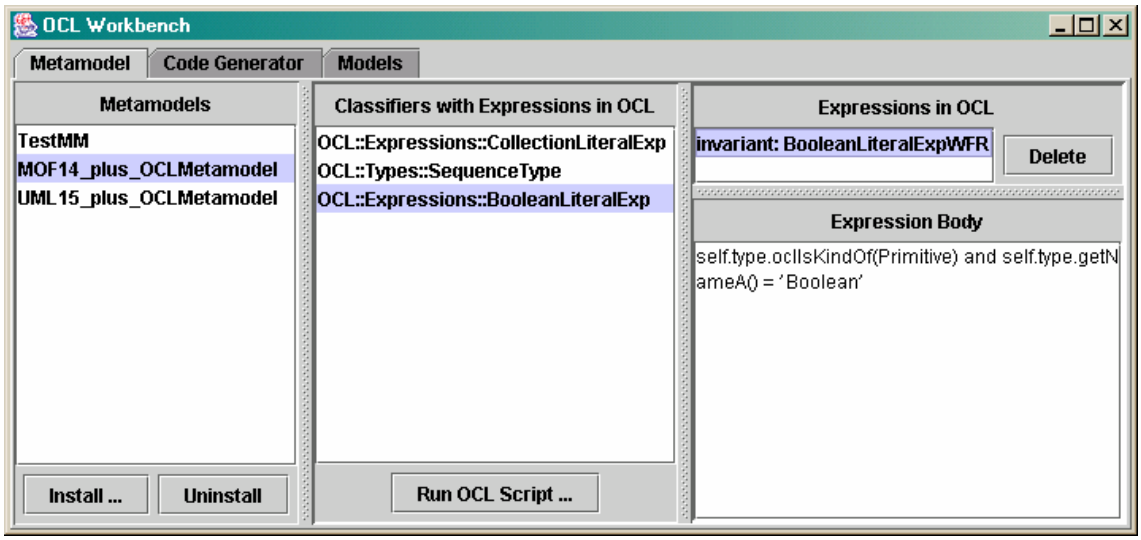


Abbildung 5-37: OCL Workbench

Für Punkt (2) ergab sich aufgrund der Tatsache, dass der Parser noch nicht implementiert ist, die Notwendigkeit für eine temporäre Ersatzlösung für das Erstellen von OCL-Ausdrücken in abstrakter Syntax.

[45] GUI: **G**raphical **U**ser **I**nterface

Daher wurde eine einfache Skriptsprache entwickelt, deren einzelne Anweisungen jeweils eine Instanz einer bestimmten Unterklasse von `OclExpression` erzeugen. Dies wird im Folgenden anhand des Skriptes für die WFR für `BooleanLiteralExp` illustriert:

```
{
//context BooleanLiteralExp inv: type.oclIsKindOf(Primitive) and
//                               type.getNameA() = 'Boolean'
Classifier   context     =   "OCL::Expressions::BooleanLiteralExp";
OpCall       oce1        =   $context "allInstances";
IteratorVar  selfDecl    =   "self" $context;

VarExp       self        =   $selfDecl;
AssocEndCall aece        =   $self "type";
OpCall       getName     =   $aece "getNameA";
String       str         =   "Boolean";
OpCall       compare     =   $getName "=" $str;
VarExp       self2       =   $selfDecl;
AssocEndCall aece1       =   $self2 "type";
IsKindOf     kindOf      =   $aece1 "OCL::CommonModel::Primitive";
OpCall       and         =   $kindOf "and" $compare;

Iterator     forAll      =   $oce1 "forAll" $selfDecl $and;
Invariant    inv         =   $context "BooleanLiteralExpWFR" $forAll
     "self.type.oclIsKindOf(Primitive) and self.type.getNameA() = 'Boolean'";
}
```

Beispielsweise erzeugt die Anweisung

```
OpCall       compare     =   $getName "=" $str;
```

eine Instanz von `OperationCallExp`. Die aufgerufene Operation ist "=". Der Quellausdruck dieses Operationsaufrufes ist `getName`, eine andere Instanz von `OperationCallExp`. Das übergebene Argument ist `str`, eine Instanz von `StringLiteralExp`.

Die Typen der einzelnen Teilausdrücke müssen nicht explizit angegeben werden. Vielmehr werden sie durch den Parser dieser Skriptsprache mit Hilfe des in Kapitel 5.3.3 vorgestellten `TypeEvaluator` ermittelt.

Da ein Ziel der vorliegenden Arbeit war, für einige einfache WFRs des OCL-Metamodells Code zu generieren, wurden entsprechende Skripte noch für Invarianten der Klassen `CollectionLiteralExp` und `SequenceType` erstellt.

6 Einschränkungen der Implementation

Die Implementation des metamodellbasierten OCL-Compilers unterstützt bereits den größten Teil der in der Spezifikation angegebenen Sprachkonzepte. In diesem Kapitel soll aufgezeigt werden, welche Einschränkungen noch bestehen.

Diese Einschränkungen betreffen zum größten Teil den Code-Generator und die OCL-Basisbibliothek. Für die Implementierung einiger der im Folgenden genannten Sprachkonzepte ist allerdings ebenfalls noch eine Vervollständigung des OCL-Metamodells nötig.

6.1 Kontexte

Laut [OCL16,S.7-3ff] gibt es zahlreiche Platzierungsmöglichkeiten für OCL-Ausdrücke in Modellen. Neben der Angabe von Invarianten für Klassen und Vor- und Nachbedingungen für Operationen ist es auch möglich, mittels OCL die initialen Werte von Attributen, die Werte abgeleiteter Assoziationen und Attribute, den Methodenrumpf von Abfrageoperationen und in der dynamischen Modellierung Bedingungen für Zustandsübergänge zu spezifizieren.

Die Implementation unterstützt gegenwärtig nur Invarianten (siehe Kapitel 5.6.3, `JmiClassCodeGenerator`) und auch diese nur unzureichend.

Im Folgenden sollen die noch zu lösenden Probleme für die verschiedenen Kontexte genannt werden. In [Fin00] und [Wie00] wurden bereits Konzepte vorgestellt und umgesetzt, die einen Teil dieser Probleme betreffen. Es ist allerdings noch zu untersuchen, inwieweit diese Konzepte auch für die JMI-Implementierung der Basisbibliothek angewandt werden können.

Invarianten:

- Berücksichtigung der Vererbung von Invarianten an Unterklassen.
- Auswertungszeitpunkt für Invarianten. So existieren z.B. im NetBeans Repository Benachrichtigunsmechanismen für Modelländerungen. Invarianten könnten also bei Auftreten von Änderungen ausgewertet werden.

Vor- und Nachbedingungen:

- Beziehungen zwischen Parametern der Operation und den entsprechenden Variablen (`VariableDeclaration`) sind in [OCL16,S.7-5] unzureichend modelliert.
- Einbringen des generierten Java-Codes in die Implementation der Operation, z.B. durch Nutzung der Ergebnisse von [Wie00]. Dadurch werden Abhängigkeiten von der jeweiligen Repository-Implementation eingeführt, denn die Konventionen für die Implementation von Operationen in Metamodellen sind nicht standardisiert (s.a. Kapitel 2.3).
- Modellierung und Code-Generierung für `@pre`.

Initiale Werte für Attribute:

- Einbringen des generierten Codes in die Konstruktoren der entsprechenden Klasse.

Werte abgeleiteter Attribute und Assoziationen:

- Auswertungszeitpunkt. Evtl. Nutzung von Benachrichtigungsmechanismen des Repository, um die Werte im Falle entsprechender Änderungen im Modell zu aktualisieren.

Bedingungen für Zustandsübergänge:

- Nur für UML relevant und dort nur dann, falls aus Zustandsdiagrammen Code generiert wird.

6.2 OrderedSet

`OrderedSet` ist ein Kollektionstyp, der sowohl Eigenschaften von `Sequence`, als auch von `Set` besitzt. Die Verwendung von `OrderedSet` anstelle von `Sequence` ist z.B. bei der Navigation entlang geordneter Assoziationen sinnvoll, da hierbei keine Duplikate auftreten.

Problematisch ist, dass die Spezifikation einiger Operationen von `OrderedSet` in [OCL16,S.6-11f] falsch ist. So wird z.B. für `append()` nicht der Spezialfall betrachtet, dass ein bereits enthaltenes Objekt erneut zu einem `OrderedSet` hinzugefügt wird.

Während `OrderedSet` in Anlage B – Wellformedness Rules bereits konsequent verwendet wird, nutzt die Implementation stattdessen `Sequence`.

Soll `OrderedSet` in der Implementation realisiert werden, so ist zunächst `OrderedSetType` zum Metamodell hinzuzufügen. Des Weiteren ist `OrderedSet` in der OCL Standard Bibliothek und in der OCL-Basisbibliothek zu ergänzen und im `TypeEvaluator` sowie im Code-Generator entsprechend der Spezifikation zu verwenden.

Diese Vervollständigung der Implementation ist von geringem bis mittlerem Schwierigkeitsgrad, jedoch mit gewissem Aufwand verbunden, da alle Subsysteme betroffen sind.

6.3 Qualifizierte Assoziationen

Sowohl im OCL-Metamodell als auch im TypeEvaluator wird die Angabe von Qualifikationen (Qualifier) bei der Navigation unterstützt.

Im Code-Generator werden diese jedoch nicht berücksichtigt. Auch die geänderte OCL-Basisbibliothek bietet noch keine entsprechenden Operationen an, um entlang von Assoziationen unter Angabe von Qualifikationen zu navigieren.

Diese Funktionalität wurde zunächst vernachlässigt, da für `MOF-OCL` Qualifikationen nicht relevant sind. Für eine Code-Generierung für OCL-Ausdrücke in UML-Diagrammen ist es jedoch sinnvoll, die Implementation diesbezüglich zu ergänzen. Das heißt z.B., dass in OclModelObject eine `getFeature()`-Methode benötigt wird, die die Angabe von Qualifikationen erlaubt.

Die nötigen Änderungen sind lokal begrenzt und von geringem Schwierigkeitsgrad.

6.4 OclMessageExp

`OclMessageExp` wird gegenwärtig weder vom Code-Generator noch von der OCL-Basisbibliothek unterstützt.

Um die Auswertung solcher Ausdrücke zu ermöglichen, ist es nötig, über während der Ausführung einer Operation gesendete Nachrichten Buch zu führen.

Hierfür ist es u.a. erforderlich, die in [Wie00] vorgestellte Code-Instrumentierung so zu erweitern, dass während der Ausführung einer Methode aufgezeichnet werden kann, welche anderen Methoden unter Verwendung welcher Argumente aufgerufen werden.

Schwierigkeitsgrad und Aufwand für die Implementierung der Code-Generierung für `OclMessageExp` werden als hoch eingeschätzt.

6.5 Iteratoren mit mehreren Variablen

Der Code-Generator berücksichtigt für `IterateExp` und `IteratorExp` gegenwärtig nur eine Iteratorvariable.

Die nötigen Änderungen beschränken sich auf die Kollektionsklassen der OCL-Basisbibliothek und auf den Code-Generator. Der Schwierigkeitsgrad wird als gering eingeschätzt.

6.6 Die Operation `oclIsInState`

Die Operation `oclIsInState` wird in [OCL16,S.6-3] für `OclAny` definiert. Als Argument erwartet sie ein Literal der Enumeration `OclType`, welche alle Zustände des Modells aufzählt.

Weder `oclIsInState` noch `OclState` wird in der gegenwärtigen Implementation unterstützt.

Die Modellierung des Parametertyps mit Hilfe einer Enumeration ist analog zum Vorgehen bei `oclAsType`, `oclIsTypeOf` und `oclIsKindOf`. Wie in Kapitel 5.4.3 für diese Operationen erläutert, sollte daher analog auch `oclIsInState` durch eine eigene Unterklasse von `OclExpression` modelliert werden (`OclIsInStateExp`).

In der Implementation des Code-Generators und in der OCL-Basisbibliothek sind Ergänzungen vorzunehmen, um `OclIsInStateExp` zu unterstützen[46].

Der Aufwand und Schwierigkeitsgrad der nötigen Erweiterungen wird als gering eingeschätzt.

---

[46] Die JMI-Implementierung der Basisbibliothek wird jedoch `OclIsInStateExp` nicht unterstützen werden, da es in MOF keine Klasse `State` gibt.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im Rahmen der vorliegenden Arbeit wurde ein metamodellbasierter OCL-Compiler entworfen und implementiert. Grundlage des Compilers ist ein gemeinsames OCL-Metamodell, das auf einer Abstraktion von MOF und UML basiert. Somit ist es möglich, sowohl OCL-Ausdrücke in Metamodellen als auch in UML-Modellen als Instanz dieses OCL-Metamodells darzustellen.

Für dieses gemeinsame OCL-Metamodell wurden die Wellformedness Rules neu und vom UML-Metamodell entkoppelt formuliert. Dabei wurde auch auf Vervollständigung und Korrektur der in [OCL16] angegebenen WFRs Wert gelegt. Es wurde entsprechend den Wellformedness Rules für das gemeinsame OCL-Metamodell ein Typinferenz-Algorithmus implementiert, der den Typ eines OCL-Ausdrucks ermittelt. Dieser wurde in einem Parser für eine einfache Skriptsprache zur Erzeugung von OCL-Metamodellinstanzen genutzt.

Basierend auf dem gemeinsamen OCL-Metamodell wurde eine Transformation von OCL-Ausdrücken in Java-Code angegeben. Die Transformationsspezifikation selbst wurde in OCL formuliert. Sie stellt somit ein ausführliches Beispiel für die Anwendung von OCL im Rahmen der Model Driven Architecture dar.

Als ein Spezialfall dieser Transformation wurde die Code-Generierung für OCL-Ausdrücke in Metamodellen implementiert. Es war eine Restrukturierung der OCL-Basisbibliothek nötig, um diese auch für JMI-basierte Modellzugriffe nutzbar zu machen.

Als Fazit in Bezug auf die Aufgabenstellung ist zu formulieren, dass ein anfangs nicht erwarteter Aufwand nötig war, um das in [OCL16] angegebene OCL-Metamodell für UML **und** MOF anwenden zu können. Dennoch ist es neben Entwurf und Implementation dieses gemeinsamen OCL-Metamodells gelungen, auch für die Code-Generierung einen weit fortgeschrittenen Prototypen zu erstellen.

Trotzdem bleibt viel Raum für weiterführende Arbeiten zu diesem Thema. Auf solche soll im Folgenden abschließend eingegangen werden.

7.2 Zukünftige Arbeiten

Die Einschränkungen der vorgestellten Implementation wurden bereits in Kapitel 6 erläutert. Eine Behebung dieser Mängel wäre also eine erste zu lösende Aufgabe, wobei allerdings teilweise besser Klarstellungen und Vervollständigungen in kommenden Versionen der OCL-Spezifikation abgewartet werden sollten. Dies betrifft z.B. die Frage der verschiedenen Kontexte und ihrer Abbildung von konkreter in abstrakte Syntax

In Kapitel 5.5 wurde eine Implementation der OCL-Basisbibliothek beschrieben, die per JMI-Reflektion auf Modelle zugreift. Der ursprünglich implementierte Modellzugriff per Java-Reflektion wurde in der restrukturierten OCL-Basisbibliothek noch nicht umgesetzt. Dies sollte nachgeholt werden, um so auch wieder das in [Fin00] und [Wie00] beschriebene Überprüfen von OCL-Invarianten und Vor- und Nachbedingungen in Java-Programmen zur Laufzeit implementieren zu können.

7.2.1 Migration nach UML 2.0

Im Rahmen der weiteren Entwicklung des Vorschlages für OCL 2.0 wird eine Integration mit dem Metamodell von UML 2.0 erfolgen. Auch für den hier vorgestellten metamodellbasierten OCL-Compiler muss diese Migration zur nächsten UML-Version vollzogen werden.

Die in dieser Arbeit vollzogene Entkopplung von OCL-Metamodell und UML-Metamodell durch Definition zahlreicher neuer Operationen für den Zugriff auf Attribute und Assoziationen des UML-Metamodells (bzw. von MOF) könnte sich auch für die Migration nach UML 2.0 als hilfreich erweisen:

In [OCL16] müssen bei Änderung des UML-Metamodells nahezu alle WFRs und auch alle OCL-Ausdrücke im Kapitel 5, "Semantics Described in UML" geändert werden. Wird dagegen der Zugriff auf das UML-Metamodell wie in dieser Arbeit immer durch zusätzliche Operationen gekapselt, beschränken sich die Änderungen auf diese Operationen.

Bei einer Migration des Compilers nach UML 2.0 wird es nötig sein, auch das verwendete Repository gegen ein MOF 2.0 Repository auszutauschen, da das UML 2.0 Metamodell eine Instanz von MOF 2.0 sein wird.

Dies wirft die Frage auf, was mit der in Kapitel 5.1 beschriebenen Metamodellintegration im Repository und mit der in Kapitel 5.2.5 beschriebenen Generierung der

spezifischen OCL-Pakete geschieht. Beide Algorithmen sind auf Basis der JMI-Schnittstellen für MOF 1.4 implementiert.

Da für OCL 2.0 jedoch ein normatives Metamodell (in Form eines XMI-Dokuments) existieren wird, das bereits entsprechende Referenzen zum UML 2.0 Metamodell enthält, ist eine Metamodellintegration nicht mehr nötig. Vielmehr wird ein Algorithmus benötigt, der das normative OCL 2.0 Metamodell mit dem in dieser Arbeit vorgestellten `Common-OCL` durch Erstellen entsprechender Vererbungsbeziehungen verknüpft. Auf diese Weise ist es möglich, den auf Basis von `Common-OCL` implementierten Parser sowie den Code-Generator weiterzuverwenden. Die auf `Common-OCL`-Ebene definierten Operationen müssen für UML 2.0 neu implementiert werden, ebenso wie sie auch für MOF 1.4 und UML 1.5 unterschiedlich implementiert sind (s.a. Anlage C – Spezifikation der Operationen).

7.2.2 Werkzeugintegration

Der in [Fin00] entwickelte OCL-Compiler verfügt über eine Schnittstelle namens `ModelFacade`, über die alle Zugriffe auf das Modell erfolgen. Ein UML-CASE-Tool, in das der OCL-Compiler integriert werden soll, muss diese Schnittstelle implementieren.

Der hier vorgestellte metamodellbasierte OCL-Compiler geht hingegen davon aus, dass das Modell im Repository residiert. Der Zugriff auf das Modell erfolgt mittels der JMI-Schnittstellen, die für das Paket `CommonModel` generiert wurden.

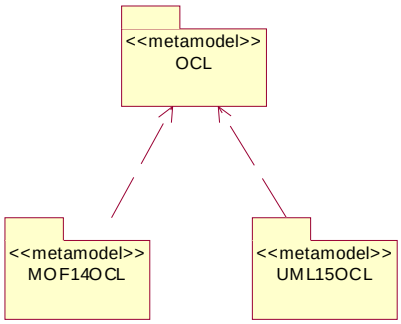
Das Problem der Integration des metamodellbasierten OCL-Compilers in UML-CASE-Tool ist – abgesehen von einem möglichen Datenaustausch per XMI – noch weitestgehend ungeklärt.

Eine mögliche Strategie wäre es, die UML 1.5 Implementation von `CommonModel` so zu erweitern, dass Operationen wie `findClassifier` oder `lookupAttribute` nicht mehr nur im Repository nach einem Modellelement suchen, sondern auch über eine Schnittstelle analog zu `ModelFacade` auf ein in einem UML-CASE-Tool erstelltes Modell zugreifen können. Für ein mit Hilfe der `ModelFacade` gefundenes Modellelement wird anschließend im Repository ein Stellvertreter erzeugt, der dann den Aufruf von weiteren in `CommonModel` definierten Operationen für dieses Modellelement ermöglicht.

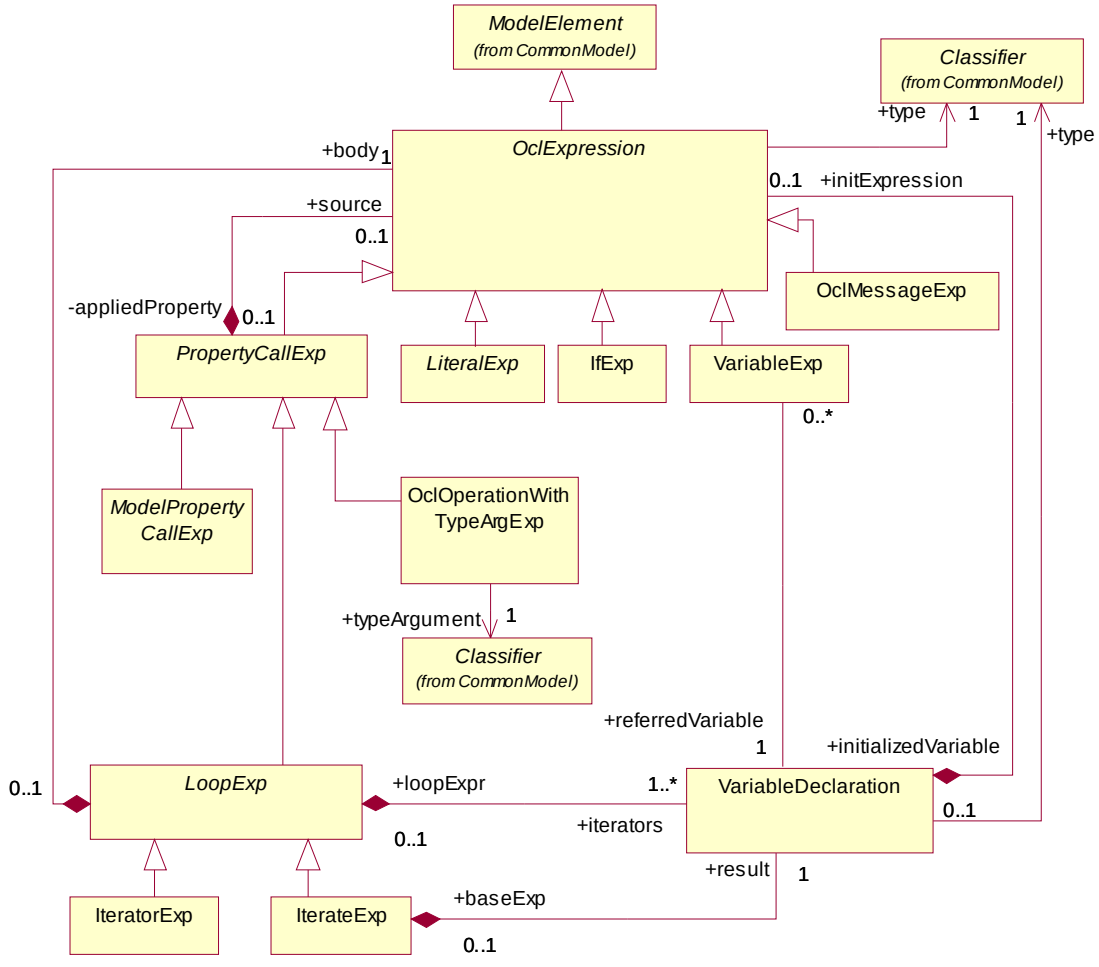
Anlage A – OCL-Metamodell

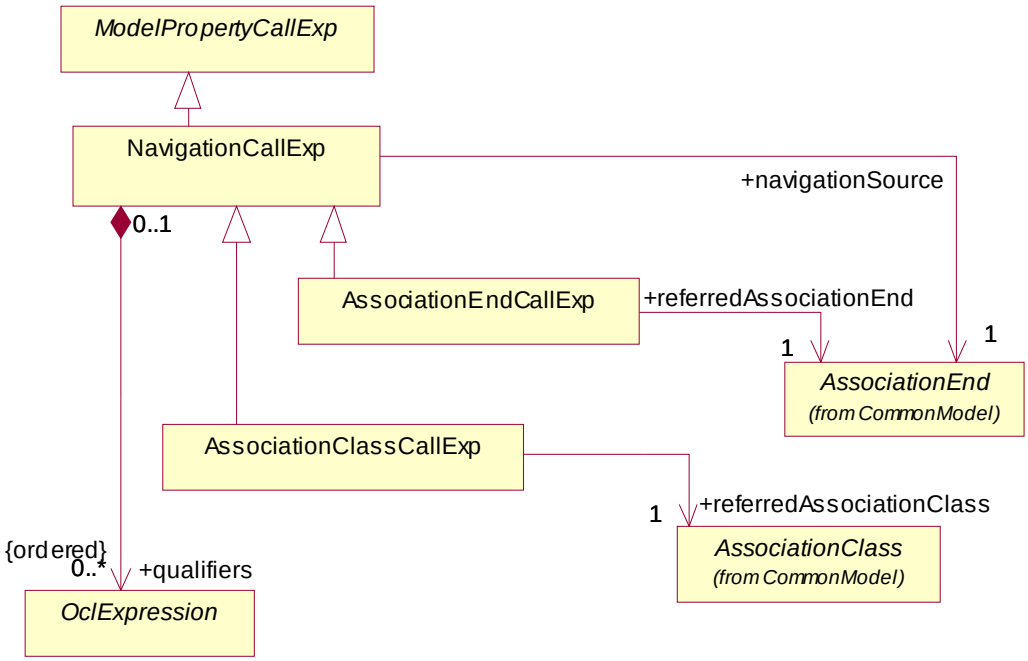
Übersicht

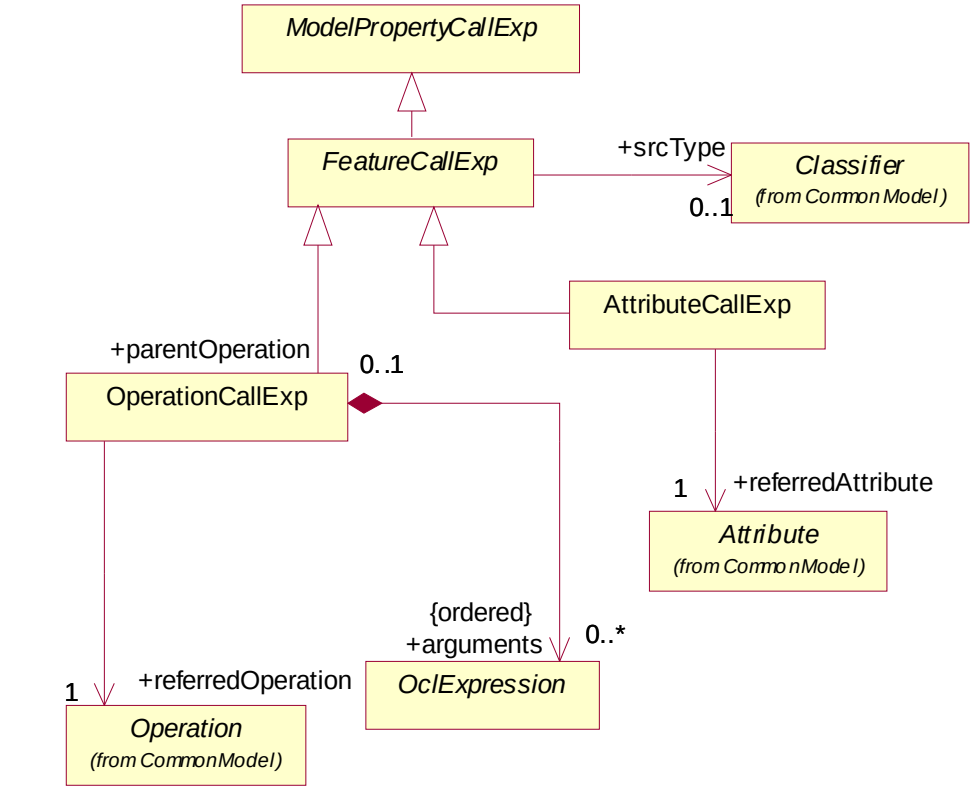
Gegenüber Kapitel 5 wurden folgende Umbenennungen durchgeführt: `Common-OCL` heißt kürzer: `OCL`. Die Namen spezifischen OCL-Pakete wurden um eine Versionsnummer ergänzt.

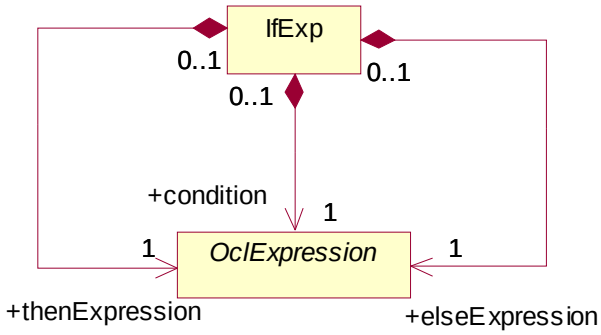


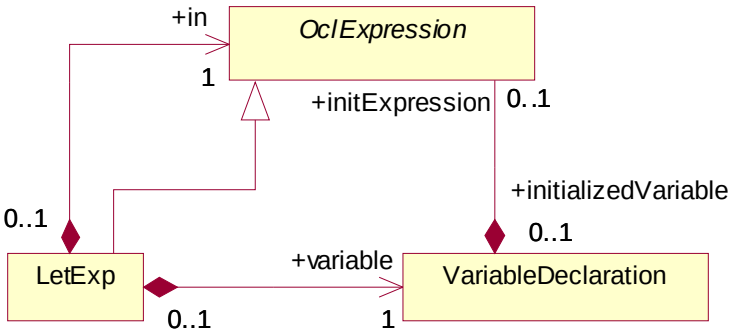
Paket OCL::Expressions

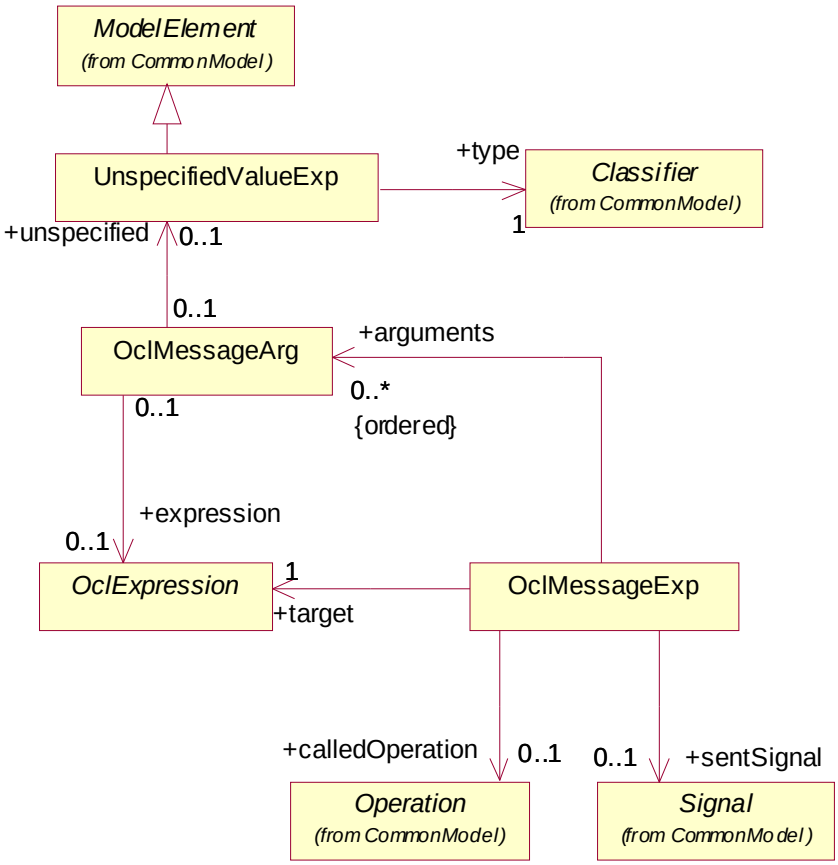


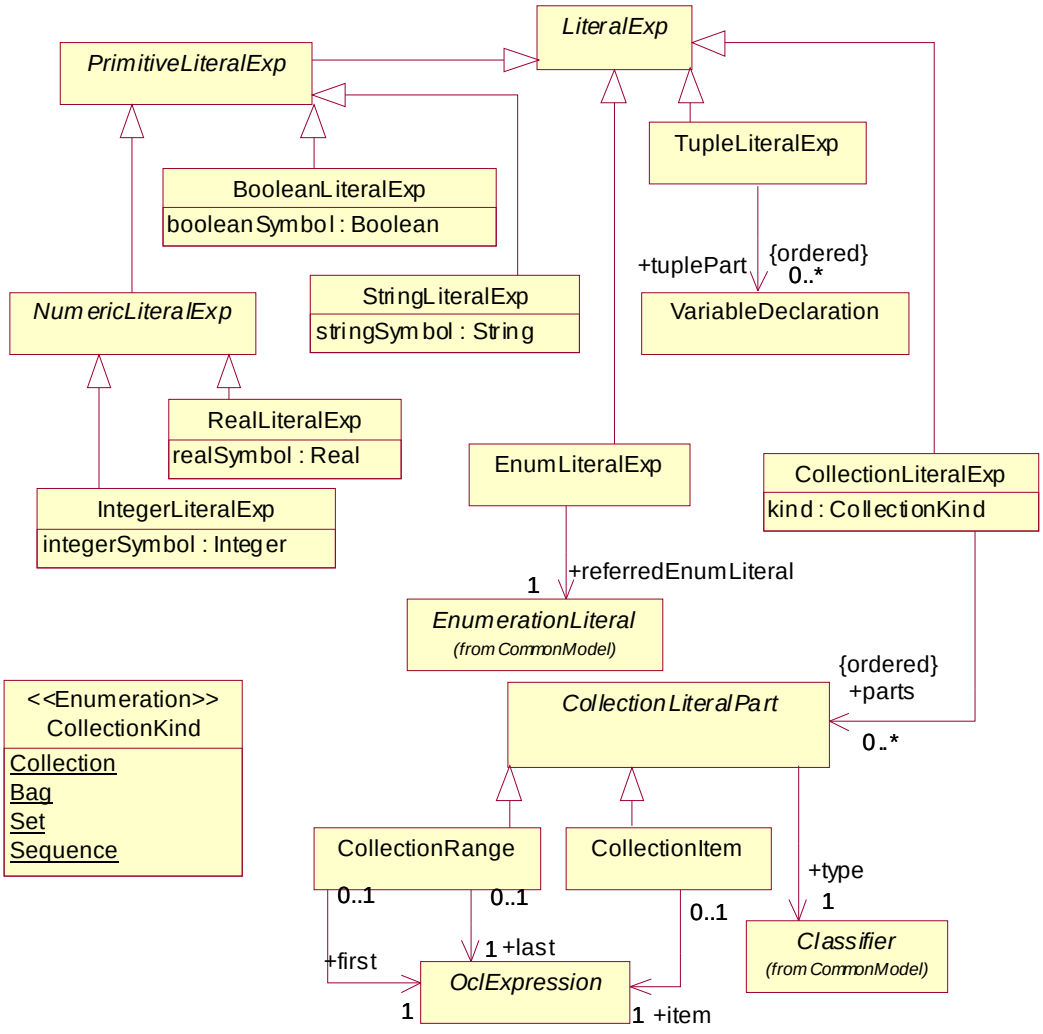

ModelPropertyCallExp
NavigationCallExp
+navigationSource
0..1
AssociationEndCallExp
+referredAssociationEnd
1
1
AssociationEnd
(from CommonModel)
AssociationClassCallExp
1
+referredAssociationClass
AssociationClass
(from CommonModel)
{ordered}
0..*
+qualifiers
OclExpression

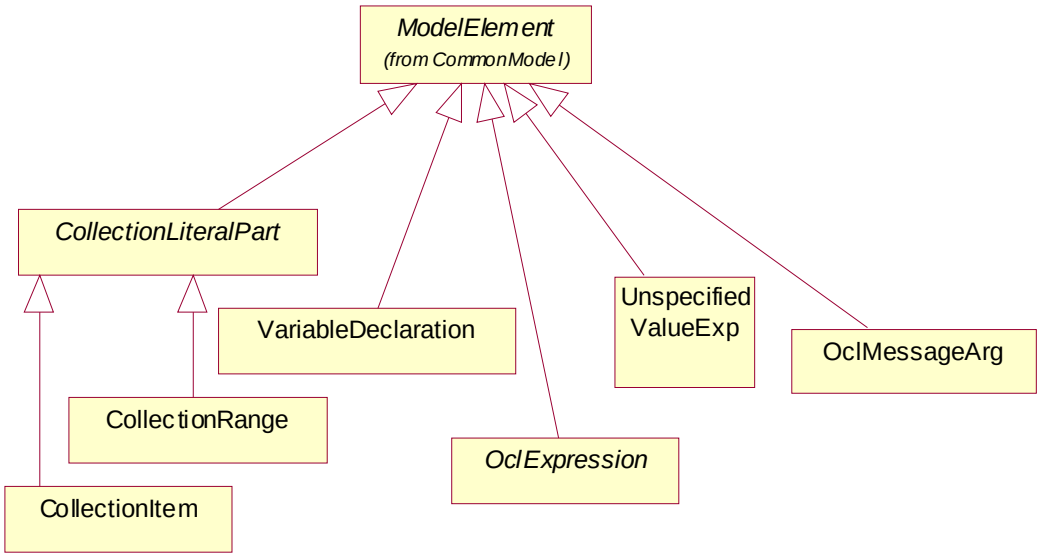

ModelPropertyCallExp
FeatureCallExp
+srcType
Classifier
(from Common Model)
0..1
AttributeCallExp
+parentOperation
0..1
OperationCallExp
1
+referredAttribute
Attribute
(from CommonModel)
{ordered}
+arguments
0..*
1
+referredOperation
OclExpression
Operation
(from CommonModel)

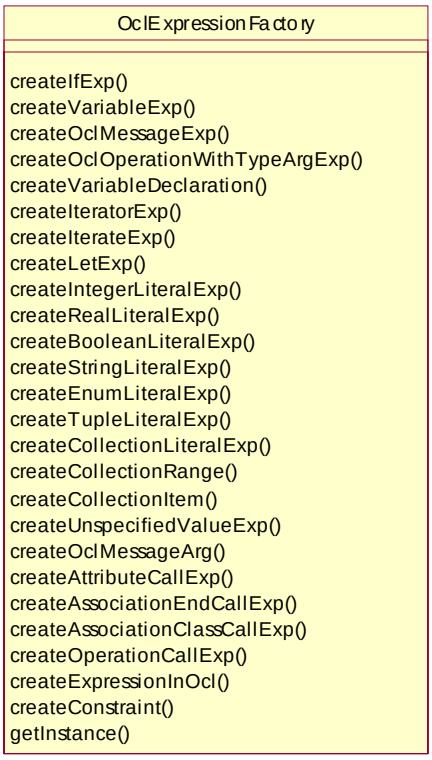

IfExp
0..1
0..1
0..1
+condition
1
1
OclExpression
1
+thenExpression
+elseExpression


+in
OclExpression
1
+initExpression
0..1
+initializedVariable
0..1
0..1
LetExp
+variable
VariableDeclaration
0..1
1

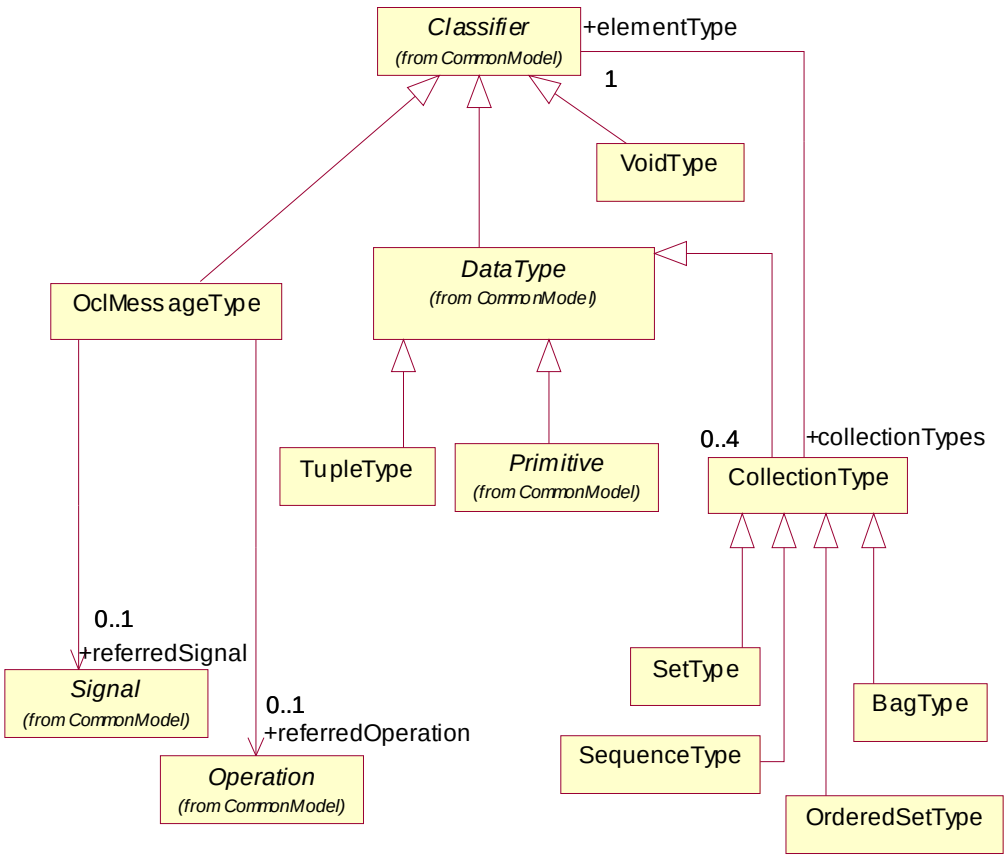

ModelElement
(from CommonModel)
UnspecifiedValueExp
+type
Classifier
(from CommonModel)
+unspecified
0..1
1
0..1
OclMessageArg
+arguments
0..1
0..*
{ordered}
+expression
0..1
OclExpression
1
+target
OclMessageExp
+calledOperation
0..1
0..1
+sentSignal
Operation
(from CommonModel)
Signal
(from CommonModel)


LiteralExp
PrimitiveLiteralExp
TupleLiteralExp
BooleanLiteralExp
booleanSymbol : Boolean
+tuplePart
{ordered}
0..*
VariableDeclaration
StringLiteralExp
stringSymbol : String
NumericLiteralExp
RealLiteralExp
realSymbol : Real
EnumLiteralExp
CollectionLiteralExp
kind : CollectionKind
IntegerLiteralExp
integerSymbol : Integer
1
+referredEnumLiteral
EnumerationLiteral
(from CommonModel)
{ordered}
+parts
<<Enumeration>>
CollectionKind
Collection
Bag
Set
Sequence
CollectionLiteralPart
0..*
CollectionRange
CollectionItem
+type
1
0..1
0..1
0..1
+first
1
1 +last
OclExpression
1
1 +item
Classifier
(from CommonModel)

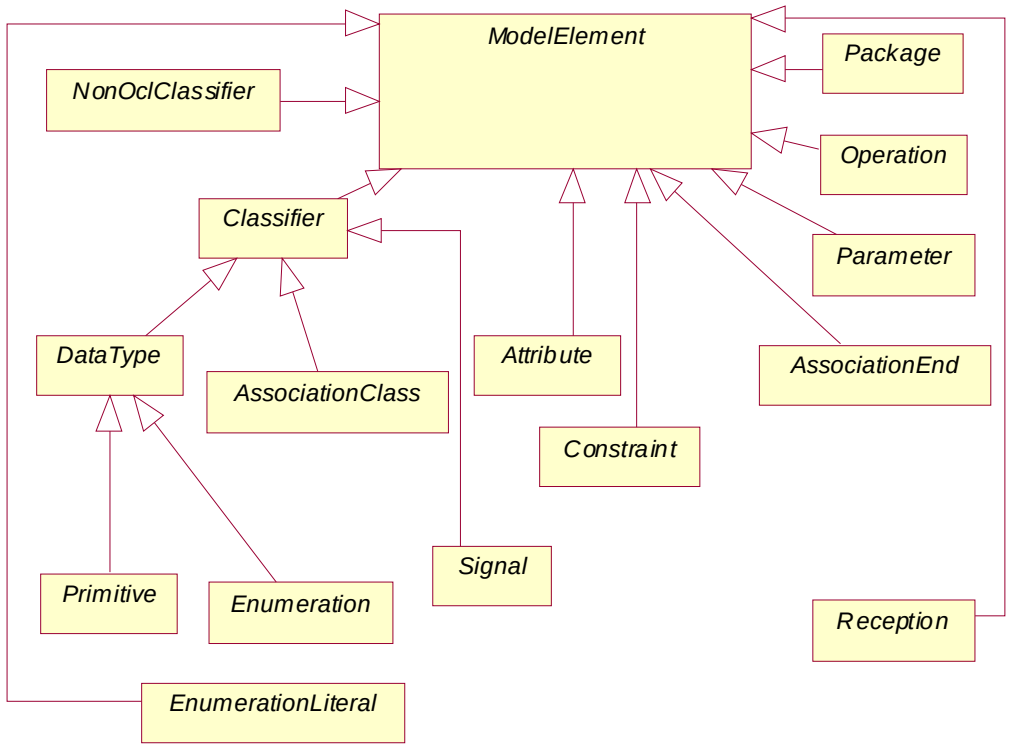

ModelElement
(from CommonModel)
CollectionLiteralPart
VariableDeclaration
Unspecified
ValueExp
OclMessageArg
CollectionRange
OclExpression
CollectionItem

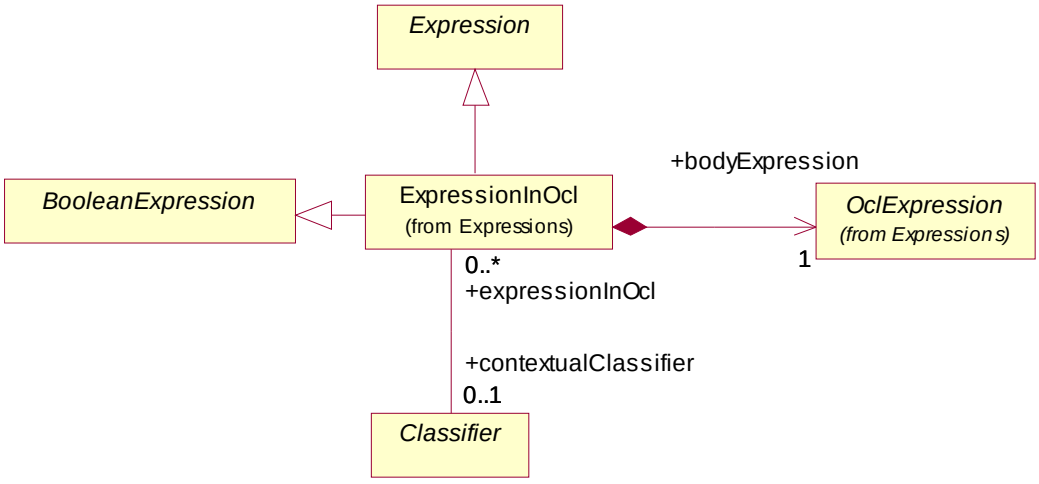


Paket OCL::Types

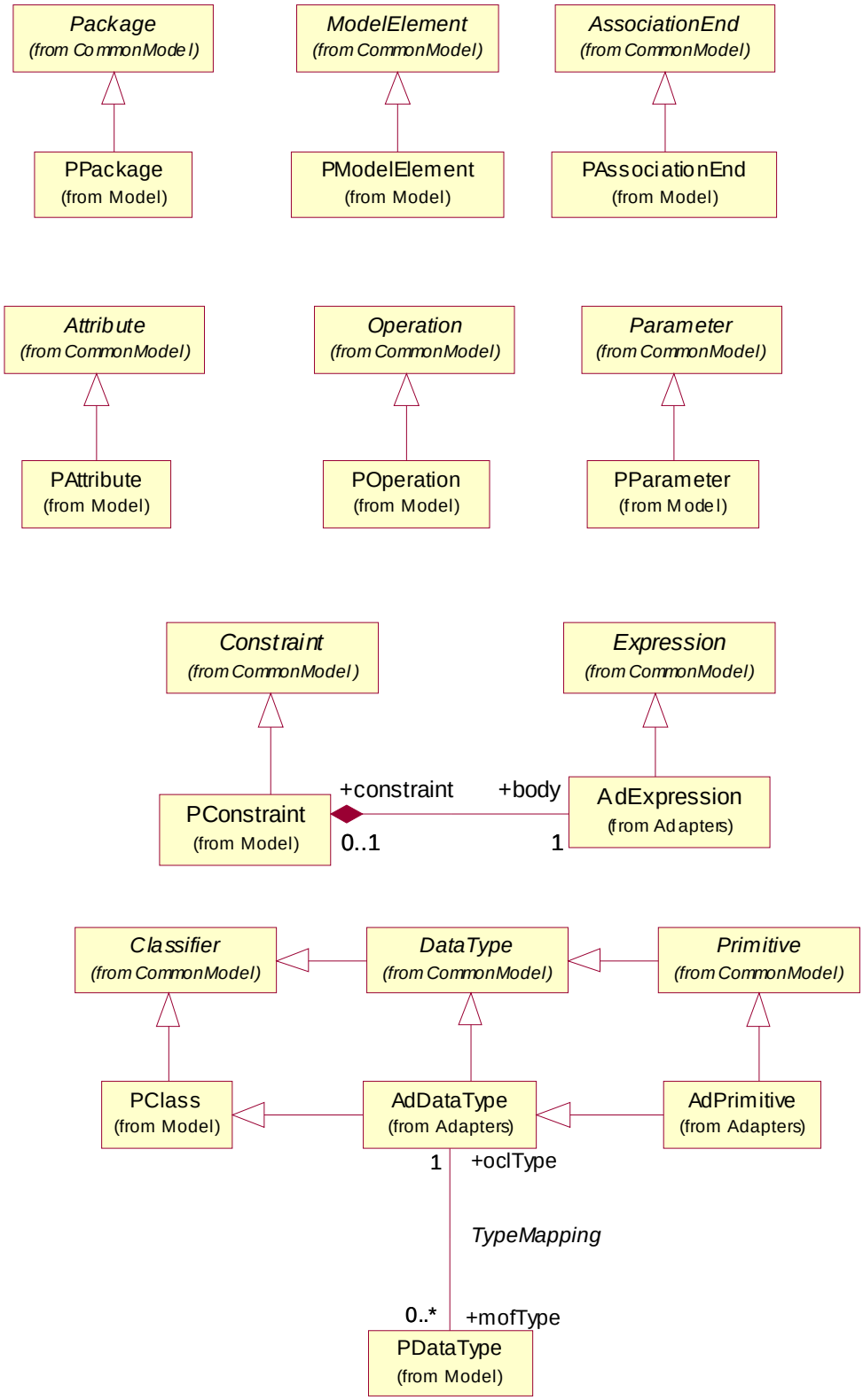


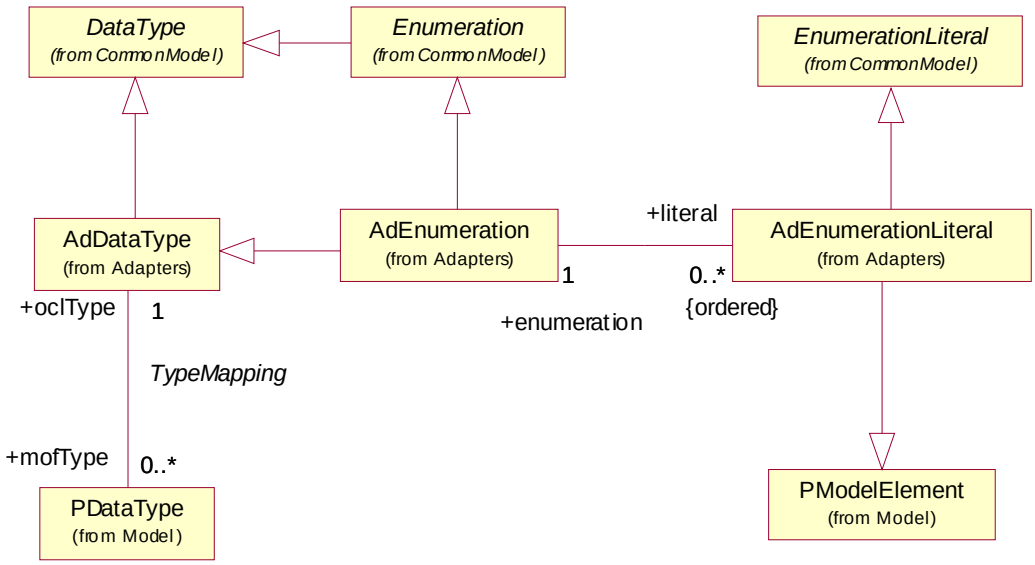
Paket OCL::CommonModel



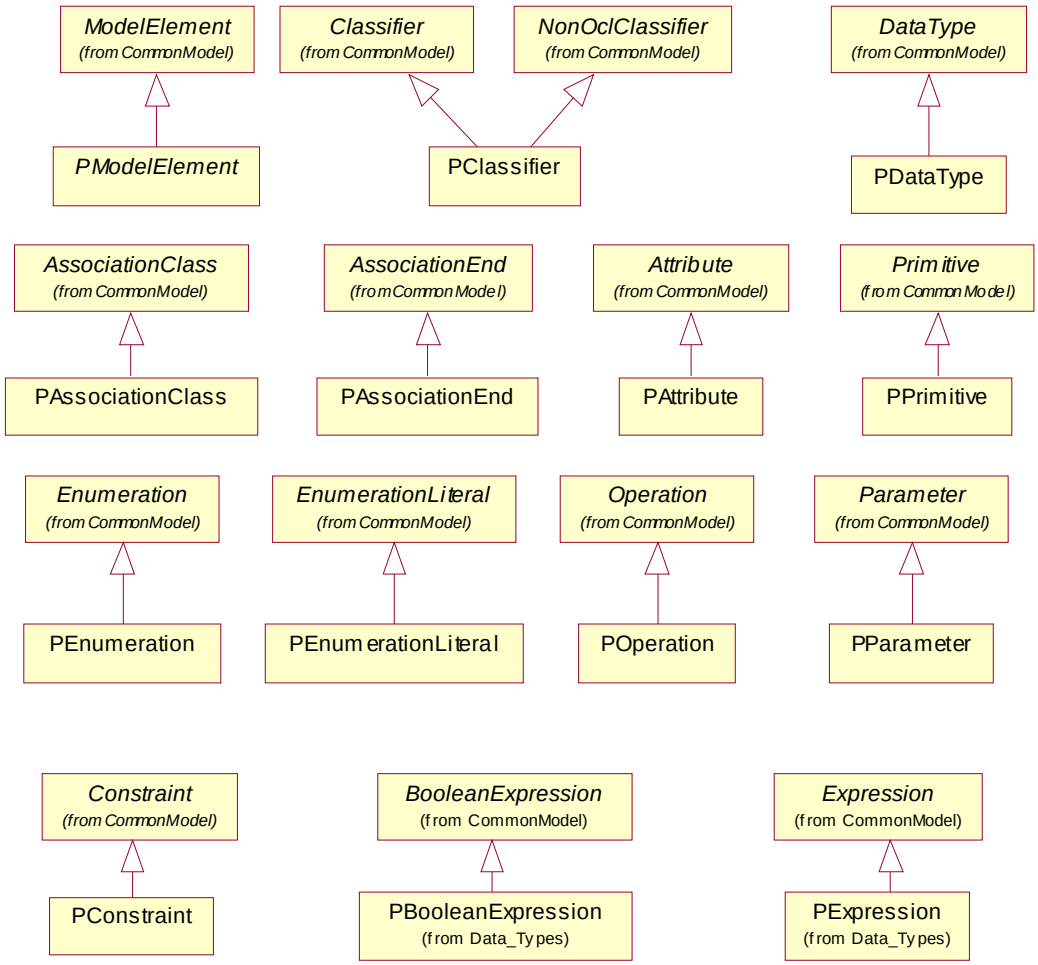


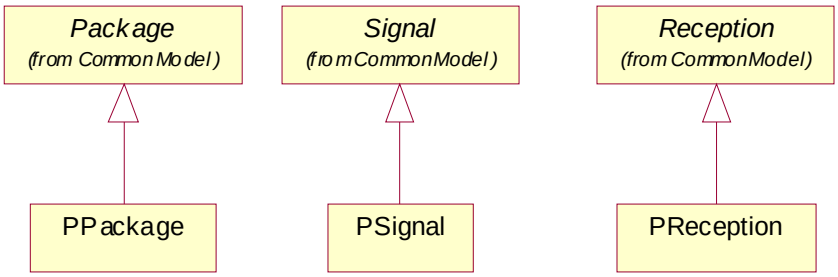
Pakete MOF14OCL::Adapters und MOF14OCL::Proxies::Model





Paket UML15OCL::Proxies




Package
(from CommonModel)
Signal
(from CommonModel)
Reception
(from CommonModel)
PPackage
PSignal
PReception

Anlage B – Wellformedness Rules

Im Folgenden werden die Wellformedness Rules für das gemeinsame OCL-Metamodell (Common-OCL) vorgestellt. Sie unterscheiden sich von den in den Kapitel 3.3.2 und 3.3.7 von [OCL16] angegebenen vor allem dadurch, dass sämtliche Navigationen entlang interner Assoziation des UML-Metamodells durch Operationsaufrufe ersetzt wurden. Die zusätzlich im Paket `OCL::CommonModel` benötigten Operationen werden in Anlage C spezifiziert. Neben diesen Änderungen wurde auch Wert darauf gelegt, offensichtlich Fehler oder Unvollständigkeiten zu beseitigen. Alle vorgenommenen Änderungen und Ergänzungen werden explizit beschrieben.

Zum einfachen Abgleich mit den originalen Wellformedness Rules wurden die Kommentare in englischer Sprache beibehalten.

Paket OCL::Types

Änderungen in allen betroffenen Constraints:

- `name` durch `getNameA()` ersetzt.

BagType

Änderungen:

- „+“ durch `concat()` ersetzt

```
[1] The name of a bag type is “Bag” followed by the element type’s name in parentheses.
context BagType
inv: self.getNameA()=’Bag(’.concat(self.elementType.getNameA()).concat(’)’)
```

CollectionType

Änderungen:

- „+“ durch `concat()` ersetzt

```
[1] The name of a collection type is “Collection” followed by the element type’s name in parentheses.
context CollectionType
inv: getNameA()=’Collection(’.concat(self.elementType.getNameA()).concat(’)’)
```

Classifier

Änderungen:

- `OrderedSetType` ergänzt.

```
[1] For each classifier at most one of each of the different collection types exist.
```

```
context Classifier
inv: collectionTypes->select(oclIsTypeOf(CollectionType))->size() <= 1
inv: collectionTypes->select(oclIsTypeOf(BagType))->size() <= 1
inv: collectionTypes->select(oclIsTypeOf(SequenceType))->size() <= 1
inv: collectionTypes->select(oclIsTypeOf(SetType))->size() <= 1
inv: collectionTypes->select(oclIsTypeOf(OrderedSetType))->size() <= 1
```

OclMessageType

[1] OclMessageType has either a link with a Signal or with an operation, but not both

```
context OclMessageType
inv: referredOperation->size() + referredSignal->size() = 1
```

Änderungen:

- `parameter` durch `getParametersA()` ersetzt.
- `feature` durch `attributesA()`ersetzt.

[2] The parameters of the referredOperation become attributes of the instance of OclMessageType

```
context OclMessageType
inv: referredOperation->size() = 1 implies
   attributesA()= referredOperation.getParametersA().asAttribute()
```

Änderungen:

- `feature` durch `attributesA()` ersetzt.
- Weder im UML-Metamodell, noch im MOF-Metamodell kann ein Attribut zu mehreren Classifiern gehören. Daher können ein `OclMessageType` und das Signal nicht die **selben** Attribute haben, wohl aber namens- und typgleiche. Das Constraint wurde entsprechend korrigiert.

[3] The attributes of the referredSignal become attributes of the instance of OclMessageType

```
context OclMessageType
inv: referredSignal->size() = 1 implies
     let attrSelf: Sequence(Attributes) = attributesA() in
     let attrOther : Sequence(Attributes) = referredSignal.attributesA()
     in
     attrSelf->size() = attrOther->size() and
     Sequence{1..attrSelf->size()}->forAll(n |
       attrSelf->at(n).getNameA() = attrOther->at(n).getNameA() and
       attrSelf->at(n).getTypeA() = attrOther->at(n).getTypeA())
```

Ergänzt, um für [2] und [3] weiterhin zu garantieren, dass der `OclMessageType` ausschließlich Attribute besitzt.

[4] The OclMessageType has no associations or operations and can not receive any signals

```
context OclMessageType
inv: self.allAssociationEndsA()->isEmpty()
     and self.allAssociationClassesA()->isEmpty()
     and self.allOperationsA()->isEmpty()
     and self.allReceptionsA()->isEmpty()
```

OrderedSetType

Ergänzt. Analog zu den entsprechenden Constraints für die anderen Kollektionstypen.

[1] The name of a ordered set type is “OrderedSet” followed by the element type’s name in parentheses.

```
context SequenceType
inv: getNameA()='OrderedSet('.concat(self.elementType.getNameA()).concat(')')
```

SequenceType

Änderungen:

- „+“ durch `concat()` ersetzt

[1] The name of a sequence type is “Sequence” followed by the element type’s name in parentheses.

```
context SequenceType
inv: getNameA() = 'Sequence('.concat(self.elementType.getNameA()).concat(')')
```

SetType

Änderungen:

- „+“ durch `concat()` ersetzt

[1] The name of a set type is “Set” followed by the element type’s name in parentheses.

```
context SetType
inv: self.getNameA() = 'Set('.concat(self.elementType.getNameA()).concat(')')
```

TupleType

Änderungen:

- `type` durch `getTypeA()` ersetzt.

[1] The name of a tuple type includes the names of the individual parts and the types of those parts.

```
context TupleType
inv: getNameA() = 'Tuple('.concat( Sequence{1..attributesA()()->size()}
       ->iterate (pn; s : String = '' |
         let p : Attribute = attributesA()()->at (pn) in
           s.concat(
             (if (pn>1) then ',' else '' endif)
             .concat(p.getNameA()).concat(':').concat(p.getTypeA().getNameA())
         ))).concat (')')
```

[2] All parts belonging to a tuple type have unique names.

```
context TupleType
inv: -- always true, because attributes must have unique names.
```

Änderungen:

- Anstelle von `feature` mit Hilfe der `all...()` Operationen zum Ausdruck gebracht.

[3] A TupleType instance has only features that are Attributes (tuple parts).

```
context TupleType
inv: self.allAttributesA()->notEmpty()
     and self.allAssociationEndsA()->isEmpty()
     and self.allAssociationClassesA()->isEmpty()
     and self.allOperationsA()->isEmpty()
     and self.allReceptionsA()->isEmpty()
```

Paket OCL::Expressions

Änderungen in allen betroffenen Constraints:

- `name` durch `getNameA()` ersetzt.
- Ermittlung von Kollektionstypen zu einem gegebenen Classifier durch Nutzung von Operationen wie `getSetType()` anstelle von länglichen Ausdrücken der Form `collectionTypes->select(oclIsTypeOf(SetType))->first()`.

AssociationClassCallExp

- Ergänzt. Fehlte in der OCL-Spezifikation.

[1] The type of the AssociationClass call expression is the referred AssociationClass or a collection type of it. The source AssociationEnd of this navigation must always be unambiguous. In case of binary associtions, the kind of the resulting collection type is determined by the properties of the destination AssocitionEnd (which is just the opposite one of the source end). In case of n-ary (n>2) associations the resulting collection type is always a set type.

```
context AssociationClassCallExp inv:
   let ac : AssociationClass = self.referredAssociationClass in
   let srcType : Classifier = source.type in
   let ends : Sequence(AssociationEnd) = ac.getEndsA() in
   let srcEnds : Sequence(AssociationEnd)=
            ends->select(ae | srcType.conformsTo(ae.getTypeA())) in
    -- srcEnd has to be stated in case of navigation disambiguities
    -- occuring with reflexive associations
   let srcEnd : AssociationEnd = if srcEnds->size() = 1 then srcEnds->first()
                                 else self.navigationSource endif in
```

```
   (not srcEnd.oclIsUndefined()) and ends ->includes(srcEnd) and
      -- the AssociationClass must be reachable from the type of the source --
.......-- expression
   srcType.allAssociationClassesA()->includes(ac) and
   if ends->size() = 2 then
      --desination end
      let ae : AssociationEnd = ends->select (ae | ae <> srcEnd)->first() in
      let aeQual : Sequence(Attribute) = ae.getQualifiersA() in
      --potentially attached qualifiers have to match
      --with AssociationEnd qualifiers
      (qualifiers->isEmpty() or
      (qualifiers->size() = aeQual->size() and
       qualifiers->forAll(qual : OclExpression |
          qual.type.conformsTo(aeQual->at(qualifiers->indexOf(qual)).type))))
      and
      type = if aeQual->isEmpty() then
         if not ae.isMultiple() then ac    -- 0..1
         else    -- 0..*
            if ae.isOrderedA() then ac.getOrderedSetType()
            else ac.getSetType()
            endif
         endif
      else
       -- AssociationEnd with qualifiers
         if qualifiers->isEmpty() then
          -- no qualifier values are stated in the
          -- AssociationEndCallExpression, that is:
          -- take the whole set of associated objects
          -- (independent of stated multiplicity)
            if ae.isOrderedA() then ac.getOrderedSetType()
            else ac.getSetType()
            endif
         else
          -- qualified navigation => consider multiplicity
           if not ae.isMultiple() then ac    -- 0..1
           else    -- 0..*
              if ae.isOrderedA() then ac.getOrderedSetType()
              else ac.getSetType()
              endif
           endif
         endif
      endif
   else
     -- n-ary associations
```

```
    -- do not take any qualifiers or ordering into account
    -- (which target end should be considered therefore?)
      self.qualifiers->isEmpty() and
      type = ac.getSetType()
   endif
```

AssociationEndCallExp

- Ergänzt. Fehlte in der OCL-Spezifikation.

[1] The type of the AssociationEnd call expression is determined by the type of the AssociationEnd, by the multiplicity and by the qualifiers.

```
context AssociationEndCallExp inv:
   let ae : AssociationEnd = self.referredAssociationEnd in
   let aeQual : Sequence(Attribute) = ae.getQualifiersA() in
   let aeType : Classifier = ae.getTypeA() in
   let srcType : Classifier = source.type in
    -- the AssociationEnd must be reachable from the
    -- type of the source expression
   srcType.allAssociationEndsA()->includes(ae) and
   if srcType.isOclKindOf(AssociationClass) and
      srcType.oclAsType(AssociationClass).getEndsA()->includes(ae)
   then
    --navigation from an AssociationClass to a participiant of the Association
     qualifiers->isEmpty() and
     type = aeType
   else
   if ae.isBinaryA() then
    --potentially attached qualifiers have to match
    --with AssociationEnd qualifiers
      (qualifiers->isEmpty() or
      (qualifiers->size() = aeQual->size() and
       qualifiers->forAll(qual : OclExpression |
          qual.type.conformsTo(aeQual->at(qualifiers->indexOf(qual)).type))))
      and
      type = if aeQual->isEmpty() then
         -- AssociationEnd has no qualifiers
         if not ae.isMultiple() then aeType    -- 0..1
         else    -- 0..*
            if ae.isOrderedA() then aeType.getOrderedSetType()
            else aeType.getSetType()
            endif
         endif
      else
```

```
      -- AssociationEnd with qualifiers
        if qualifiers->isEmpty() then
         -- no qualifier values are stated in the
         -- AssociationEndCallExpression, that is:
         -- take the whole set of associated objects
         -- (independent of stated multiplicity)
           if ae.isOrderedA() then aeType.getOrderedSetType()
           else aeType.getSetType()
           endif
        else
         -- qualified navigation => consider multiplicity
          if not ae.isMultiple() then aeType    -- 0..1
          else    -- 0..*
             if ae.isOrderedA() then aeType.getOrderedSetType()
             else aeType.getSetType()
             endif
          endif
        endif
     endif
  else
  -- n-ary associations (n>2)
  -- they may not have any qualifiers, so no qualifiers are allowed
  -- in the navigation expression as well
   qualifiers->isEmpty() and
   --according to the semantics chapter
   --we get always Sets, even with multiplicity 0..1
   type = if ae.isOrderedA() then aeType.getOrderedSetType()
          else aeType.getSetType()
          endif
  endif
  endif
```

AttributeCallExp

Änderungen:

- Berücksichtigung der Multiplizität.
- Für mehrwertige Attribute Berücksichtigung von Ordnung und Einzigartigkeit.
- Für Klassenattribute wird explizite Angabe eines Quelltyps gefordert.

[1] The type of the Attribute call expression is the type of the referred attribute or – for an attribute with multiplicity greater than 1 – a collection type with the attribute type as element type. The kind of the collection is determined by the ordering and uniqueness of the attribute.

```
context AttributeCallExp
```

```
inv: let attr : Attribute = self.referredAttribute in
     let attrType : Classifier = attr.getTypeA() in
      -- classifier level attributes may have no source expression
      -- in this case, self.srcType has to be stated explicitly
     let sourceType: Classifier =
        if source->notEmpty() or attr.isInstanceLevelA()
        then source.type else self.srcType endif in
     (not sourceType.oclIsUndefined()) and
     sourceType.allAttributesA()->includes(attr) and
     type = if attr.isMultipleA() then
               if attr.isUniqueA() then
                  if attr.isOrderedA() then attrType.getOrderedSetType()
                  else attrType.getSetType()
                  endif
               else
                  if attr.isOrderedA() then attrType.getSequenceType()
                  else attrType.getBagType()
                  endif
               endif
            else
               attrType
            endif
```

BooleanLiteralExp

Änderungen:

- Test auf primitiven Typ ergänzt.

```
[1] The type of a Boolean literal expression is the type Boolean.
context BooleanLiteralExp
inv: self.type.oclIsKindOf(Primitive) and self.type.getNameA() = 'Boolean'
```

CollectionLiteralExp

```
[1] 'Collection' is an abstract class on the M1 level and has no M0 instances.
context CollectionLiteralExp
inv: kind <> CollectionKind::Collection
```

Änderungen:

- `OrderedSetType` ergänzt.
- `c : Classifier = OclVoid` geändert in
  `c : Classifier = VoidType.allInstances()->any(true)`

(zum Vergleich: ersteres ist genauso syntaktisch und semantisch falsch wie z.B. ein Ausdruck der Art `p:Person = Ingolf` in einem UML-Modell mit der Klasse `Person`)

[2] The type of a collection literal expression is determined by the collection kind selection and the common supertype of all elements. Note that the definition below implicitly states that empty collections have *OclVoid* as their *elementType*.

```
context CollectionLiteralExp
inv: kind=CollectionKind::Set implies type.oclIsKindOf(SetType)
inv: kind=CollectionKind::Sequence implies type.oclIsKindOf(SequenceType)
inv: kind=CollectionKind::Bag implies type.oclIsKindOf(BagType)
inv: kind=CollectionKind::OrderedSet implies type.oclIsKindOf(OrderedSetType)
inv: type.oclAsType(CollectionType).elementType =
   parts->iterate (p; c : Classifier = VoidType.allInstances()->any(true)
                   | c.commonSuperType (p.type))
```

CollectionItem

[1] The type of a CollectionItem is the type of the item expression.

```
context CollectionItem
inv: type = item.type
```

CollectionRange

Änderungen:

- Grundsätzlich nur `Integer` zulässig. [OCL16,S.3-16] [OCL16,S.5-21]

[1] The type of a CollectionRange is Integer.

```
context CollectionRange
inv: first.type.oclIsKindOf(Primitive) and
     first.type.name = 'Integer' and
     last.type.oclIsKindOf(Primitive) and
     last.type.name = 'Integer' and
     type.oclIsKindOf(Primitive) and
     type.name = 'Integer'
```

EnumLiteralExp

Änderungen:

- `enumeration` durch `getEnumerationA()` ersetzt.

[1] The type of an enum Literal expression is the type of the referred literal.

```
context EnumLiteralExp
inv: self.type = referredEnumLiteral.getEnumerationA()
```

IfExp

[1] The type of the condition of an if expression must be Boolean.

```
context IfExp
inv: self.condition.type.oclIsKindOf(Primitive) and
     self.condition.type.name = 'Boolean'
```

[2] The type of the if expression is the most common supertype of the else and then expressions.

```
context IfExp
inv: self.type = thenExpression.type.commonSuperType(elseExpression.type)
```

IntegerLiteralExp

Änderungen:

- Test auf primitiven Typ ergänzt.

[1] The type of an integer Literal expression is the type Integer.

```
context IntegerLiteralExp
inv: self.type.oclIsKindOf(Primitive) and self.type.getNameA() = 'Integer'
```

IteratorExp

Änderungen:

- „one“ ergänzt.

[1] If the iterator is 'forAll', 'isUnique', 'exists' or 'one' the type of the iterator must be Boolean.

```
context IteratorExp
inv: let name : String = getNameA() in
   name = 'exists' or name = 'forAll' or name = 'isUnique' or name = 'one'
   implies type.oclIsKindOf(Primitive) and type.getNameA() = 'Boolean'
```

Änderungen:

- Separierung in zwei Constraints: [2a] für „collectNested“, [2b] für „collect“.

[2a] The result type of the collectNested operation on a sequence or ordered set type is a sequence, the result type of 'collectNested' on any other collection type is a Bag. The type of the body is always the type of the elements in the return collection.

```
context IteratorExp
inv: let name : String =getNameA() in
   name = 'collectNested' implies
   if source.type.oclIsKindOf(SequenceType) or
      source.type.oclIsKindOf(OrderedSetType)
   then type = body.type.getSequenceType()
   else type = body.type.getBagType()
   endif
```

[2b] The result type of the collect operation on a sequence or ordered set type is a sequence, the result type of 'collect' on any other collection type is a Bag. If the type of the body is collection type, the type of

the elements in the return collection is the element type of this collection type. Otherwise the type of the elements in the return collection is the type of the body.

```
context IteratorExp
inv:
   let name : String =getNameA() in
   let flatType : Classifier =
      if body.type.oclIsKindOf(CollectionType) then
         body.type.oclAsType(CollectionType).getElementType()
      else
         body.type
      endif
   in
   name = 'collect' implies
   if source.type.oclIsKindOf(SequenceType) or
      source.type.oclIsKindOf(OrderedSetType)
   then type = flatType.getSequenceType()
   else type = flatType.getBagType()
   endif
```

[3] The 'select'and 'reject' iterators have the same type as its source.

```
context IteratorExp
inv:
   let name : String = getNameA() in
   name = 'select' or name = 'reject' implies type = source.type
```

Änderungen:

- „any“ ergänzt.

[4] The type of the body of 'select', 'reject' , 'exists', 'forAll' and 'any' must be boolean.

```
context IteratorExp
inv:
   let name : String =getNameA() in
   name = 'exists' or name = 'forAll' or name = 'select'
   or name = 'reject' or name = 'any'
   implies
   body.type.oclIsKindOf(Primitive) and body.type.getNameA() = 'Boolean'
```

Ergänzt.

[5] The type of the body of 'sortedBy' must have the operation < defined, which must return a Boolean value. The result type of 'sortedBy' is an ordered set type or a sequence type with the same element type as in the source collection type.

```
context IteratorExp
inv:
   let name : String = getNameA() in
```

```
   let elementType: Classifier
      = source.type.oclAsType(CollectionType).elementType in
   name = 'sortedBy' implies
   let cmpOp : Operation = body.type.lookupOperation('<', Sequence{body.type})
   in (not cmpOp.oclIsUndefined()) and
   cmpOp.getReturnParameterA().getTypeA().oclIsKindOf(Primitive) and
   cmpOp.getReturnParameterA().getTypeA().getNameA() = 'Boolean'
   if source.type.oclIsKindOf(SetType) or
      source.type.oclIsKindOf(OrderedSetType)
   then type = elementType.getOrderedSetType()
   else type = elementType.getSequenceType()
   endif
```

IterateExp

[1] The type of the iterate is the type of the result variable.

```
context IterateExp
inv: type = result.type
```

[2] The type of the body expression must conform to the declared type of the result variable.

```
context IterateExp
body.type.conformsTo(result.type)
```

[3] A result variable must have an init expression.

```
context IterateExp
inv: self.result.initExpression->size() = 1
```

LetExp

[1] The type of a Let expression is the type of the in expression.

```
context LetExp
inv: type = in.type
```

LoopExp

[1] The type of the source expression must be a collection.

```
context LoopExp
inv: source.type.oclIsKindOf(CollectionType)
```

[2] The loop variable of an iterator expression has no init expression.

```
context LoopExp
inv: self.iterators->forAll(initExpression->isEmpty())
```

[3] The type of each iterator variable must be the type of the elements of the source collection.

```
context IteratorExp
inv: iterators->forAll(type=source.type.oclAsType(CollectionType).elementType)
```

OclMessageArg

[1] There is either an expression or an unspecified value.

```
context OclMessageArg
inv: expression->size() + unspecified->size() = 1
```

OclMessageExp

Änderungen:

- Wiederverwendung der Operation `hasMatchingSignature()`.
- Beschreibung für den Typ der `OclMessageExp` ergänzt.

[1] If the message is a called operation, the arguments must conform to the parameters of the operation.

```
context OclMessageExp
inv: let argTypes : Sequence(Classifier) =
        arguments->collect(a | if a.expression->notEmpty() then
                        a.expression.type
                        else unspecified.type endif) in
     calledOperation->notEmpty() implies
     (calledOperation.hasMatchingSignature(argTypes) and
     type.oclIsTypeOf(OclMessageType) and
     type.oclAsType(OclMessageType).referredOperation = calledOperation)
```

Änderungen:

- Wiederverwendung der Operation `hasMatchingSignature()`.
- Beschreibung für den Typ der `OclMessageExp` ergänzt.

[2] If the message is a sent signal, the arguments must conform to the attributes of the signal.

```
context OclMessageExp
inv: let argTypes : Sequence(Classifier) =
        arguments->collect(a | if a.expression->notEmpty() then
                        a.expression.type
                        else unspecified.type endif) in
     sentSignal->notEmpty() implies
     (sentSignal.hasMatchingSignature(argTypes) and
     type.oclIsTypeOf(OclMessageType) and
     type.oclAsType(OclMessageType).referredSignal = sentSignal)
```

[3] If the message is a call action, the operation must be an operation of the type of the target expression.

```
context OclMessageExp
inv: calledOperation->notEmpty() implies
```

```
     target.type.allOperationsA()->includes(calledOperation)
```

[4] An OCL message has either a called operation or a sent signal.

```
context OclMessageExp
inv: calledOperation->size() + sentMessage->size() = 1
```

[5] The target of an OCL message cannot be a collection.

```
context OclMessageExp
inv: not target.type.oclIsKindOf(CollectionType)
```

OclOperationWithTypeArgExp

Diese Metaklasse wurde dem OCL-Metamodell hinzugefügt, um die Operationen „oclAsType()“, „oclsIsTypeOf()“, „oclIsKindOf“ modellieren zu können. Die folgenden Constraints sind daher in der OCL-Spezifikation nicht vorhanden.

[1] For 'oclAsType' the given typeArgument must conform to the type of the source expression (or vice versa). The type of this expression is the type referred by typeArgument.

```
context OclOperationWithTypeArgExp
inv: getNameA() = 'oclAsType' implies
     (typeArgument.conformsTo(source.type) or
      source.type.conformsTo(typeArgument)) and
      type = typeArgument
```

[2] For 'oclIsTypeOf' and 'oclIsKindOf' the type of this expression is Boolean.

```
context OclOperationWithTypeArgExp
inv: getNameA() = 'oclIsTypeOf'or getNameA() = 'oclIsKindOf' implies
     type.oclIsKindOf(Primitive) and type.getNameA() = 'Boolean'
```

OperationCallExp

Änderungen:

- Wiederverwendung der Operation `hasMatchingSignature()`. Constraints [2] und [3] werden somit überflüssig.

[1]All the arguments must conform to the parameters of the referred operation

```
context OperationCallExp
inv: let argTypes : Sequence(Classifier) =
           arguments->collect(a | a.type) in
     referredOperation.hasMatchingSignature(argTypes)
```

[2],[3] omitted

Ergänzt. Beschreibt den Typ der `OperationCallExp`.

[4] The referred operation must be defined on the source type. The type of the expression is the return type of the operation or a tuple type comprising the return type and the types of all out and inout parameters. The operation 'allInstances' can not be applied to primitive types, tuple types, collection types or message types. In case of 'allInstances ' the type of the expression is the set type of the source type. For the operation 'product', the type of the expression is the set type of a tuple type. This tuple type has to fields named 'first' and 'second' which have as types the element types of the source collection type and the one and only argument type, which is a collection type as well.

```
context OperationCallExp
inv: let name : String = getNameA() in
     let op : Operation = referredOperation in
       -- outparams: all out, inout parameters and the return parameter
     let outparams : Sequence(Parameter) = op.getOutAndInOutParametersA() in
     let sourceType: Classifier =
        if source->notEmpty() or op.isInstanceLevelA()
        then source.type else self.srcType endif in
     (not sourceType.oclIsUndefined()) and
     sourceType.allOperationsA()->includes(op) and
     if(name = 'allInstances') then
        not sourceType.oclIsKindOf(Primitive) and
        not sourceType.oclIsKindOf(CollectionType) and
        not sourceType.oclIsKindOf(TupleType) and
        not sourceType.oclIsKindOf(OclMessageType) and
        self.type = sourceType.getSetType()
     else
       if(name = 'product' and sourceType.oclIsKindOf(CollectionType)) then
          let tpt1 : Classifier=
             sourceType.oclAsType(CollectionType).elementType in
          let tpt2 : Classifier =
             arguments->first().oclAsType(CollectionType).elementType in
          self.type.oclIsTypeOf(SetType) and
          self.type.oclAsType(SetType).elementType.oclIsTypeOf(TupleType) and
          let tt : TupleType =
             self.type.oclAsType(SetType).elementType.oclAsType(TupleType) in
          let attr : Sequence(Attribute) = tt.attributesA()() in
          attr->size() = 2 and
          attr->first().getNameA()= 'first' and
          attr->first().getTypeA() = tpt1 and
          attr->last().getNameA()= 'second' and
          attr->last().getTypeA() = tpt2
       else
        -- „normal“ operations
          outparams->notEmpty() and
          if outparams->size() = 1 then
             self.type = outparams->first().getTypeA()
          else
```

```
            self.type.oclIsTypeOf(TupleType) and
            self.type.oclAsType(TupleType).attributesA()() =
               outparams->collect(p | p.asAttribute())
         endif
       endif
     endif
```

RealLiteralExp

Änderungen:

- Test auf primitiven Typ ergänzt

```
[1] The type of a real literal expression is the type Real.
context RealLiteralExp
inv: self.type.oclIsKindOf(Primitive) and self.type.getNameA() = 'Real'
```

StringLiteralExp

Änderungen:

- Test auf primitiven Typ ergänzt

```
[1] The type of a string literal expression is the type String.
context StringLiteralExp
inv: self.type.oclIsKindOf(Primitive) and self.type.getNameA() = 'String'
```

TupleLiteralExp

Änderungen:

- Die Spezifikation enthält widersprüchliche Angaben, ob eine `TupleLiteralExp` `VariableDeclaration` oder `TupleLiteralExpPart` aggregiert. Die Entscheidung hier fiel auf ersteres. Zum Vergleich mit den Attributen des Tupletyps wird die auf `VariableDeclaration` definierte Operation `asAttribute()` verwendet.

```
[1] The type of a TupleLiteralExp is a TupleType with the specified parts.
context TupleLiteralExp
inv: type.oclIsKindOf(TupleType) and
     let tt : TupleType = type.oclAsType(TupleType) in
     tuplePart.asAttribute() = tt.attributesA()->asSet()
```

Änderungen:

- Verwendung von `VariableDeclaration` statt `TupleLiteralExpPart.`

```
[2] All tuple literal expression parts of one tuple literal expression have unique names.
context TupleLiteralExp
```

```
inv: tuplePart->isUnique (vd : VariableDeclaration | vd.getNameA())
```

VariableDeclaration

[1] For initialized variable declarations, the type of the initExpression must conform to the type of the declared variable.

```
context VariableDeclaration
inv: initExpression->notEmpty() implies initExpression.type.conformsTo(type)
```

VariableExp

[1] The type of a VariableExp is the type of the variable to which it refers.

```
context VariableExp
inv: type = referredVariable.type
```

Anlage C – Spezifikation der Operationen

Im Folgenden werden die in `OCL::CommonModel`, `OCL::Types` und `OCL::Expressions` definierten Operationen spezifiziert. In einigen Fällen kann dies auf `Common-OCL`-Ebene geschehen. Die meisten Operationen kapseln jedoch Navigationen entlang interner Assoziationen des MOF- bzw. UML-Metamodells oder Zugriffe auf Attribute und sind dementsprechend für MOF und UML separat spezifiziert.

Paket OCL::CommonModel

AssociationClass

getEndsA

```
context UML::Core::AssociationClass
def: getEndsA() : Sequence(OCL::CommonModel::AssociationEnd) = self.connection
```

AssociationEnd

isBinaryA

```
context UML::Core::AssociationEnd
def: isBinaryA() : Boolean = self.association.connection->size() = 2
```

```
context MOF::Model::AssociationEnd
def: isBinaryA() : Boolean = true
```

isMultipleA

```
context UML::Core::AssociationEnd
def: isMultipleA() : Boolean =
    self.multiplicity.upperbound()> 1 or multiplicity.upperbound() = unlimited
```

```
context MOF::Model::AssociationEnd
def: isMultipleA() : Boolean =
    self.multiplicity.upper > 1 or self.multiplicity.upper = UNBOUNDED
```

isOrderedA

```
context UML::Core::AssociationEnd
def: isOrderedA() : Boolean = (self.ordering = OrderingKind::ok_ordered)
```

```
context MOF::Model::AssociationEnd
def: isOrderedA() : Boolean = self.multiplicity.isOrdered
```

isUniqueA

```
context OCL::CommonModel::AssociationEnd
def: isUniqueA() : Boolean = true
```

getQualifiersA

```
context UML::Core::AssociationEnd
def: getQualifiersA() : Sequence(OCL::CommonModel::Attribute) = self.qualifier
```

```
context MOF::Model::AssociationEnd
def: getQualifiersA() : Sequence(OCL::CommonModel::Attribute) = Sequence{}
```

getTypeA

```
context UML::Core::AssociationEnd
def: getTypeA() : OCL::CommonModel::Classifier = self.participant.toOclType()
```

```
context MOF::Model::AssociationEnd
def: getTypeA() : OCL::CommonModel::Classifier = self.type.toOclType()
```

Attribute

IsMultipleA

```
context UML::Core::Attribute
def: isMultipleA() : Boolean =
   self.multiplicity.upperbound() > 1 or multiplicity.upperbound() = unlimited
```

```
context MOF::Model::Attribute
def: isMultipleA() : Boolean =
   self.multiplicity.upper > 1 or self.multiplicity.upper = UNBOUNDED
```

isOrderedA

```
context UML::Core::Attribute
def: isOrderedA() : Boolean = (self.ordering = OrderingKind::ok_ordered)
```

```
context MOF::Model::Attribute
def: isOrderedA() : Boolean = self.multiplicity.isOrdered
```

isUniqueA

```
context UML::Core::Attribute
def: isUniqueA() : Boolean = false
```

```
context MOF::Model::Attribute
def: isUniqueA() : Boolean = self.multiplicity.isUnique
```

isInstanceLevelA

```
context UML::Core::Attribute
def: isInstanceLevelA() : Boolean = (ownerScope = ScopeKind::sk_instance)
```

```
context MOF::Model::Attribute
def: isInstanceLevelA() : Boolean = (scope = ScopeKind::instance_level)
```

getTypeA

```
context UML::Core::Attribute
def: getTypeA() : OCL::CommonModel::Classifier = self.type.toOclType()
```

```
context MOF::Model::Attribute
def: getTypeA() : OCL::CommonModel::Classifier = self.type.toOclType()
```

Classifier

conformsTo

Die Regeln für Typkonformität können weitestgehend aus [OCL16,S.3-3ff] unverändert für Common-OCL übernommen werden. Lediglich die Regeln [3] und [4] für Classifier [OCL16,S.3-4] sind UML-spezifisch:

[3] Classes conform to superclasses and interfaces that they realize.

```
context UML::Core::Class
inv: self.generalization.parent->forAll (p |
     (p.oclIsKindOf(Class) or p.oclIsKindOf(Interface)) implies
     self.conformsTo(p.oclAsType(Classifier)))
```

[4] Interfaces conforms to super interfaces.

```
context UML::Core::Interface
inv: self.generalization.parent->forAll (p |
     p.oclIsKindOf(Interface) implies self.conformsTo(p.oclAsType(Interface)))
```

Für MOF entfällt [4] und aus [3] ergibt sich:

```
context MOF::Model::Class
inv: self.supertype->forAll (p |
     p.oclIsKindOf(Class) implies self.conformsTo(p.oclAsType(Class)))
```

commonSuperType

Wie in [OCL16,S.3-22]:

```
context OCL::CommonModel::Classifier
def: commonSuperType(c : Classifier) : Classifier =
     Classifier.allInstances()->select (cst |
        c.conformsTo(cst) and
        self.conformsTo(cst) and
        not Classifier.allInstances()->exists (clst |
           c.conformsTo(clst) and
           self.conformsTo(clst) and
           clst.conformsTo(cst) and
           clst <> cst
        )
     )->any(true)
```

getCollectionType

```
context OCL::CommonModel::Classifier
def: getCollectionType() : OCL::CommonModel::CollectionType =
     collectionTypes->select(oclIsTypeOf(OCL::Types::CollectionType))->any()
```

getSetType

```
context OCL::CommonModel::Classifier
def: getSetType() : OCL::CommonModel::SetType =
     collectionTypes->select(oclIsTypeOf(OCL::Types::SetType))->any()
```

getBagType

```
context OCL::CommonModel::Classifier
def: getBagType() : OCL::CommonModel::BagType =
     collectionTypes->select(oclIsTypeOf(OCL::Types::BagType))->any()
```

getSequenceType

```
context OCL::CommonModel::Classifier
def: getSequenceType() : OCL::CommonModel::SequenceType =
     collectionTypes->select(oclIsTypeOf(OCL::Types::SequenceType))->any()
```

getOrderedSetType

```
context OCL::CommonModel::Classifier
def: getOrderedSetType() : OCL::CommonModel::OrderedSetType =
     collectionTypes->select(oclIsTypeOf(OCL::Types::OrderedSetType))->any()
```

attributesA

- liefert – im Gegensatz zu `allAttributesA()` – die Attribute in geordneter Reihenfolge. Da allerdings im Falle von Mehrfachvererbung in UML eine solche definierte Ordnung nicht gegeben ist, werden grundsätzlich die vererbten Attribute nicht betrachtet. Dies bringt jedoch keine Einschränkungen mit sich, da die WFRs, die Ordnung der Attribute voraussetzen, nur OCL-Typen betreffen, die keine Vererbung verwenden (siehe `OclMessageType`, `TupleType`).

```
context UML::Core::Classifier
def: attributesA() : Sequence(OCL::CommonModel::Attribute) =
     self.feature->select(f : Feature | f.oclIsKindOf(UML::Core::Attribute))
```

```
context MOF::Model::Classifier
def: attributesA() : Sequence(OCL::CommonModel::Attribute) =
     self.contents->select(me : ModelElement | f.oclIsKindOf(Attribute))
```

allAttributesA

```
context UML::Core::Classifier
def: allAttributesA() : Set(OCL::CommonModel::Attribute) =
     self.allAttributes()    -- see [UML15, p.2-61]
```

```
context MOF::Model::Classifier
def: allAttributesA() : Set(OCL::CommonModel::Attribute) =
     self.contents->select(me : ModelElement | me.oclIsKindOf(Attribute))
     ->union(supertypes.allAttributesA())
```

allAssociationEndsA

```
context UML::Core::Classifier
def: allAssociationEndsA() : Set(OCL::CommonModel::AssociationEnd) =
     self.allOppositeAssociationEnds()   -- see [UML15, p.2-62]
```

```
context MOF::Model::Classifier
def: allAssociationEndsA() : Set(OCL::CommonModel::AssociationEnd) =
     self.contents->select(me : ModelElement | me.oclIsKindOf(Reference))
     ->collect(me : ModelElement | me.oclAsType(Reference).exposedEnd)
     ->union(supertypes.allAttributesA())
```

allAssociationClassesA

```
context UML::Core::Classifier
def: allAssociationClassesA() : Set(OCL::CommonModel::AssociationClass) =
  allAssociations()->select(a : Association | a.oclIsKindOf(AssociationClass))
     -- for allAssociations() see [UML15, p.2-61]
```

```
context MOF::Model::Classifier
def: allAssociationClassesA() : Set(OCL::CommonModel::AssociationClass) =
     Set{}
```

allOperationsA

```
context UML::Core::Classifier
def: allOperationsA() : Set(OCL::CommonModel::Operation) =
     self.allOperations()    -- see [UML15, p.2-61]
```

```
context MOF::Model::Classifier
def: allOperationsA() : Set(OCL::CommonModel::Operation) =
     self.contents->select(me : ModelElement | me.oclIsKindOf(Operation))
     ->union(supertypes.allOperationsA())
```

allSignalsA

```
context UML::Core::Classifier
def: allReceptionsA() : Set(OCL::CommonModel::Signal) =
     self.allFeatures()->select(f : Feature | f.oclIsKindOf(Reception))
     ->collect(signal)
     -- for allFeatures see [UML15, p.2-61]
```

```
context MOF::Model::Classifier
def: allReceptionsA() : Set(OCL::CommonModel::Reception) = Set{}
```

lookup-Operationen

- wie in [OCL16,S.3-23], jedoch auf Common-OCL-Ebene:

```
context OCL::CommonModel::Classifier
def: lookupAttribute(attName : String) : OCL::CommonModel::Attribute =
     self.allAttributesA()->any(a | a.getNameA() = attName)

def: lookupAssociationEnd(name : String) : OCL::CommonModel::AssociationEnd =
     self.allAssociationEndsA()->any(ae | ae.getNameA() = name)

def: lookupAssociationClass(name : String):OCL::CommonModel::AssociationClass=
     self.allAssociationClassesA()->any (ac | ac.getNameA() = name)
```

```
def: lookupOperation(name : String, paramTypes: Sequence(Classifier))
          : OCL::CommonModel::Operation =
     self.allOperationsA()->any (op | op.getNameA() = name and
     op.hasMatchingSignature(paramTypes))

def: lookupSignal (sigName: String, paramTypes: Sequence(Classifier))
     : OCL::CommonModel::Signal =
     self.allSignalsA()->any(sig | sig.name = sigName and
     sig.hasMatchingSignature(paramTypes))
```

getPackageA

- innere Klassen werden hier nicht berücksichtigt

```
context UML::Core::Classifier
def: getPackageA():OCL::CommonModel::Package =
     self.namespace.oclAsType(Package)
```

```
context MOF::Model::Classifier
def: getPackageA():OCL::CommonModel::Package =
     self.container.oclAsType(Package)
```

getPathNameA

```
context OCL::CommonModel::Classifier
def: getPathNameA():String = getNameA().concat(getPackageA().getPathNameA())
```

Enumeration

getLiteralA

```
context UML::Core::Enumeration
def: getLiteralA() : Sequence(OCL::CommonModel::EnumerationLiteral) = literal
```

```
context MOF14OCL::Adapters::AdEnumeration
def: getLiteralA() : Sequence(OCL::CommonModel::EnumerationLiteral) = literal
```

getPathNameA

```
context MOF14OCL::Adapters::AdEnumeration
def: getPathNameA() : String = self.mofType.getPathNameA()
    -- always return the pathname of the 'original' enumeration
```

EnumerationLiteral

getEnumerationA

```
context UML::Core::EnumerationLiteral
def: getEnumerationA() : OCL::CommonModel::Enumeration = self.enumeration
```

```
context MOF14OCL::Adapters::AdEnumerationLiteral
def: getEnumerationA() : OCL::CommonModel::Enumeration = self.enumeration
```

ModelElement

getNameA

```
context UML::Core::ModelElement
def: getNameA() : String = self.name
```

```
context MOF::Model::ModelElement
def: getNameA() : String = self.name
```

setNameA

```
context UML::Core::ModelElement::setNameA(newName : String)
post: self.name = newName
```

```
context MOF::Model::ModelElement::setNameA(newName : String)
post: self.name = newName
```

NonOclClassifier

toOclType

- für UML: hier nicht spezifiziert, da keine vordefinierten Datentypen existieren (siehe Kapitel 5.2.4)

```
context MOF::Model::Classifier
def: toOclType() : OCL::CommonModel::Classifier
     = if self.oclIsKindOf(MOF::Model::Class) then self
       else if self.oclIsKindOf(MOF::Model::DataType) then
          self.oclAsType(MOF::Model::DataType).oclType
          else OclUndefined endif
       endif
```

Operation

isInstanceLevelA

```
context UML::Core::Operation
def: isInstanceLevelA() : Boolean = (ownerScope = ScopeKind::sk_instance)
```

```
context MOF::Model::Operation
def: isInstanceLevelA() : Boolean = (scope = ScopeKind::instance_level)
```

hasMatchingSignature

```
context OCL::CommonModel::Operation def:
hasMatchingSignature(pTypes : Sequence(OCL::CommonModel::Classifier)):Boolean
   = (getInAndInOutParametersA()->collect(p | p.getTypeA()) = pTypes)
```

getOutAndInOutParametersA()

```
context UML::Core::Operation
def: getOutAndInOutParametersA(): Sequence(OCL::CommonModel::Parameter) =
     self.parameter->select(p | p.kind = ParameterDirectionKind::pdk_out or
                                p.kind = ParameterDirectionKind::pdk_inout)
```

```
context MOF::Model::Operation
def: getOutAndInOutParametersA() : Sequence(OCL::CommonModel::Parameter) =
```

```
     self.contents->select(me : ModelElement | me.oclKindOf(Parameter))
     ->collect(me | me.oclAsType(MOF::Model::Parameter))
     ->select(p | p.direction = DirectionKind::out_dir or
                  p.direction = DirectionKind::inout_dir)
```

getOutParametersA()

```
context UML::Core::Operation
def: getOutParametersA(): Sequence(OCL::CommonModel::Parameter) =
     self.parameter->select(p | p.kind = ParameterDirectionKind::pdk_out)
```

```
context MOF::Model::Operation
def: getOutParametersA() : Sequence(OCL::CommonModel::Parameter) =
     self.contents->select(me : ModelElement | me.oclKindOf(Parameter))
     ->collect(me | me.oclAsType(MOF::Model::Parameter))
     ->select(p | p.direction = DirectionKind::out_dir)
```

getInAndInOutParametersA

```
context UML::Core::Operation
def: getInAndInOutParametersA(): Sequence(OCL::CommonModel::Parameter) =
     self.parameter->select(p | p.kind = ParameterDirectionKind::pdk_in or
                                p.kind = ParameterDirectionKind::pdk_inout)
```

```
context MOF::Model::Operation
def: getInAndInOutParametersA : Sequence(OCL::CommonModel::Parameter) =
     self.contents->select(me : ModelElement | me.oclKindOf(Parameter))
     ->collect(me | me.oclAsType(MOF::Model::Parameter))
     ->select(p | p.direction = DirectionKind::in_dir or
                  p.direction = DirectionKind::inout_dir)
```

getInParametersA

```
context UML::Core::Operation
def: getInParametersA(): Sequence(OCL::CommonModel::Parameter) =
     self.parameter->select(p | p.kind = ParameterDirectionKind::pdk_in)
```

```
context MOF::Model::Operation
def: getInParametersA : Sequence(OCL::CommonModel::Parameter) =
```

```
     self.contents->select(me : ModelElement | me.oclKindOf(Parameter))
     ->collect(me | me.oclAsType(MOF::Model::Parameter))
     ->select(p | p.direction = DirectionKind::in_dir)
```

getParametersA

```
context UML::Core::Operation
def: getParametersA() : Sequence(OCL::CommonModel::Parameter) = self.parameter
```

```
context MOF::Model::Operation
def: getNameA() : Sequence(OCL::CommonModel::Parameter) =
     self.contents->select(me : ModelElement | me.oclKindOf(Parameter))
     ->collect(me | me.oclAsType(Parameter))
```

getReturnParameterA

```
context UML::Core::Operation
def: getParametersA() : Sequence(OCL::CommonModel::Parameter) =
     self.parameter->select(p | p.kind = ParameterDirectionKind::pdk_return)
     ->first()
```

```
context MOF::Model::Operation
def: getNameA() : Sequence(OCL::CommonModel::Parameter) =
     self.contents->select(me : ModelElement | me.oclKindOf(Parameter))
     ->collect(me | me.oclAsType(MOF::Model::Parameter))
     ->select(p | p.direction = DirectionKind::return_dir)->first()
```

getOwnerA

```
context UML::Core::Operation
def: getOwnerA() : OCL::CommonModel::Classifier =
     self.owner
```

```
context MOF::Model::Operation
def: getOwnerA() : OCL::CommonModel::Classifier =
     self.container.oclAsType(Classifier)
```

Package

findClassifier

- innere Klassen werden hier nicht berücksichtigt

```
context UML::Model_Management::Package
def: findClassifier(path : Sequence(String)) : OCL::CommonModel::Classifier =
     let class :String = path->last() in
     let pkgname : Sequence(String) = path->subSequence(1, path->size()-1) in
     findPackage(pkgname).oclAsKind(Package).ownedElement->
         any(me | me.oclIsKindOf(UML::Core::Classifier) and
                  me.name = class).toOclType()
```

```
context MOF::Model::Package
def: findClassifier(path : Sequence(String)) : OCL::CommonModel::Classifier =
     let class :String = path->last() in
     let pkgname : Sequence(String) = path->subSequence(1, path->size()-1) in
     findPackage(pkgname).oclAsKind(MOF::Model::Package).contents->
         any(me | me.oclIsKindOf(MOF::Model::Classifier)
              and me.name = class).toOclType()
```

findPackage

```
context UML::Model_Management::Package
def: findPackage(pathname : Sequence(String)) : OCL::CommonModel::Package =
     pathname->iterate(name : String; p : Package = self | p.ownedElement->
     any(me | me.oclIsKindOf(Package) and me.name = name).oclAsType(Package))
```

```
context MOF::Model::Package
def: findPackage(pathname : Sequence(String)) : OCL::CommonModel::Package =
     pathname->iterate(name : String; p : Package = self | p.contents->
     any(me | me.oclIsKindOf(Package) and me.name = name).oclAsType(Package))
```

getPathNameA

```
context UML::Model_Management::Package
def: getPathNameA():String =
    if namespace.oclIsUndefined() then self.name
    else self.name.concat(namespace.oclAsType(Package).getPathNameA()) endif
```

```
context MOF::Model::Package
def: getPathNameA():String =
    if container.oclIsUndefined() then self.name
    else self.name.concat(container.oclAsType(Package).getPathNameA()) endif
```

asAttribute

```
context OCL::CommonModel::Parameter::asAttribute():OCL::CommonModel::Attribute
post: result.getNameA() = self.getNameA() and
      result.getTypeA() = self.getTypeA() and
      not result.isMultipleA() and
      result.isInstanceLevelA()
```

```
context UML::Core::Parameter::asAttribute() : OCL::CommonModel::Attribute
post: result.oclIsTypeOf(UML::Core::Attribute) and
      let attr: UML::Core::Attribute = result.oclAsType(UML::Core::Attribute)
      in attr.targetscope = ScopeKind::sk_instance and
         attr.visibility = VisibilityKind::vk_public and
         attr.stereotype.name = 'OclHelper'
```

```
context MOF::Model::Parameter::asAttribute() : OCL::CommonModel::Attribute
post: result.oclIsTypeOf(MOF::Model::Attribute) and
     let attr: MOF::Model::Attribute = result.oclAsType(MOF::Model::Attribute)
     in attr.visibility = VisibilityKind::public_vis
```

getTypeA

```
context UML::Core::Parameter
def: getTypeA() : OCL::CommonModel::Classifier = self.type.toOclType()
```

- für MOF wird die Multiplizität des Parameters in die Typabbildung einbezogen

```
context MOF::Model::Parameter
def: getTypeA() : OCL::CommonModel::Classifier =
     let oclType : OCL::CommonModel::Classifier = self.type.toOclType() in
     if  multiplicity.upper > 1 or multiplicity.upper = UNBOUNDED then
         if multiplicity.isOrdered then
            if multiplicity.isUnique then oclType.getOrderedSetType()
            else oclType.getSequenceType() endif
        else
```

```
            if multiplicity.isUnique then oclType.getSetType()
            else oclType.getBagType() endif
         endif
      else
         oclType
      endif
```

Signal

hasMatchingSignature

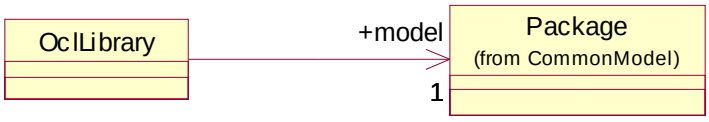
```
context UML::Common_Behavior::Signal
def:
hasMatchingSignature(pTypes : Sequence(OCL::CommonModel::Classifier)):Boolean
   = let sigPTypes: Sequence(Classifier) = attributesA().getTypeA()
     in (( sigPTypes->size() = pTypes->size()) and
         ( Set{1..pTypes->size()}
             ->forAll ( i | pTypes->at (i).conformsTo (sigPTypes->at (i)))))
```

Pakete OCL::Expressions und OCL::Types

OclLibrary

`OclLibrary` ist eine Hilfsklasse für den einfachen Zugriff auf die in jedem Modell vorhandenen primitiven Datentypen der OCL Standard Library. Des Weiteren bietet sie Methoden, um Instanzen von `TupleType` und `OclMessageType` zu erzeugen.

Für jedes Modell liefert die Klassenoperation `getInstance()` die Singleton-Instanz von `OclLibrary`. Bei der folgenden Spezifikation wird davon ausgegangen, dass das Modell durch ein Paket `topPackage` repräsentiert wird, welches alle anderen Pakete, Klassen etc. enthält.



getInstance

```
context OCL::Types::OclLibrary::
        getInstance(topPackage : OCL::CommonModel::Package):OclLibrary
post: result.model = topPackage
```

```
context OCL::Types::OclLibrary
inv: OclLibrary::allInstances()->forAll(ol | ol.model = self.model
                                        implies ol = self)
```

getOclLibPackage

- das Paket im Modell, dass die Typen der OCL Standard Library enthält. Mangels Definition in der OCL-Spezifikation wurde diese mit „OclLib“ benannt.

```
context OCL::Types::OclLibrary
def: getOclLibPackage():OCL::CommonModel::Package =
         self.model.findPackage(Sequence{'OclLib'})
```

contains

- prüft, ob die Operation zur OCL Standard Library gehört

```
context OCL::Types::OclLibrary
def: contains(op : OCL:CommonModel::Operation) : Boolean =
    op.getOwnerA().getPackageA() = getOclLibPackage()
```

getOclVoid

```
context OCL::Types::OclLibrary
def: getOclVoid():OCL::Types::VoidType =
   getOclLibPackage().findClassifier(Sequence{'OclVoid'})->oclAsType(VoidType)
```

getOclAny

```
context OCL::Types::OclLibrary
def: getOclAny():OCL::CommonModel::Classifier =
   getOclLibPackage().findClassifier(Sequence{'OclAny'})
```

getOclInteger

```
context OCL::Types::OclLibrary
def: getOclInteger():OCL::CommonModel::Primitive =
  getOclLibPackage().findClassifier(Sequence{'Integer'})->oclAsType(Primitive)
```

getOclReal

```
context OCL::Types::OclLibrary
def: getOclReal():OCL::CommonModel::Primitive =
  getOclLibPackage().findClassifier(Sequence{'Real'})->oclAsType(Primitive)
```

getOclString

```
context OCL::Types::OclLibrary
def: getOclString():OCL::CommonModel::Primitive =
  getOclLibPackage().findClassifier(Sequence{'String'})->oclAsType(Primitive)
```

getOclBoolean

```
context OCL::Types::OclLibrary
def:getOclReal():OCL::CommonModel::Primitive =
  getOclLibPackage().findClassifier(Sequence{'Boolean'})->oclAsType(Primitive)
```

makeOclMessageType

```
context OCL::Types::OclLibrary::
makeOclMessageType(sig : OCL::CommonModel::Signal):OCL::Types::OclMessageType
post: result.getPackageA() = self.getOclLibPackage() and
      result.referredSignal = sig and
      let attrRes: Sequence(Attributes) = result.attributesA() in
      let attrSig : Sequence(Attributes) = sig.attributesA()
      in
      attrRes->size() = attrSig->size() and
      Sequence{1..attrRes->size()}->forAll(n |
       attrRes->at(n).getNameA() = attrSig->at(n).getNameA() and
       attrRes->at(n).getTypeA() = attrSig->at(n).getTypeA())
```

```
context OCL::Types::OclLibrary::
makeOclMessageType(op:OCL::CommonModel::Operation):OCL::Types::OclMessageType
post: result.getPackageA() = self.getOclLibPackage() and
      result.referredOperation = op and
      result.attributesA()= op.getParametersA().asAttribute()
```

makeTupleType

```
context OCL::Types::OclLibrary::
makeTupleType(atts : Sequence(OCL::CommonModel::Attribute)):TupleType
post: result.getPackageA() = self.getOclLibPackage() and
      result.attributesA() = atts
```

VariableDeclaration

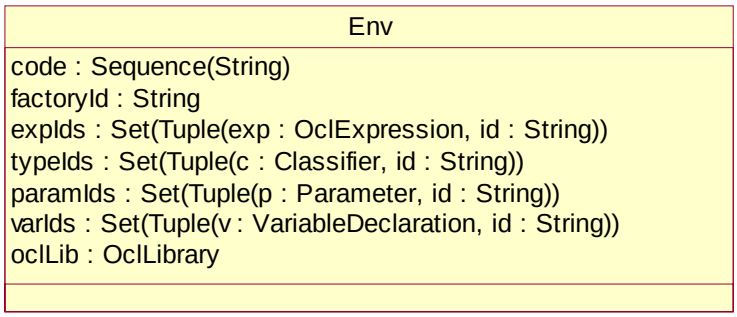
asAttribute

```
context OCL::Expressions::VariableDeclaration::
        asAttribute():OCL::CommonModel::Attribute
post: result.getNameA() = self.getNameA() and
      result.getTypeA() = self.type
```

Anlage D – Spezifikation der Codegenerierung

Umgebung für die Codegenerierung

Env



init

```
context Env::init(preCode : Sequence(String),
                  factoryId : String,
                  model : Package) : Env
post: result.code = preCode and
      result.factoryId = factoryId and
      result.expIds = Set{}and
      result.typeIds = Set{} and
      result.paramIds = Set{} and
      result.varIds = Set{} and
      result.oclLib = OclLibrary::getInstance(model)
```

appendLine

```
context Env
def: appendLine(template : String, args : Sequence(String)) : Env
     = appendLine(Template::fill(template, args))
```

```
context Env::appendLine(line : String) : Env
post: result.code = self.code.append(line) and
      result.factoryId = self.factoryId and
      result.expIds = self.expIds and
      result.typeIds = self.typeIds and
      result.paramIds = self.paramIds and
      result.varIds = self.varIds and
```

```
      result.oclLib = self.oclLib
```

getId

```
context Env
def: getId(exp : OclExpression) : String =
    if exp.oclIsKindOf(VariableExp) then
        getVarId(exp.oclAsType(VariableExp).referredVariable)
    else
        expIds->select(tuple | tuple.exp = exp).id
    endif
```

```
context Env
def: getId(exp : Classifier) : String =
    typeIds->select(tuple | tuple.exp = exp).id
```

createId

```
context Env::createId(anExp : OclExpression) : Env
pre: self.getId(anExp).oclIsUndefined()
post: result.code = self.code and
      result.factoryId = self.factoryId and
      result.expIds = self.expIds->including( Tuple{exp = anExp,
          value = 'tudOcl20Exp'.concat(self.expIds->size().toString())}) and
      result.typeIds = self.typeIds and
      result.paramIds = self.paramIds and
      result.varIds = self.varIds and
      result.oclLib = self.oclLib
```

```
context Env::createId(cl : Classifier) : Env
pre: self.getId(cl).oclIsUndefined()
post: result.code = self.code and
      result.factoryId = self.factoryId and
      result.expIds = self.expIds and
      result.typeIds = self. typeIds->including( Tuple{c = cl,
          value = 'tudOcl20Type'.concat(self.typeIds->size().toString())}) and
      result.paramIds = self.paramIds and
      result.varIds = self.varIds and
      result.oclLib = self.oclLib
```

getAccuId

```
context Env
def: getAccuId(exp : IterateExp) : String = getId(exp).concat('Accu')
```

getIteratorId

```
context Env
def: getIteratorId(exp : LoopExp) : String = getId(exp).concat('Iter')
```

getEvalId

```
context Env
def: getEvalId(exp : LoopExp) : String = getId(exp).concat('Eval')
```

getParamId

```
context Env
def: getParamId (p : Parameter) : String =
    paramIds->select(tuple | tuple.exp = exp).id
```

createParamId

```
context Env::createParamId(param : Parameter) : Env
pre: self.getParamId(param).oclIsUndefined()
post: result.code = self.code and
      result.factoryId = self.factoryId and
      result.expIds = self.expIds and
      result.typeIds = self.typeIds and
      result.paramIds = self.paramIds->including(Tuple{p = param,
        value = 'tudOcl20Param'.concat(self.paramIds->size().toString())}) and
      result.varIds = self.varIds and
      result.oclLib = self.oclLib
```

getVarId

```
context Env
def: getVarId (vd : VariableDeclaration) : String =
    varIds->select(tuple | tuple.exp = exp).id
```

createVarId

```
context Env::createVarId(var : VariableDeclaration) : Env
pre: self.getVarId(var).oclIsUndefined()
post: result.code = self.code and
      result.factoryId = self.factoryId and
      result.expIds = self.expIds and
      result.typeIds = self.typeIds and
      result.paramIds = self.paramIds and
      result.varIds = self. varIds ->including(Tuple{v = var,
          value = 'tudOcl20Var'.concat(self.varIds->size().toString())}) and
      result.oclLib = self.oclLib
```

bind

```
context Env::bind(var : VariableDeclaration, exp : OclExpression) : Env
pre: self.getVarId(var).oclIsUndefined()
post: result.code = self.code and
      result.factoryId = self.factoryId and
      result.expIds = self.expIds and
      result.typeIds = self.typeIds and
      result.paramIds = self.paramIds and
      result.varIds = self. varIds ->including(
          Tuple{v = var, value = self.getId(exp)}) and
      result.oclLib = self.oclLib
```

Initiale Umgebung der Codegenerierung für JMI

Um für einen OCL-Ausdruck `exp:OclExpression` über einem Metamodell, repräsentiert durch das Paket `metamodel:Package` Code zu generieren, ist Folgendes auszuwerten:

```
exp.appendJavaCode(Env::init(
    Sequence{'JmiOclFactory tudOcl20Fact = JmiOclFactory.getInstance(model)'},
    'tudOcl20Fact',
    metamodel
)).code
```

Codegenerierung für die Unterklassen von `OclExpression`

Die folgenden Definitionen beziehen sich alle auf das Paket `OCL::Expressions` und sind unabhängig von der jeweiligen konkreten Implementationen der OCL-Basisbibliothek (z.B. für JMI oder für Java-Reflection).

```
package OCL::Expressions
```

AttributeCallExp

```
context AttributeCallExp
def: appendJavaCode(env: Env) : Env =
let attrName : String = referredAttribute.getNameA() in
let javaOclType: String = self.type.mapToJavaOcl(env, true) in
let withId : Env = env.createId(self) in
let isClassAttr: Boolean = source.oclIsUndefined() in
let withSrc: Env = if isClassAttr then
                       srcType.appendJavaCode(withId)
                   else source.appendJavaCode(withId) endif in
let srcId : String = if isClassAttr then
                       withSrc.getTypeId(srcType)
                     else withSrc.getId(source) endif in
    if not source.type.oclIsKindOf(TupleType) then
        let withType: Env = self.type.appendJavaCode(withSrc) in
        withType.appendLine('final $1 $2 = $3;',
            Sequence{javaOclType, withType.getId(self),
                Cast::withCast(javaOclType,'$1.getFeature($2, $3)',
                            Sequence{srcId,
                                    withType.getTypeId(self.type),
                                    attrName})})
    else
        withSource.appendLine('final $1 $2 = $3;',
            Sequence{javaOclType, withSource.getId(self),
                Cast::withCast(javaOclType,'$1.getFeature($2)',
                            Sequence{srcId,
                                    attrName})})
    endif
```

BooleanLiteralExp

```
context BooleanLiteralExp
def: appendJavaCode(env: Env) : Env =
let withId : Env = env.createId(self) in
withId.appendLine('final OclBoolean $1 = OclBoolean.$2;',
                  Sequence{
                      withId.getId(self),
                      if booleanSymbol then 'TRUE' else 'FALSE' endif
                  })
```

CollectionItem

```
context CollectionItem
def: appendJavaCode(env: Env, collId : String) : Env =
let withItemExp : Env = self.item.appendJavaCode(env) in
withItemExp.appendLine('$1.setToInclude($2);',
                       Sequence{collId, withItemExp.getId(self.item)})
```

CollectionLiteralExp

```
context CollectionLiteralExp
def: appendJavaCode(env: Env) : Env =
let withId : Env = env.createId(self) in
let collId: String = withId.getId(self) in
let withEmptyColl : Env = withId.appendLine('final $1 $2 = $1.getEmpty$1();',
               Sequence{self.type.mapToJavaOcl(withEmptyColl, true),collId})
in parts->iterate(p : CollectionLiteralPart; acc : Env = withEmptyColl |
               p.appendJavaCode(acc))
```

CollectionRange

```
context CollectionRange
def: appendJavaCode(env: Env, collId : String) : Env =
let withFirst : Env = self.first.appendJavaCode(env) in
let withLast : Env = self.last.appendJavaCode(withFirst) in
withItemExp.appendLine('$1.setToRange($2, $3);',
                       Sequence{collId,
                                withLast.getId(self.first),
                                withLast.getId(self.last)})
```

EnumLiteralExp

```
context EnumLiteralExp
def: appendJavaCode(env: Env) : Env =
let withId : Env =self.type.appendJavaCode(env).createId(self) in
withId.appendLine('final OclEnumLiteral $1 = $2.getLiteralFor($3);',
                  Sequence{withId.getId(self),
                           withId.getId(self.type),
```

```
                                referredEnumLiteral.getNameA()})
```

IfExp

```
context IfExp
def: appendJavaCode(env: Env) : Env =
let javaOclType : String = self.type.mapToJavaOcl(env, true) in
let withCond : Env = self.condition.appendJavaCode(env) in
let withThen : Env = self.thenExpression.appendJavaCode(withCond) in
let withElse : Env = self.elseExpression.appendJavaCode(withThen) in
let withId : Env = withElse.createId(self) in
withId.appendLine('final $1 $2 = $3;',
    Sequence{javaOclType, withId.getId(self),
        Cast::withCast(javaOclType,'$1.ifThenElse($2,$3)',
                                    Sequence{withId.getId(condition),
                                             withId.getId(thenExpression),
                                             withId.getId(elseExpression)})})
```

IntegerLiteralExp

```
context IntegerLiteralExp
def: appendJavaCode(env: Env) : Env =
let withId : Env = env.createId(self) in
withId.appendLine('final OclInteger $1 = new OclInteger($2);',
                  Sequence{withId.getId(self), integerSymbol.toString()})
```

IterateExp

- max. eine Iteratorvariable wird unterstützt

```
context IterateExp
def: appendJavaCode(env: Env) : Env =
if iterators->size() > 1 then OclUndefined else
let withSrc : Env = self.source.appendJavaCode(env) in
let withId : Env = withSrc.createId(self) in
let accuInit : OclExpression = self.result.initExpression in
let withAccuInit : Env = accuInit.appendJavaCode(withId) in
let accuId : String = withAccuInit.getAccuId(self) in
let withContainer : Env =
    withAccuInit.appendLine('final OclContainer $1 = new OclContainer($2)',
```

```
                Sequence{accuId, withAccuInit.getId(result.initExpression)}) in
let itId : String = withContainer.getIteratorId(self) in
let withIterator: Env =
withContainer.appendLine('final OclIterator $1 = $2.getIterator();',
                        Sequence{itId, withContainer.getId(source)}) in
let withEvalClass : Env =
withIterator.appendLine(
    'final OclRootEvaluatable $1=new OclRootEvaluatable(){',
        Sequence{withEvalId.getEvalId(self)}).appendLine(
            'public OclRoot evaluate(){') in
let itVar : VariableDeclaration = iterators->any(true) in
let itVarType : String = itVar.type.mapToJavaOcl(env, true) in
let withItVarId : Env = withEvalClass.createVarId(itVar) in
let withUnwrapItVar: Env =
withItVarId.appendLine('final $1 $2 = $3;',
    Sequence{itVarType, withItVarId.getId(itVar),
        Cast::withCast(itVarType,'$1.getValue()',
                                    Sequence{itId})}) in
let accuType : String = result.type.mapToJavaOcl(env, true) in
let withAccuVarId: Env = withUnwrapItVar.createVarId(result) in
let withUnwrapAccu: Env =
withAccuVarId.appendLine('final $1 $2 = $3;',
    Sequence{accuType, withAccuVarId.getId(result),
        Cast::withCast(accuType,'$1.getValue()',Sequence{accuId})})
in
let withBody : Env = self.body.appendJavaCode(withUnwrapAccu) in
let withReturn : Env =
withBody.appendLine('return $1;}};', Sequence{withBody.getId(body)}) in
let resultType = self.type.mapToJavaOcl(env, true)
in
withReturn.appendLine('final $1 $2 = $3;',
    Sequence{resultType, withReturn.getId(self),
        Cast::withCast(resultType,'$1.iterate($2, $3, $4)',
                            Sequence{withReturn.getId(source),
                                    itId, accuId,
                                    withReturn.getEvalId(self)})})
endif
```

IteratorExp

- max. eine Iteratorvariable wird unterstützt

```
context IteratorExp
def: appendJavaCode(env: Env) : Env =
if iterators->size() > 1 then OclUndefined else
let withSrc : Env = self.source.appendJavaCode(env) in
let withId : Env = withSrc.createId(self) in
let itId : String = withId.getIteratorId(self) in
let withIterator: Env =
withContainer.appendLine('final OclIterator $1 = $2.getIterator();',
                      Sequence{itId, withContainer.getId(source)}) in
let withEvalClass : Env = withIterator.appendLine('final $1 $2=new $1(){',
     Sequence{getJavaOclEvalName(), withEvalId.getEvalId(self)}).appendLine(
      'public $1 evaluate(){',Sequence{getJavaOclEvalType()}) in
let itVar : VariableDeclaration = iterators->any(true) in
let itVarType : String = itVar.type.mapToJavaOcl(env, true) in
let withItVarId : Env = withEvalClass.createVarId(itVar) in
let withUnwrapItVar: Env =
withItVarId.appendLine('final $1 $2 = $3;',
    Sequence{itVarType, withItVarId.getId(itVar),
        Cast::withCast(itVarType,'$1.getValue()',
                                Sequence{itId})}) in
let withBody : Env = self.body.appendJavaCode(withUnwrapItVar) in
let withReturn : Env =
withBody.appendLine('return $1;}};', Sequence{withBody.getId(body)}) in
let resultType = self.type.mapToJavaOcl(env, true)
in
withReturn.appendLine('final $1 $2 = $3;',
    Sequence{resultType, withReturn.getId(self),
        Cast::withCast(resultType,'$1.$2($3, $4)',
                          Sequence{withReturn.getId(source),
                                   self.getNameA(), itId,
                                   withReturn.getEvalId(self)})})
endif
```

```
context IteratorExp
def: getJavaOclEvalType() : String =
if Set{'forAll', 'exists', 'select', 'reject', 'any', 'one'}
    ->includes(getNameA()) then 'OclBoolean'
else
    if getNameA() = 'sortedBy' then 'OclComparable'
    else 'OclRoot' endif
endif
```

```
context IteratorExp
def: getJavaOclEvalName():String = getJavaOclEvalType().concat('Evaluatable')
```

LetExp

```
context LetExp
def: appendJavaCode(env : Env) : Env =
let initExp : OclExpression = variable.initExpression in
let withInitExp : Env = initExp.appendJavaCode(env) in
in.appendJavaCode(withInitExp.bind(variable, initExp))
```

NavigationCallExp

```
context NaviagtionCallExp
def: appendJavaCode(env: Env) : Env =
let withSrc : Env = self.source.appendJavaCode(env) in
let featureName =
     if self.oclIsKindOf(AttributeCallExp) then
       self.oclAsType(AttributeCallExp).referredAttribute.getNameA()
     else
       self.oclAsType(AssociationClassCallExp).
         referredAssociationClass.getNameA()
    endif in
let withId : Env = withSrc.createId(self) in
let withType: Env = self.type.appendJavaCode(withId) in
let javaOclType: String = self.type.mapToJavaOcl(env, true) in
withType.appendLine('final $1 $2 = $3;',
    Sequence{javaOclType, withType.getId(self),
        Cast::withCast(javaOclType,'$1.getFeature($2, $3)',
                        Sequence{withType.getId(source),
```

```
                                    withType.getTypeId(self.type),
                                    featureName})})
```

OclOperationWithTypeArgExp

```
context OclOperationWithTypeArgExp
def: appendJavaCode(env: Env) : Env =
let withSrc : Env = self.source.appendJavaCode(env) in
let srcId : String = withSrc.getId(source) in
let withId: Env = withSrc.createId(self) in
let selfId : String = withId.getId(self) in
let withTypeArg: Env = self.typeArgument.appendJavaCode(withId) in
let opCall : String = Template::fill('$1.$2($3)',
            Sequence{srcId, getNameA(), withTypeArg.getTypeId(typeArgument)})
in
    let castedOpCall : String =
        if OperationHelper::needsCast(implName) then
            Cast::withCast(javaOclType, opCall)
        else opCall endif in
    withTypeArg.appendLine('final $1 $2 = $3',
                            Sequence{javaOclType, selfId, castedOpCall})
```

OperationCallExp

```
context OperationCallExp
def: appendJavaCode(env: Env) : Env =
let op : Operation = referredOperation in
let opName : String = op.getNameA() in
let javaOclType: String = self.type.mapToJavaOcl(env, true) in
let withId : Env = env.createId(self) in
let selfId : String = withId.getId(self) in
let isClassOp: Boolean = source.oclIsUndefined() in
let withSrc: Env = if isClassOp then
                        srcType.appendJavaCode(withId)
                   else source.appendJavaCode(withId) endif in
let srcId : String = if isClassOp then
                        withSrc.getTypeId(srcType)
                     else withSrc.getId(source) endif in
let withArguments : Env = arguments->iterate(arg : OclExpression; acc : Env |
                                            arg.appendJavaCode(acc)) in
if withArguments.oclLib.contains(op) then
```

```
    -- an OCL Standard Library Operation
    let implName = OperationHelper::mapOpName(opName) in
    let argumentIds: String = Template::commaSepList('$1',arguments
        ->collect(arg | withArguments.getId(arg))) in
    let opCall : String = Template::fill('$1.$2($3)',
                            Sequence{srcId,implName, argumentIds }) in
    let castedOpCall : String =
        if OperationHelper::needsCast(implName) then
            Cast::withCast(javaOclType, opCall)
        else opCall endif in
    withArguments.appendLine('final $1 $2 = $3',
                            Sequence{javaOclType, selfId, castedOpCall})
else
    let params : Sequence(Parameter) = op.getParametersA() in
    let inParams : Sequence(Parameter) = getInParametersA() in
    let inIoParams : Sequence(Parameter) = getInAndInOutParametersA() in
    let outParams : Sequence(Parameter) = getInAndInOutParametersA() in
    let withInIoParams : Env =
     Sequence{1..inIoParams->size}->iterate(n; acc : Env = withArguments |
        let p : Parameter = inIoParams->at(n) in
        let argId : String = withArguments.getId(arguments->at(n)) in
        let withParamId : Env = acc.createParamId(p) in
        let pId : String = withParamId.getParamId(p) in
        let javaNonOclType : String = p.getJavaNonOclTypeExp() in
        if inParams->includes(p) then
         acc.appendLine('final OclParameter $1 = new OclParameter($2, $3);',
                        Sequence{pId, javaNonOclType, argId})
        else
         --an inout parameter ... it's name is needed for proper assignment
         --in the result tuple of the operation
         acc.appendLine('final OclParameter $1 = new OclParameter($2,$3,$4);',
                        Sequence{pId, p.getNameA(), javaNonOclType, argId})
        endif
     ) in
    let withOutParams : Env =
     outParams->iterate(p : Parameter; acc : Env = withInIoParams |
        let withParamId : Env = acc.createParamId(p) in
        let pId : String = withParamId.getParamId(p) in
        let javaNonOclType : String = p.getJavaNonOclTypeExp(withParamId) in
         --an out parameter ... it's name is needed for proper assignment
         --in the result tuple of the operation
         acc.appendLine('final OclParameter $1 = new OclParameter($2,$3);',
                        Sequence{pId, p.getNameA(), javaNonOclType})
```

```
        endif
     ) in
    let paramIds : String = Template::commaSepList('$1',params
        ->collect(p | withOutParams.getParamId(p))) in
    let withResultType : Env = type.appendJavaCode(withOutParams) in
    withResultType.appendLine('final $1 $2 = $3;', Sequence{javaOclType,
        selfId, Cast::withCast(javaOclType,
            '$1.getFeature($2, $3, new OclParameter[]{$4})',
            Sequence{srcId, withResultType.getTypeId(self.type),
                                    opName, paramIds})})
endif
```

RealLiteralExp

```
context RealLiteralExp
def: appendJavaCode(env: Env) : Env =
let withId : Env = env.createId(self) in
withId.appendLine('final OclReal $1 = new OclReal($2);',
                  Sequence{withId.getId(self), realSymbol.toString()})
```

StringLiteralExp

```
context StringLiteralExp
def: appendJavaCode(env: Env) : Env =
let withId : Env = env.createId(self) in
withId.appendLine('final OclString $1 = new OclString("$2");',
                  Sequence{withId.getId(self), stringSymbol})
```

TupleLiteralExp

```
context TupleLiteralExp
def: appendJavaCode(env: Env) : Env =
let withParts : Env = tuplePart->iterate(vd : VariableDeclaration; acc : Env
    | vd.initExpression.appendJavaCode(acc)) in
let partNames : String =
    Template::commaSepList('"$1"',tuplePart->collect(getNameA())) in
let partValues : String =
    Template::commaSepList('$1',tuplePart
    ->collect(vd | env.getId(vd.initExpression))) in
let withId : Env = withParts.createId(self) in
```

```
withId.appendLine(
    'OclTuple $1 = new OclTuple(new String []{$2}, new OclRoot[]{$3});',
    Sequence{withId.getId(self),partNames,partValues})
```

VariableExp

```
context VariableExp
def: appendJavaCode(env : Env) : Env = env
```

Typabbildung OCL-Metamodell -> OCL-Basisbibliothek

BagType

```
context OCL::Types::BagType
def: mapToJavaOcl(env: Env, asInstance : Boolean) : String =
    if asInstance then 'OclBag' else 'OclCollectionType' endif
```

CollectionType

```
context OCL::Types::CollectionType
def: mapToJavaOcl (env: Env, asInstance : Boolean) : String =
    if asInstance then 'OclCollection' else 'OclCollectionType' endif
```

```
context OCL::Types::CollectionType
def: mapToJavaOclTypeExp(env: Env) : String =
    Template::fill('$1.get$2()',
                   Sequence{elementType.mapToJavaOclTypeExp(),
                            self.mapToJavaOcl(false)})
```

Classifier

```
context OCL::CommonModel::Classifier
def: appendJavaCode(env: Env) : Env =
if code.getId(this).oclIsUndefined then
let withId : Env = env.createId(self) in
    withId.appendLine('final $1 $2 = $3;'
                      Sequence{self.mapToJavaOcl(env, false),
```

```
                                withId.getId(self),
                                self.mapToJavaOclTypeExp(env)})
else code endif
```

```
context OCL::CommonModel::Classifier
def: mapToJavaOcl(env: Env, asInstance : Boolean) : String =
if self=env.oclLib.getOclAny() then
    if asInstance then 'OclAny' else 'OclType' endif
else
    if asInstance then 'OclModelObject' else 'OclModelType' endif
endif
```

```
context OCL::CommonModel::Classifier
def: mapToJavaOclTypeExp(env: Env) : String =
if self=env.oclLib.getOclAny() then 'OclType.getOclAny()'
else
    Template::fill('$1.getOclModelTypeFor("$2")',
                   Sequence{env.factoryId, self.getPathNameA()})
endif
```

Enumeration

```
context OCL::CommonModel::Enumeration
def: mapToJavaOcl(env: Env, asInstance : Boolean) : String =
    if asInstance then 'OclEnumLiteral' else 'OclEnumType' endif
```

```
context OCL::CommonModel::Enumeration
def: mapToJavaOclTypeExp(env: Env) : String =
    Template::fill('$1.getOclEnumTypeFor("$2")',
                   Sequence{env.factoryId, self.getPathNameA()})
endif
```

OrderedSetType

```
context OCL::Types::OrderedSetType
def: mapToJavaOcl(env: Env, asInstance : Boolean) : String =
    if asInstance then 'OclOrderedSet' else 'OclCollectionType' endif
```

Primitive

```
context OCL::CommonModel::Primitive
def: mapToJavaOcl (env: Env, asInstance : Boolean) : String =
if self=env.oclLib.getOclInteger() then
    if asInstance then 'OclInteger' else 'OclPrimitiveType' endif
else if self=env.oclLib.getOclReal() then
    if asInstance then 'OclReal' else 'OclPrimitiveType' endif
else if self=env.oclLib.getOclBoolean() then
    if asInstance then 'OclBoolean' else 'OclPrimitiveType' endif
else if self=env.oclLib.getOclString() then
    if asInstance then 'OclString' else 'OclPrimitiveType' endif
else
    OclUndefined    -- unknown primitive
endif endif endif endif
```

```
context OCL::CommonModel::Primitive
def: mapToJavaOclTypeExp(env: Env) : String =
if self=env.oclLib.getOclInteger() then
    Template::fill('OclPrimitiveType.get$1()'
                   Sequence{self.mapToJavaOcl(env, true)})
endif
```

SequenceType

```
context OCL::Types::SequenceType
def: mapToJavaOcl(env: Env, asInstance : Boolean) : String =
    if asInstance then 'OclSequence' else 'OclCollectionType' endif
```

SetType

```
context OCL::Types::SetType
def: mapToJavaOcl(env: Env, asInstance : Boolean) : String =
    if asInstance then 'OclSet' else 'OclCollectionType' endif
```

TupleType

```
context OCL::Types::TupleType
def: mapToJavaOcl(env: Env, asInstance : Boolean) : String =
```

```
    if asInstance then 'OclTuple' else 'OclTupleType' endif
```

```
context OCL::Types::TupleType
def: mapToJavaOclTypeExp(env: Env) : String =
let attrNames : String =
    Template::commaSepList('"$1"', attributesA().getNameA()) in
let attrTypes : String =
    Template::commaSepList('$1', attributesA().getTypeA().mapToJavaOcl(env))
in
    Template::fill('$1.getOclTupleType(new String[]{$2}, new OclType[]{$3}',
                    Sequence{env.factoryId, attrNames, attrTypes})
```

VoidType

```
context OCL::Types::VoidType
def: mapToJavaOcl(env: Env, asInstance : Boolean) : String =
    if asInstance then 'OclUndefined' else 'OclType' endif
```

```
context OCL::Types::VoidType
def: mapToJavaOclTypeExp(env: Env) : String = 'OclType.getOclVoid()'
```

Typabbildung OCL-Basisbibliothek -> JMI

Folgende Definitionen erzeugen Code, der spezifisch für die JMI-Implementation der OCL-Basisbibliothek Informationen darüber liefert, welchen JMI-Typ ein Parameter einer Operation hat. Für Implementationen der OCL-Basisbibliothek, für die eine solche Information redundant ist, sollte `Parameter::getJavaNonOclTypeExp()` immer `'null'` liefern.

```
context MOF::Model::Parameter
def:: getJavaNonOclTypeExp() : String =
    let result : String = self.type.mapToJavaNonOclTypeExp() in
    let upper : Integer = multiplicity.upper in
    if upper > 1 or upper = -1 then
        if multiplicity.isOrdered then
            result.concat('.getListType()')
        else
            result.concat('.getCollectionType()')
        endif
    else result endif
```

AliasType

```
context MOF::Model::AliasType
def:: mapToJavaNonOclTypeExp() : String = self.type.mapToJavaNonOclTypeExp()
```

Class

```
context MOF::Model::Class
def:: mapToJavaNonOclTypeExp() : String = 'JmiType.MODELTYPE'
```

CollectionType

```
context MOF::Model:: CollectionType
def:: mapToJavaNonOclTypeExp(env: Env) : String =
    let result : String = self.type.mapToJavaNonOclTypeExp() in
    if multiplicity.isOrdered then
        result.concat('.getListType()')
    else
        result.concat('.getCollectionType()')
    endif
```

EnumerationType

```
context MOF::Model:: EnumerationType
def:: mapToJavaNonOclTypeExp(env: Env) : String = JmiType.ENUMTYPE
```

PrimitiveType

```
context MOF::Model::PrimitiveType
def:: mapToJavaNonOclTypeExp(env: Env) : String = 'JmiType.'.concat(
   if name = Boolean then BOOLEAN else
   if name = Integer then INTEGER else
   if name = Long then LONG else
   if name = Float then FLOAT else
   if name = Double then DOUBLE else
   if name = String then STRING else
   endif endif endif endif endif endif
)
```

StructureType

```
context MOF::Model:: StructureType
def:: mapToJavaNonOclTypeExp(env: Env) : String =
let fields: Sequence(StructureField) =
    contents->select(me : ModelElement | me.oclIsKindOf(StructureField))
        ->collect(me | me.oclAsType(StructureField)) in
let fieldnames : String =
    Template::commaSepList('"$1"', fields ->collect(name)) in
let fieldTypes : String =
    Template::commaSepList('$1', fields ->collect(type)) in
let pathname : String = self.getPathNameA() in
Template::fill(
    '$1.getJmiStructTypeFor($2, new String[]{$3}, new JmiType[]{$4})',
     Sequence{env.factoryId, pathname, fieldnames, fieldtypes})
```

Hilfsklassen

Alle im Folgenden spezifizierten Operationen sind Klassenoperationen.

Cast

withCast

- umschließt einen Ausdruck mit einem Type-Cast

```
context Cast
def: withCast(type:String, template:String, params:Sequence(String)):String=
    withCast(type, Template::fill(template, params))
```

```
context Cast
def: withCast(type:String, exp:String):String=
    Template::fill('Ocl.to$1($2)', Template::fill(type, exp))
```

OperationHelper

mapOpName

- bildet Namen von Operationen der OCL Standard Library auf ihre Entsprechungen der OCL-Basisbibliothek ab.

```
context OperationHelper
def: mapOpName(name : String) : String =
    if name = '<>' then 'isNotEqualTo' else
    if name = '=' then 'isEqualTo' else
    if name = 'oclIsNew' then 'isNew' else
    if name = 'oclIsUndefined' then 'isUndefined' else
    if name = '+' then 'add' else
    if name = '-' then 'substract' else
    if name = '*' then 'multiply' else
    if name = '/' then 'divide' else
    if name = '<' then 'isLessThan' nelse
    if name = '>' then 'isGreaterThan' else
    if name = '<=' then 'isLessEqual' else
    if name = '>=' then 'isGreaterEqual' else
        name
    endif endif endif endif endif endif endif endif endif endif endif
```

needsCast

- liefert `true` für genau die Methoden der OCL-Basisbibliothek, die als Rückgabetyp `OclRoot` haben und daher einen Type-Cast benötigen

```
context OperationHelper
def: needsCast(name : String) : String =
   Set{'sum', 'at', 'first', 'last', 'oclAsType', 'any'}
       ->includes(name)
```

Template

fill

- Ersetzt Platzhalter in einem Template durch die übergebenen Argumente. Als Platzhalter werden die Zeichen `'$1'` bis `'$9'` erkannt.
- Bsp.: `Template::fill('$1.$2($1)', Sequence{'x', 'add'})` liefert: `'x.add(x)'`

```
context Template
def: fill(template : String, arguments : Sequence(String)) : String =
-- OCL is not really suitable for scanning strings
```

commaSepList

- Ersetzt den Platzhalter `'$1'` in einem Template sukzessive durch die übergebenen Argumente. Die hierbei entstehenden Zeichenketten werden durch Komma getrennt konkateniert.
- Bsp.: `Template::fill(' "$1" ', Sequence{'A', 'B', 'C'})` liefert: `' "A" , "B" , "C" '`

```
context Template
def: commaSepList(template : String, arguments : Sequence(String)) : String =
    arguments->iterate(arg : String; acc : String = '' |
        let part = Template::fill(template, arg) in
        if arguments->indexOf(arg) < arguments->size() then
           acc.concat(part)
        else
           acc.concat(part).concat(',')
        endif)
```

Zusätzliche Operationen in der OCL Standard Library

Die Transformation benötigt Operationen, die die String-Repräsentation einer ganzen oder reellen Zahl liefern. Aufgrund der allgemeinen Nützlichkeit solcher Operationen, wird eine entsprechende Erweiterung der OCL Standard Library vorausgesetzt:

```
contextInteger
def: toString() : String = -- string representation of self
```

```
contextReal
def: toString() : String = -- string representation of self
```

Literaturverzeichnis

[CWM10] Object Management Group (OMG). *Common Warehouse Metamodel (CWM) Specification*. Version 1.0, Februar 2001.

[Fin99] Frank Finger. *Java-Implementierung der OCL-Basisbibliothek*. Großer Beleg, TU Dresden, 1999.

[Fin00] Frank Finger. *Entwurf und Implementierung eines modularen OCL-Compilers*. Diplomarbeit, TU Dresden, 2000.

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*. Addison-Wesley, Januar 1995.

[JMI10] Java Community Process. *Java™ Metadata Interface(JMI) Specification.* Version 1.0, Juni 2002.

[Kon03] Ansgar Konermann. *Entwurf und prototypische Implementation eines OCL2.0-Parser*. Diplomarbeit, TU Dresden, 2003.

[Lis00] Rüdiger Liskowsky. *Vorlesungsskript Softwareentwicklungswerkzeuge*. Technische Universität Dresden, 2000.

[Loe01] Sten Löcher. *UML/OCL für die Integritätssicherung in Datenbankanwendungen.* Diplomarbeit, TU Dresden, 2001.

[MOF14] Object Management Group (OMG). *Meta Object Facility (MOF) Speci-*

fication. Version 1.4, April 2002.

[MOF2FO] Object Management Group (OMG). *MOF2 Facility and Object Lifecycle Request For Proposal*. März 2003.

[OCL16] Boldsoft, Rational Software Corporation, IONA, Adaptive Ltd. *Response to the UML 2.0 OCL RfP (ad/2000-09-03)*, Version 1.6, Januar 2003.

[OCLRFP] Object Management Group (OMG). *Request For Proposal UML 2.0 OCL. September 2000.*

[QVT] Object Management Group (OMG). *Request for Proposal: MOF 2.0 Query / Views / Transformations.* Oktober 2002.

[Ric01] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints.* Logos-Verlag, 2002.

[UML11] Object Management Group (OMG). *Unified Modeling Language Specification*. Version 1.1, November 1997.

[UML15] Object Management Group (OMG). *Unified Modeling Language Specification*. Version 1.5, September 2002.

[XMI12] Object Management Group (OMG). *XML Metadata Interchange (XMI) Specification*. Version 1.2, January 2002.

[Wie00] Ralf Wiebicke. Utility Support for Checking OCL Business Rules in Java Programs. Diplomarbeit, TU Dresden, 2000.

[WWW1] Object Management Group (OMG). *UML 1.5 Interchange artifacts*. September 2002.

<http://www.omg.org/cgi-bin/doc?ptc/02-09-03>

[WWW2] Object Management Group (OMG). *MOF 1.4 Model package expressed in XMI 1.1 / MOF 1.4*. Oktober 2002.
<http://www.omg.org/cgi-bin/doc?ptc/01-10-05>

[WWW3] Rational Software. *Rational Rose Add-ins*.
<http://www.rational.com/support/downloadcenter/addins/index.jsp>

[WWW4] Distributed Systems Technology Centre. *dMOF 1.1 – An OMG Meta Object Facility Implementation*.
<http://dstc.com/Products/CORBA/MOF/>

[WWW5] Sun Microsystems. *Metadata Repository.*
<http://mdr.netbeans.org/>

[WWW6] Unisys. *JMI Reference Implementation.*
<http://ecommunity.unisys.com>

[WWW7] Novosoft Metadata Framework.
<http://nsuml.sourceforge.net/>

[WWW8] Dresden OCL Toolkit
<http://dresden-ocl.sourceforge.net/>

[WWW9] USE - A UML-based Specification Environment
<http://dustbin.informatik.uni-bremen.de/projects/USE/>

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 30. Juni 2003.