Dresden OCL Toolkit

Metadata Repository und Java Metadata Interface

bearbeitet von:

Mirko Stölzel

s2729561@mail.inf.tu-dresden.de

Technische Universität Dresden

Fakultät Informatik

Institut für Software- und Multimediatechnik

Lehrstuhl für Softwaretechnologie

1. Einleitung

Das Dresden OCL Toolkit [WWW1] wird seit 1999 am Lehrstuhl für Softwaretechnolgie der TU Dresden als Ergebnis der Arbeit von mehreren Beleg-, Diplomstudenten und wissenschaftlichen Mitarbeitern entwickelt. Derzeit wird an der Veröffentlichung der Version 2.0 gearbeitet, mit der es möglich ist OCL-Ausdrücke einzulesen bzw. als String zu übergeben, diese einer Konsistenzprüfung gegenüber dem zugehörigen Modell zu unterziehen und letztendlich den entsprechenden Java-Code zu erzeugen. Das Dresden OCL Toolkit soll dabei nicht als eigenständige Anwendung, sondern eher als eine Art Bibliothek fungieren.

Das Ziel dieser Arbeit ist es zukünftigen Beleg- und Diplomstudenten den Einstieg im Umgang mit dem Dresden OCL Toolkit zu erleichtern.

Der wichtigste Bestandteil des Dresden OCL Toolkits ist das Repository, da dieses alle Modellinformationen, die zur Konsistenzprüfung eines OCL-Ausdrucks erforderlich sind, enthält. Da sämtliche Strukturen des Dresden OCL Toolkits auf dem Repository arbeiten, ist es am besten dort den Einstiegspunkt für eine genauere Betrachtung des Dresden OCL Toolkits zu wählen.

In Kapitel 2 werden aus diesem Grund zunächst die relevanten Technologien des Repositories erläutert. Anschließend wird dann im Kapitel 3 darauf aufbauend das Metamodell des Dresden OCL Toolkits vorgestellt. Nachdem auf diese Weise mit dem Repository die grundlegende Struktur des Dresden OCL Toolkits vermittelt wurde, folgt abschließend im Kapitel 4 eine ausführliche Erläuterung, wie man mit den vorgestellten Technologien ein UML-Modell als Instanz des Metamodells erstellt.

2. Repository-Technologien

Beim Repository des Dresden OCL Toolkits kommen zwei Technologien namens Java Metadata Interface (JMI) [2], und Metadata Repository (MDR) [3] zum Einsatz. Anhand ausführlicher Beispiele werden die beiden Technologien in den folgenden Abschnitten 2.1, 2.2 und 2.3 näher vorgestellt.

2.1. Java Metadata Interfaces

JMI ist eine allgemeine Metadaten-Infrastruktur zum gemeinsamen Metadatenzugriff unter Java. JMI basiert dabei auf der MOF-Spezifikation [4] der OMG und liefert zu MOF-basierten Metamodellen und Modellen eine dynamische Infrastruktur. Im speziellen bedeutet dies, dass JMI eine Reihe von Regeln spezifiziert [2, Kapitel 4], über die man, ausgehend von einem MOF-Metamodell, die entsprechenden Java-Interfaces generiert. Auf diese Weise ist es möglich jeder beliebigen Java-Anwendung den Zugang zu MOF-basierten Metadaten zu ermöglichen.

Ein kurzes Beispiel zu JMI

Gegeben sei das folgende Metamodell zur Vater-Sohn-Beziehung:

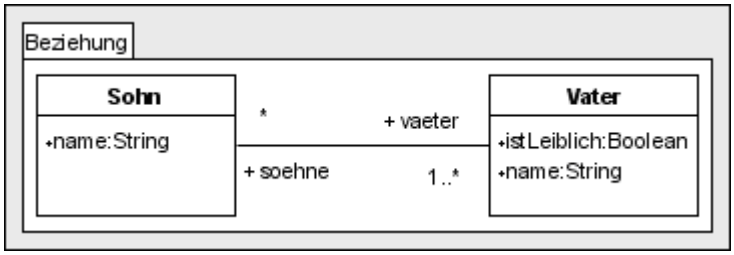

Abbildung 2.1: JMI-Beispiel: Metamodell

Entsprechend des in Abbildung 2.1 dargestellten Metamodells werden über die Regeln der JMI-Spezifikation [2, Kapitel 4] zur Abbildung von MOF nach Java folgende Schnittstellen generiert:

Zur Erzeugung von Instanzen der beiden Klassen `Vater` und `Sohn` werden die beiden Klassen-Interfaces `VaterClass` und `SohnClass`, zu erkennen in Abbildung 2.2 und 2.3, generiert. Diese beiden Klassen-Interfaces enthalten dazu die entsprechenden `create()`−Methoden.

```
public interface VaterClass extends javax.jmi.reflect.RefClass
{
    public Vater createVater();
    public Vater createVater(java.lang.String name,
                             boolean leiblich);
}
```

Abbildung 2.2: JMI-Beispiel: Klassen-Interface `VaterClass`

```
public interface SohnClass extends javax.jmi.reflect.RefClass
{
      public Sohn createSohn();
      public Sohn createSohn(java.lang.String name);
}
```

Abbildung 2.3: JMI-Beispiel: Klassen-Interface `SohnClass`

Nachdem mittels der oben abgebildeten Interfaces Instanzen der Klassen `Vater` und `Sohn` erzeugt werden können, kann über die zwei folgenden Instanzen-Interfaces auf die Attribute der beiden Instanzen zugegriffen werden. Diese stellen dazu die entsprechenden `get()` und `set()`-Methoden zur Verfügung. Wie in Abbildung 2.4 und 2.5 zu erkennen ist, sind dies zur Bearbeitung der Attribute die `getName()`, `setName()`, `getLeiblich()` und `setLeiblich()`-Methoden. Für die Abfrage der Werte der beiden Assoziationsenden stehen die beiden Methoden `getSoehne()` und `getVaeter()` zur Verfügung.

```
public interface Vater extends javax.jmi.reflect.RefObject
{
      public java.lang.String getName();
      public void setName(java.lang.String newValue);
      public java.util.Collection getSoehne();
      public void setIstLeiblich(boolean newValue);
      public boolean getIstLeiblich();

}
```

Abbildung 2.4: JMI-Beispiel: Instanzen-Interface `Vater`

```
public interface Sohn extends javax.jmi.reflect.RefObject
{
      public java.lang.String getName();
      public void setName(java.lang.String newValue);
      public java.util.Collection getVaeter();
}
```

Abbildung 2.5: JMI-Beispiel: Instanzen-Interface `Sohn`

Um die Werte der Assoziationsenden zu Bearbeiten wird zusätzlich noch das Assoziationen-Interface `VaeterSoehne` generiert. Dieses enthält Methoden, um Assoziation zwischen der Klasse `Vater` und `Sohn` zu definieren. Wie aus Abbildung 2.6 ersichtlich, kann man mittels des Assoziationen-Interface `VaeterSoehne` die entsprechenden beteiligten Assoziationsenden definieren, entfernen oder aber überprüfen, ob eine Assoziation mit bestimmten Assoziationsenden existiert.

```
public interface VaeterSoehne
extends javax.jmi.reflect.RefAssociation {
      public boolean exists(Vater vater, Sohn sohn);
      public Vater getVater(Sohn sohn);
      public java.util.Collection getSoehne(Vater vater);
      public boolean add(Vater vater, Sohn sohn);
      public boolean remove(Vater vater, Sohn sohn);
}
```

Abbildung 2.6: JMI-Beispiel: Assoziationen-Interface `VaterSohn`

Als letztes werden noch die Paket-Interfaces erzeugt. Diese enthalten `get()` und `set()`-Methoden, um Referenzen auf die Interfaces der Elemente zu erhalten, die sich im jeweiligen Paket befinden. Für unser Beispiel werden insgesamt zwei Paket-Schnittstellen erzeugt. Die erste Schnittstelle ist für das `Beziehung`-Paket, die in Abbildung 2.7 dargestellt ist. Die zweite Schnittstelle ist für das so genannte `RefOutermostPackage`. Dabei handelt es sich um das oberste Element unseres Metamodells, dessen Name sich aus dem <Metamodellnamen + Package> ergibt. Über dieses Paket-Interface können sämtliche Schnittstellen der Metamodellelemente referenziert werden. Eine Instanz dieses Paketes repräsentiert somit ein Modell des Metamodells. Ein Beispiel der `RefOutermostPackage`-Schnittstelle unseres Metamodells ist in Abbildung 2.8 zu erkennen.

```
public interface BeziehungPackage extends
javax.jmi.reflect.RefPackage
{
      public SohnClass getSohn();
      public VaterClass getVater();
}
```

Abbildung 2.7: JMI-Beispiel: Paket-Interface `BeziehungPackage`

```
public interface FamiliePackage extends
javax.jmi.reflect.RefPackage
{
      public Beziehung getBeziehung();
}
```

Abbildung 2.8: JMI-Beispiel: `RefOutermostPackage`*-Interface*

2.2. Metadata Repository

Das MDR ist eine Entwicklung des NetBeans-Projektes [WWW2], aus dem auch die Entwicklungsumgebung Netbeans stammt. Als erweiterte Implementation der MOF [4], von XMI und von JMI [2] fungiert es als Repository, also als ein Datenspeicher, für MOF-Metadaten. MOF basierte Metamodelle, wie z. B. das UML Metamodell, können mittels bestehender CASE-Tools erstellt, nach XMI exportiert und schließlich ins MDR geladen werden. Auf das MDR kann dann

mittels der zugehörigen JMI-Schnittstellen zugegriffen werden. Diese können dazu vom MDR automatisch generiert und für die Erstellung von Modellen der geladenen Metamodelle genutzt werden.

Ein kurzes Beispiel zu MDR

Man hat im oben aufgezeigten JMI-Beispiel bereits gesehen, wie die entsprechenden JMI-Schnittstellen des Beispiel-Metamodells aussehen. In diesem Abschnitt soll es nun darum gehen, wie man mittels der generierten Schnittstellen auf das Repository zur Laufzeit zugreift, und ein Modell unseres Beispiel-Metamodells (siehe Abb. 2.1) erstellt. Dazu ist in Abbildung 2.9 ein entsprechendes Code-Beispiel abgebildet, in dem ein Modell namens „BspModell“ erzeugt wird. Dieses enthält eine Instanz der Klasse `Sohn` mit dem Namen „Paul“. Unser Beispiel-Metamodell wurde bereits geladen und ist im MDR mit dem Namen `Familie` vorhanden.
Für den Zugriff auf das MDR existiert die Klasse `MDRManager`. Über diese erhält man mittels der Funktion `getDefaultRepository()` eine Instanz der Klasse `MDRRepository`. Diese stellt zum Zugriff auf das Repository verschiedene Methoden zur Verfügung. Mit der Methode `createExtend()` besteht die Möglichkeit, Instanzen von Metamodellelementen im MDR zu erzeugen. Dazu übergibt man dieser Methode den Name der Instanz sowie das Metamodellelement. Wie man im JMI-Beispiel lesen konnte, wird ein Modell eines Metamodells durch eine Instanz des `RefOutermostPackage` des Metamodells repräsentiert. Somit würde die Methode `createExtent()` durch Übergage des `RefOutermostPackage` eines Metamodells ein neues Modell dieses Metamodells im MDR erzeugen. Die Elemente innerhalb des MDR, also auch das `RefOutermostPackage`, können über die Methode `getExtent()` der Klasse `MDRRepository` geladen werden. Dazu muss man lediglich den Namen des Metamodellelements an diese Methode übergeben. Für unser Beispiel würde man, wie in Abbildung 3.9 dargestellt, den Namen `Familie` an die Methode `getExtent()` übergeben und erhält das `RefOutermostPackage`. Dieses kann man nun zusammen mit dem Namen „BspModell“ an die Methode `createExtend()` übergeben. Damit wird nun unser Modell als eine neue Instanz des `RefOutermostPackage` unseres Metamodells mit dem Namen `BspModell` im MDR erzeugt. Über diese Instanz können wir nun mittels der `Beziehung`-Paketschnittstelle eine neue Instanz der Klasse `Sohn` erstellen und dieser über deren `setName()`-Methode den Namen „Udo“ zuweisen.

```
MDRepository rep = MDRManager.getDefault().getDefaultRepository();

//Erstellen und ermitteln des Modells des Metamodells Familie
Repository.createExtent("BspModell", Repository.getExtent(„Familie“);
FamiliePackage model = (FamiliePackage) rep.getExtent("BspModell");

//Ermitteln Paketschnittstelle BeziehungPackage
BeziehungPackage beziehung = model.getBeziehung();

//Erstellung Klasse Sohn und setzen des Namens “Udo”
Sohn sohn = beziehung.getSohn().createSohn();
sohn.setName("Udo");
```

Abbildung 2.9: MDR-Beispiel: MDR-Zugriff

2.3. Benutzerdefinierte JMI Schnittstellen

Im MDR ist es möglich die generierten JMI-Schnittstellen durch Angabe einer benutzerdefinierten Implementierung an seine eigenen Bedürfnisse anzupassen [WWW1]. Dazu muss eine abstrakte Klasse mit dem Namen <Name des Interface + Impl> erstellt werden. Diese Klasse sollte dabei das entsprechende Interface implementieren und den entsprechenden Handler (PackageProxyHandler, ClassProxyHandler, InstanceProxyHandler, AssociationProxyHandler) erweitern. Um nun eine Methode des Interfaces benutzerdefiniert zu implementieren, muss man lediglich die abstrakte Supermethode mit dem Namen super_<Methodenname> definieren und die eigentliche Methode überschreiben. Die abstrakte Supermethode wird zur Laufzeit erzeugt und kann somit auch aus der benutzerdefinierten Methode aufgerufen werden. Sie erfüllt dabei die gleiche Funktion, die sie ausgefüllt hätte, wenn sie nicht überschrieben wäre. Auf diese Weise kann man die eigene Implementierung z. B. nur unter bestimmten Bedingungen ausführen oder aber die Ergebnisse der Supermethode nach den eigenen Bedürfnissen filtern.

Die folgende Abbildung zeigt die benutzerdefinierte Implementierung der `Vater`-Schnittstelle, bei der die Methode `getSoehne()` benutzerdefiniert implementiert wurde. Dadurch erhält man beim Aufruf dieser Methode als Ergebnis lediglich die Instanzen der Klasse `Sohn`, deren Attribut `name` den Wert „Paul“ aufweist. Alle weiteren Instanzen der Klasse `Sohn` sind im Resultat der Methode `getSoehne()` nicht enthalten und können über die so implementierte Methode auch nicht abgefragt werden.

```
public abstract class VaterImpl extends InstanceHandler
                                implements Vater
{
     protected abstract java.util.Collection super_getSoehne();

     public java.util.Collection getSoehne()
     {
          ArrayList soehne = super_getSoehne();
          ArrayList result = new ArrayList();
          Sohn tmpSohn = null;
          Iterator it = soehne.iterator();
          while (it.hasNext())
          {
               tmpSohn = (Sohn)it.next();
               if (tmpSohn.getName().equals("Paul"))
                    result.add(tmpSohn);
          }
          return result;
     }
}
```

Abbildung 2.10: Benutzerdefinierte JMI-Schnittstellen

3. Repository-Metamodell

Die vorherigen Abschnitte haben einen Einblick in die vom Dresden OCL Toolkit verwendeten Technologien gegeben. Man konnte anhand verschiedener Beispiele sehen, wie zu einem Metamodell die JMI-Schnittstellen generiert werden und wie man mittels dieser Schnittstellen auf das Metamodell im MDR zurückgreift. Um jedoch den Modellzugriff innerhalb des Dresden OCL Toolkits zu verstehen, wird in diesem Abschnitt dessen Metamodell näher betrachtet.

Die Metamodellstruktur des Dresden OCL Toolkit ergibt sich aus dessen Aufgabe. Diese besteht darin OCL-Ausdrücke gegenüber einem gegebenen UML-Model oder MOF-Metamodell auf Konsistenz zu überprüfen und den entsprechenden Java-Code zu erzeugen. Dafür wird ein textueller OCL-Ausdruck in ein OCL-Modell entsprechend der abstrakten Syntax der OCL-Spezifikation [5] (OCL-Metamodel) überführt und einer Syntax- sowie einer Typüberprüfung unterzogen. Auf Basis dieses OCL-Modells wird dann der zugehörige Java-Code erzeugt. Für das Dresden OCL Toolkit folgt daraus, dass in dessen MDR zum Umgang mit den aufgeführten Modellen, Metamodelle für UML, MOF und OCL vorhanden sein müssen. Dabei muss das OCL-Metamodell in das Metamodell der UML integriert sein (UMLOCL), da zwischen dem OCL-Metamodell und dem UML-Metamodell Abhängigkeiten bestehen. Diese Abhängigkeiten folgen aus verschiedenen Vererbungsbeziehungen und Assoziationen, die zwischen den Paketen des OCL-Metamodells und hauptsächlich dem `Core`-Paket des UML-Metamodells vorhanden sind. Deutlich zu erkennen ist dies anhand zweier Auszüge aus der OCL-Spezifikation [5] in Abbildung 3.1 und 3.2.

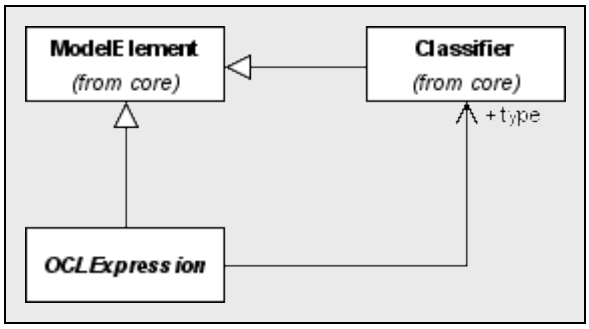

Abbildung 3.1: UML-OCL Abhängigkeiten im Expression-Paket des OCL-Metamodells

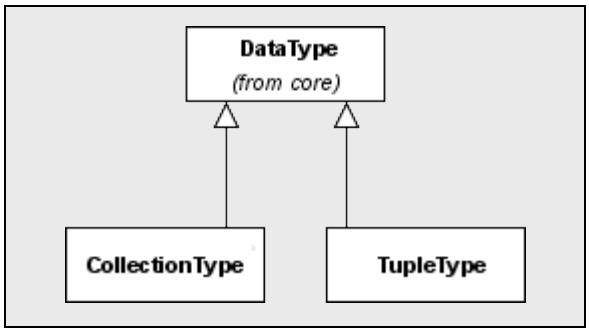

Abbildung 3.2: UML-OCL Abhängigkeiten im Types-Paket des OCL-Metamodells

In Folge der UML 2.0 besitzen UML und MOF einen gemeinsamen Kern. Dadurch könnte das OCL-Metamodell für MOF (MOFOCL) eine Teilmenge von UMLOCL sein. Da das Dresden OCL Toolkit jedoch auf UML 1.5 und MOF 1.4 basiert, muss das OCL-Metamodell auch in MOF integriert werden. Zur genaueren Betrachtung dieser MOF-Integration sei an dieser Stelle auf die Diplomarbeit von Stefan Ocke [1] verwiesen.

In Abbildung 3.3 ist mit dem Metamodell UMLOCL das Ergebnis der Integration von OCL in

UML für die Abhängigkeiten aus Abbildung 3.1 und 3.2 dargestellt. Nicht beteiligte Bestandteile der UML wurden zugunsten der Übersichtlichkeit ausgeblendet und können der UML-Spezifikation [6] entnommen werden.

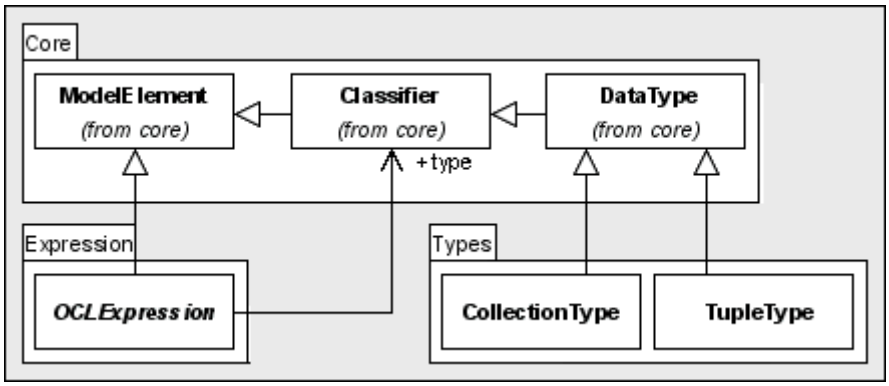

Abbildung 3.3: Auszug von UMLOCL

Auf die Betrachtung von MOFOCL wird an dieser Stelle verzichtet, da der Fokus dieser Arbeit auf UML liegt. Der Grund dafür ist der Umstand, dass der Parser derzeitig lediglich die Konsistenzprüfung von OCL-Ausdrücken für UML-Modelle unterstützt. Jedoch sei darauf hingewiesen, dass sich, bedingt durch bestehende Unterschiede von UML und MOF, das OCL-Metamodell bei der Integration nicht ohne Änderungen auf MOF übertragen lässt [1, Kapitel 5.2.1]. Folglich müsste der Parser des OCL-Toolkits separat sowohl für UML als auch für MOF implementiert werden. Um dies zu umgehen und um dem Parser die Möglichkeit zu geben sowohl mit UMLOCL, als auch MOFOCL-Modellen gleich zu verfahren, muss eine Möglichkeit gefunden werden, beide Metamodelle in einem zu vereinen. Realisiert wird dies durch die Einführung eines gemeinsamen Metamodells namens CommonOCL. Dieses entspricht vom Aufbau her dem OCL-Metamodell und soll als allgemeine Schnittstelle für den Parser fungieren. Dazu müssen die direkten Abhängigkeiten zwischen OCL-Metamodell und dem Metamodell der UML aufgelöst werden. Im CommonOCL existiert dafür ein zusätzliches Paket names `CommonModel`. Dieses enthält Klassen, die an die Stelle der UML-Metaklassen treten, zu denen Abhängigkeiten im OCL-Metamodell bestehen. Durch Herstellung der Vererbungsbeziehungen zu den entsprechenden Klassen von UMLOCL und MOFOCL wird somit erreicht, dass der Parser unabhängig vom zugrunde liegenden Metamodell arbeiten kann. In Abbildung 3.4 sind der Aufbau vom CommonOCL und die Vererbungsbeziehung am Beispiel der Abhängigkeit von OCL-Metamodell zum `Classifier`-Konstrukt des UML-Metamodells dargestellt. Es ist zu erkennen, dass im `CommonModel`-Paket u. a. die Stellvertreterklassen `Classifier` `ModelElement` und `OCLExpression` vorhanden sind, die die Superklassen der zugehörigen Klassen im UMLOCL sind.

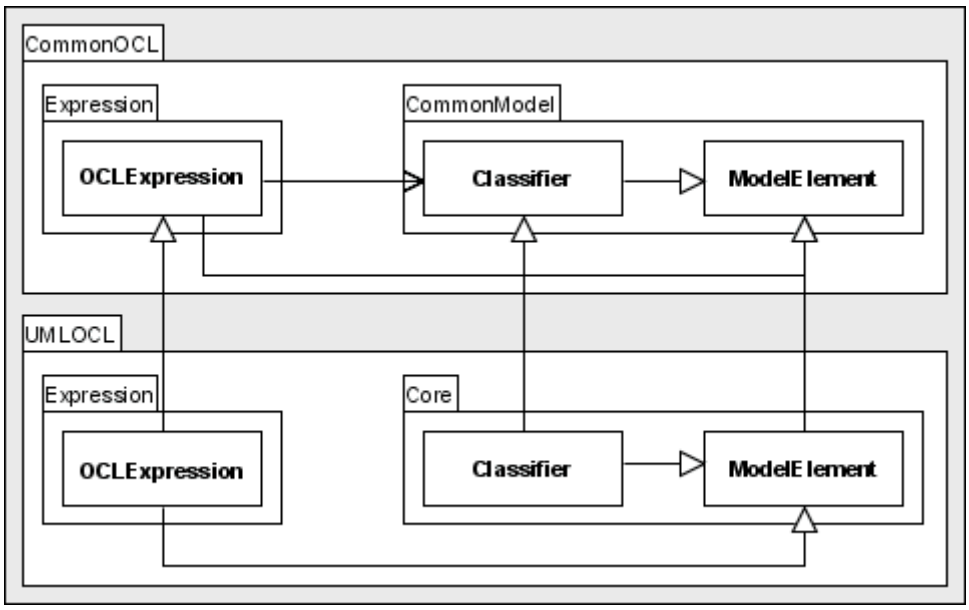

Abbildung 3.4: Auszug CommonOCL

Bei der Betrachtung von Abbildung 3.4 fällt auf, dass die Vererbungsbeziehungen auf Ebene vom UMLOCL erhalten bleiben. Der Grund dafür liegt in der Tatsache, dass sich für die Konstrukte des OCL-Metamodells durch die Vererbungsbeziehungen bestimmte Eigenschaften ergeben. Dies lässt sich besonders gut anhand der beiden Klassen `CollectionType` und `TupleType` des `Types`-Paketes im OCL-Metamodell erläutern. Für diese beiden Klassen gilt durch die Vererbungsbeziehung zur `Classifier`-Klasse des UML-Metamodells, dass sie Methoden besitzen können. Würde diese Vererbungsbeziehung lediglich auf Ebene des CommonOCL bestehen, hätte dies zur Folge, dass diese beiden Klassen keine Methoden besitzen können.

Das letzte noch zu lösende Problem, besteht nun darin, dass der Parser über die Klassen von CommonOCL keinerlei Zugriffmöglichkeit auf Attribute, Assoziation, Operationen usw. des zugrunde liegenden Modells hat. Diese Informationen sind jedoch bei der Umsetzung eines textuellen OCL-Ausdrucks in das entsprechende OCL-Modell von großer Bedeutung. Um trotzdem auf Ebene von CommonOCL auf diese Informationen zu zugreifen, werden für die Klassen des `CommonModel`-Paketes zusätzliche Methoden definiert. Diese Methoden müssen dann auf Ebene vom UMLOCL und MOFOCL in den entsprechenden benutzerdefinierten JMI-Schnittstellen implementiert werden. Eine komplette Übersicht sämtlicher hinzugefügter Methoden findet man in [1, Anhang C].

Abschließend zu diesem Abschnitt sollen noch zwei zusätzliche Hilfsklassen von CommonOCL erwähnt werden. Die Klasse `OCLExpressionFactory` dient dazu Instanzen der Klasse `OCLExpression` zu erzeugen. Sie befindet sich im `Expression`-Paket des CommonOCL und definiert zur Erzeugung der Instanzen des OCL-Metamodells verschiedene Methoden. Bei der

zweiten Klasse handelt es sich um die Klasse `OCLLibrary`. Sie bietet verschiedene Funktionen, um auf vordefinierten primitiven Datentypen der OCL-Standardbibliothek zuzugreifen [5, Kapitel 11]. Notwendig werden diese beiden Klassen durch den Umstand, dass die Erzeugung der Instanzen des OCL-Metamodells und der Zugriff auf die primitiven Datentypen ebenfalls unabhängig vom zugehörigen Modell erfolgen sollen. Die Implementierung dieser Klassen erfolgt deshalb ebenfalls auf Ebene der benutzerdefinierten JMI-Schnittstellen.

4. Umgang mit dem Dresden OCL Toolkit

In diesem abschließenden Abschnitt soll es nun darum gehen, den praktischen Umgang mit dem Dresden OCL Toolkit näher zu erläutern. Dazu wird schrittweise erklärt, wie man das Beispiel aus dem vorangegangenen Kapitel als UML-Modell (nicht als Metamodell) im MDR des Dresden OCL Toolkits erzeugt. Zum besseren Verständnis ist zu jedem Schritt der entsprechende Codeauszug abgebildet.

Schritt 1: Erstellen eines initialen UML-Modells

Die zentale Klasse zur Erzeugung neuer Modelle ist die Klasse `tudresden.ocl20.core.ModelManager`. Dazu muss der Methode `createOclModel()` dieser Klasse lediglich der Name des jeweiligen Metamodells und der Name des neuen Modells übergeben werden. Als Ergebnis dieser Methode erhält man eine neue Instanz des `RefOutermostPackage`. Für unser Beispiel bedeutet dies, dass wir den Metamodellnamen „UML15“ übergeben. Der Name des Metamodells kann auch über die Klasse `MetaModelConst` ermittelt werden, die sich im gleichen Paket wie die Klasse `ModelManager` befindet. Unser neues UML-Modell erhält, wie aus der dem folgenden Codeauszug ersichtlich, den Namen „NeuesModel“. Das `RefOutermostPackage` ist vom Typ `UML15Package`. Der Aufruf der Methode `beginTrans(boolean b)` bedeutet, dass wir den Zugriff auf das Repository für Andere bis zum Aufruf der Methode `endTrans()` (siehe Schritt 6) sperren. Über den Parameter b wird angeben, dass wir schreibend auf das Repository zugreifen wollen.

```
ModelManager mm = ModelManager.getInstance();
mm.beginTrans(true);
model = (Uml15Package) mm.createOclModel(MetaModelConst.UML15,
                                         "NeuesModel");
...
```

Schritt 2: Erzeugen der TopPackage-Instanz (optional)

Das TopPackage ist eine Instanz der UML-Metaklasse `Model` und ist das oberste Element unseres UML-Modells. Die Erzeugung der TopPackage-Instanz ist optional, da das TopPackage vom Dresden OCL Toolkit bei Nicht-Existenz automatisch erzeugt wird.

Wie im folgenden Codeauszug zu erkennen ist, muss zur Erzeugung des TopPackage zunächst das entsprechende Klassen-Interface (`ModelClass`) über die Instanz des `RefOutermostPackage` herausgesucht werden. Über die `createModel()`-Methode des Klassen-Interfaces kann dann eine Instanz der UML-Metaklasse `Model` als Repräsentant des TopPackage erzeugt werden.

```
...
Model topPackage = null;
topPackage = model.
             getModelManagement().//Paket-Interface ModelManagement
             getModel().          //Klassen-Interface ModelClass
             createModel();       //Erzeugung der TopPackage-Instanz
topPackage.setNameA("topPackage");//Setzen des Namens
...
```

Schritt 3: Erzeugen des Beziehungen-Paketes

Zur Erzeugung des Paketes „Beziehungen“ geht man analog zum Schritt 2 vor. Man sucht, wie im folgenden Codeauszug abgebildet, zunächst über die Instanz des `RefOutermostPackage` das Klassen-Interface `PackageClass` heraus. Über die `createPackage()`-Methode kann dann eine Instanz der UML-Metaklasse `Package` erzeugt werden. Diesem kann dann der Name „Beziehung“ und das TopPackage als Namensraum zugewiesen werden.

```
...
Package package = null;
package = model.
          getModelManagement().  //Paket-Interface ModelManagement
          getPackage().          //Klassen-Interface PackageClass
          createPackage();       //Erzeugung der Package-Instanz
package.setNameA("Beziehung");   //Setzen des Namens
package.setNamespace(topPackage);//Setzen des Namenraums
...
```

Schritt 4: Erzeugen der Klassen Sohn und Vater

Anhand des folgenden Codeauszuges ist zu erkennen, dass man im Vergleich zum vorangegangenen Schritt lediglich ein anderes Klassen-Interface (`UMLClassClass`) benutzt. Sämtliche weiteren Schritte sind identisch.

```
...
UMLClass vater = null;
vater = model.
        getCore().             //Paket-Interface Core
        getUmlClass().         //Klassen-Interface UmlClassClass
        createUmlClass();      //Erzeugung der UmlClass-Instanz
vater.setNameA("Vater");       //Setzen des Namens
vater.setNamespace(topPackage);//Setzen des Namenraums
...                            //analog mit Klasse Sohn
...
```

Schritt 5: Erzeugen des Attributes

Für das Attribut `istLeiblich` nutzen wir abermals die entsprechende Klassen-Schnittstelle. Dem so erzeugten Attribut weisen wir, wie aus dem folgenden Codeauszug ersichtlich, den Namen „istLeiblich“ und als Besitzer die eben erzeugte Klasse `Vater` zu. Außerdem erzeugen wir einen Datentyp namens „boolean“, der über die Methode `setType()` unserem Attribut als Typ übergeben wird.

```
...
Datatype boolean = null;
boolean = model.
          getCore().                //Paket-Interface Core
          getDatatype().            //Klassen-Interface DatatypeClass
          createDatatype();         //Erzeugung der Datatype-Instanz
boolean.setNameA(“boolean”);        //Setzen des Namens

Attribute istLeiblich = null;
istLeiblich = model.
              getCore().            //Paket-Interface Core
              getAttribute().       //Klassen-Interface AttributeClass
              createAttribute();    //Erzeugung der Attribute-Instanz
istLeiblich.setNameA("istLeiblich");//Setzen des Namens
istLeiblich.setType(boolean);       //Setzen des Datentyps
istLeiblich.setOwner(vater);        //Setzen des Besitzers
...
```

Schritt 6: Erzeugen der Association

Zur Erzeugung der Assoziation zwischen der Klasse `Vater` und `Sohn` sind zwei Teilschritte erforderlich. Als erstes müssen zwei Assoziationsenden erzeugt und diesen die beiden Klassen `Vater` und `Sohn` zugewiesen werden. Dies erfolgt, wie im folgenden Codeauszug zu erkennen, über das Klassen-Interface `AssociationEnd` und über das Assoziationen-Interface `AParticipantAssociation`.

```
...
AssociationEnd aeSohn = null;
AssociationEnd aeVater = null;
aeSohn = model.
         getCore().                  //Paket-Interface Core
         getAssociationEnd().        //Klassen-Interface Assoc.EndClass
         createAssociationEnd();     //Erzeugung der Assoc.End-Instanz
aeVater = model.
          getCore().                 //Paket-Interface Core
          getAssociationEnd().       //Klassen-Interface Assoc.EndClass
          createAssociationEnd();    //Erzeugung der Assoc.End-Instanz

AParticipantAssociation pCon = null;
pcon = model.
       getCore().                    //Paket-Interface Core
       getAParticipantAssociation();//Assoz.-Interface
                                     //AParticipantAssociation

con.add(aeVater, vater);             //Hinzufügen Assoz.enden, die
con.add(aeSohn, sohn);               //Klassen Vater und Sohn
...
```

Der zweite Teilschritt, der erforderlich ist, ist die Erzeugung der Assoziation und die Zuweisung der erstellten Assoziationsenden. Dazu nutzen wir das Klassen-Interface `Association` und das Assoziationen-Interface `AAssociationConnection`. Zu erkennen ist dieser Schritt in dem folgenden Codeauszug. Auf die Erstellung und Zuweisung der zusätzlichen Attribute wie Multiplizitäten, Navigation usw. wurde an dieser Stelle verzichtet, da die dafür notwendigen Schritte aus den vorangegangenen Schilderungen deutlich geworden sein sollten.

```
...
Association a = model.
                     getCore().
                          getAssociation().
                               createAssociation();
AAssociationConnetion aCon = model.
                               getCore().
                                    getAAssociationConnection();
con.add(a, aeVater);
con.add(a, aeSohn);
... // Multiplizitäten, Navigation usw.

mm.endTrans(false);
```

Abschließend muss die im Schritt 1 durchgeführte Sperrung des Repositories aufgehoben werden. Dazu wird die Methode `endTrans(boolean b)` aufgerufen. Der Parameter `b` gibt dabei an, ob wir einen Rollback durchführen oder aber unser Modell bzw. die Änderungen im MDR festschreiben möchten.

5. Zusammenfassung

Das Ziel dieser Arbeit war es einen leichteren Einstieg im Umgang mit dem Dresden OCL Toolkit zu ermöglichen. Dazu wurden im ersten Abschnitt dieser Arbeit die grundlegenden Technologien, das Metadata Repository und Java Metadata Interface (JMI), vorgestellt. Im zweiten Abschnitt wurde dann der Aufbau des Metamodells des Dresden OCL Toolkits vorgestellt.

Aufbauend auf diese beiden Abschnitte konnte dann im letzten Abschnitt schrittweise erläutert werden, wie man mittels JMI ein UML-Modell im MDR des Dresden OCL Toolkits erstellt. Unterstützt wurde dies durch einige Codeauszüge.

Literaturverzeichnis

[1] Stefan Ocke, *Entwurf und Implementation eines metamodellbasierten OCL-Compilers,* Diplomarbeit, TU-Dresden/Fakultät Informatik, 2003

[2] Java Community Process, Java *Metadata Interface (JMI) Specification,* JSR 040, Version 1.0, Juni 2002

[3] Martin Matula, *Netbeans Metadata Repository,* http://mdr.netbeans.org/MDR-whitepaper.pdf, 03.03.2003

[4] Object Management Group (OMG). *Meta Object Facility (MOF) Specification,* OMG Dokument formal/02-04-03, Version 1.4, April 2002

[5] Object Management Group (OMG), *UML 2.0 OCL Specification*, OMG Dokument ptc/03-14-10, Version 2.0, Oktober 2003

[6] Object Management Group (OMG), *Unified Modeling Language Specification,* OMG Dokument formal/03-03-01, Version 1.5, März 2003

[WWW1] TU-Dresden, Institut für Software- und Multimediatechnik, Dresden OCL Toolkit-Homepage, 29.10.2005, <http://dresden-ocl.sourceforge .net>

[WWW2] Sun Microsystems, Benutzerdefinierte Java Metadata Interfaces, 29.10.2005, <http://mdr.netbeans.org/custom_implementations.html>