Konzeption und Implementierung einer regelbasierten Transformationskomponente für das Eclipse Modeling Framework

Enrico Biermann

Günter Kuhns

Diplomarbeit

an der Fakultät für Elektrotechnik und Informatik
der Technischen Universität Berlin
Gutachter: Prof. Dr. H. Ehrig und Dr. G. Taentzer

8. Mai 2006

Die selbständige und eigenhändige Anfertigung
der folgenden Kapitel, versichere ich, Enrico Biermann, an Eides statt:

Kapitel 2 - Grundlagen
Kapitel 3 - Anforderungen
Kapitel 4 - Transformation
Kapitel 8 - Zusammenfassung

Berlin, den 8. Mai 2006

Unterschrift

Die selbständige und eigenhändige Anfertigung der folgenden Kapitel, versichere ich, Günter Kuhns, an Eides statt:

Kapitel 1 - Einleitung
Kapitel 5 - Interpreter
Kapitel 6 - Compiler
Kapitel 7 - Beispiele

Berlin, den 8. Mai 2006

Unterschrift

Danksagung - Enrico Biermann

Mein Dank gilt den Betreuern Frau Dr. Ing.Gabi Taentzer und Herrn Dipl.-Inf. Karsten Ehrig für die konstruktive Unterstützung und Kritik bei der Erstellung der Arbeit.
Darüberhinaus möchte ich mich bei Frau Dipl.-Inf. (FH) Olga Runge und Herrn Dipl.-Inf. Eduard Weiss für ihre Hinweise zur Anbindung von AGG und Eclipse bedanken.
Meiner Familie und meinem Mitbewohner Ramon Wegner, möchte ich für ihre Geduld und moralische Unterstützung danken.

Danksagung - Günter Kuhns

Auch ich möchte mich bei den Betreuern Frau Dr. Ing. Gabi Taentzer und Herrn Dipl.-Inf. Karsten Ehrig dafür bedanken, dass sie sich die Zeit genommen haben, diese Arbeit durch Anregungen und konstruktive Kritik zu verbessern.
Weiterhin danke ich Frau Dipl.-Inf. (FH) Olga Runge für Ratschläge im Zusammenhang mit AGG sowie Herrn Dipl.-Inf. Eduard Weiss für die Entwicklung des Regeleditors, Tips in Bezug auf Eclipse und einige Korrekturvorschläge zu dieser Arbeit. Herrn Dipl.-Inf. Stefan Hänsgen gilt mein Dank für eine kurze Einführung in JET.
Meiner Familie danke ich dafür, dass sie mich zu dem Menschen gemacht haben, der ich bin, sowie für alle Unterstützung, die sie mir zukommen liessen.
Dem Team des icafés möchte ich für Momente der Ruhe und Entspannung danken - darüber hinaus Annett Rettke für moralische Unterstützung und einige Korrekturen, sowie Maod Leconte für eine kurze Einführung in die Kategorientheorie.
Mein Dank gilt: William Gibson, Neal Stephenson, Bruce Sterling und Tad Williams - für interessante und inspirierende Zukunftsvisionen,
Colin Low dafür, dass er altes Wissen durch eine zeitgemässe neue Erkärung wiederbelebt hat.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Ziel dieser Diplomarbeit ist es, eine Softwarekomponente zur Modelltransformation für das Eclipse Modeling Framework (EMF) zu entwickeln und diese zu implementieren. Eine Transformation soll dabei durch eine Menge von Regeln beschrieben werden, die grafisch erstellt und editiert werden können. Dadurch sind Änderungen an einer Modellinstanz leichter nachzuvollziehen als bei textuellen Regeln. Da EMF bereits eine graphische Darstellung der erstellten Modelle bietet, begeben wir uns mit graphischen Regeln zu deren Veränderung auf dieselbe Ebene.
Bei der Form der Regeln orientieren wir uns stark am Graphtransformationsansatz. Regeln sollen nicht nur einzeln, sondern auch als Regelmenge so oft angewendet werden, bis keine Regel mehr ausgeführt werden kann. Auf diese Weise kann eine komplexe Änderung durch eine Reihe von einfachen Regeln beschrieben und durchgeführt werden.
Wir haben schliesslich zwei Ansätze implementiert, die wir im folgenden Interpreter und Compiler nennen werden. Beim Interpreter wird eine Regel in AGG - einem Tool zur Graphtransformation - auf einen Graphen, welcher der Instanz entspricht, angewendet. Anschliessend werden die Veränderungen an diesem Graphen ausgewertet und auf die Instanz übertragen. Der Compiler erzeugt zu jeder Regel Java-Code, mit dem diese angewendet werden kann. Auf diese Weise können AGG-Regeln für ein EMF-Modell durch Regelklassen in Java ersetzt werden, wie es auch beim Tiger-Projekt geschehen soll.

Unsere Transformationskomponente führt eine endogene Modelltransformation durch - das heisst Quell- und Ziel-Modell sind identisch. Die Semantik von EMF-Modellen wird soweit berücksichtigt, dass Teile einer Regel, die eine EMF-Instanz inkonsistent machen, nicht ausgeführt und eine Instanz nach einer Regelanwendung wieder konsistent gemacht wird.
Alle Objekte, die durch eine Regel nicht verändert werden, bleiben bestehen, sodass Referenzen von aussen auf eine Instanz auch nach einer Anwendung noch funktionieren. Dadurch kann man zum Beispiel in einem Editor Commands durch die Ausführung einer Regel ersetzen.

In unserer Diplomarbeit wollen wir zunächst auf die theoretischen und technischen Grundlagen eingehen, die unserer Arbeit zugrunde liegen, sowie andere Tools vorstellen, die unserem ähnlich sind, um sie mit diesem zu vergleichen. Dann gehen wir auf die Anforderungen ein, die wir uns zu Beginn der Arbeit gesetzt haben und beschreiben unser Modell mit dem wir Regeln formulieren. Hierbei erklären wir Designentscheidungen bei der Entwicklung des Modells, wie man Regeln entweder in einem Regeleditor oder in AGG editiert und schliesslich, welche Veränderungen diese bei der Ausführung verursachen. Die beiden Ansätze - Interpreter und Compiler - sind Themen der beiden folgenden Kapitel, wobei wir jeweils kurz zusammengefasst deren Arbeitsweise beschreiben, auf die einzelnen Schritte näher eingehen, und schliesslich einzelne Komponenten bzw. Klassen und deren Funktionalität genauer vorstellen. Beim Compiler muss dabei noch zwischen der Codeerzeugung und der Anwendung des generierten Codes unterschieden werden. In zwei Anwendungsbeispielen wollen wir anschliessend die Verwendung beider Ansätze demonstrieren und zum Schluss in der Zusammenfassung einen kurzen Rück- und einen weiteren Ausblick geben, wie diese Arbeit fortgesetzt und weiterentwickelt werden kann.

Kapitel 2

Grundlagen

2.1 Theoretische Grundlagen

2.1.1 Graphtransformationen

Graphtransformationen [11] sind in den 70er Jahren entstanden als Generalisierung von Text und Baumgrammatiken. Heute nehmen Graphgrammatiken eine wichtige Rolle in der theoretischen Informatik, sowie in der Softwareentwicklung und Modellierung verteilter und konkurrierender Systeme ein.
Die Grundidee von Graphgrammatiken ist dabei die regelbasierte Veränderung von Graphen, wobei jede Anwendung einer Regel einen Transformationsschritt darstellt. Aus dieser Eigenschaft heraus lassen sich mit Graphgrammatiken Graphsprachen definieren ähnlich der Beschreibung textueller Sprachen durch Chomsky-Grammatiken. Darüberhinaus kann man mit Graphen den Zustand eines Systems modellieren und mit Hilfe von Graphtransformationen kann man Änderungen dieses Systems beschreiben.

Graphen

Es gibt eine Vielzahl von verschiedene Arten von Graphen, allen gemein ist, dass sie ein Konstrukt aus Knoten und Kanten sind, wobei Knoten über Kanten miteinander verbunden sein können. Einige Graphtypen sind beispielsweise

- einfache Graphen die nur aus Knoten und Kanten bestehen

- getypte Graphen bei denen man Knoten und Kanten einen bestimmten Typ zuweisen kann um sie so zu unterscheiden

- attributierte Graphen, wo Knoten und Kanten Attribute enthalten können

Für unsere Diplomarbeit betrachten wir getypte attributierte Graphen, da diese Klassendiagrammen am nächsten kommen. Um das Aussehen und die Eigenschaften eines getypten attributierten Graphen besser zu verstehen, betrachten wir dies am Beispiel von Petrinetzen.
Petrinetze sind ein Modellierungssprache zur Beschreibung prozessorientierter Systeme. Sie bestehen aus Stellen und Transitionen, die über gerichtete Kanten verbunden sind. Dabei können jeweils nur Stellen mit Transitionen oder Transitionen mit Stellen verbunden werden, nicht aber beispielsweise Stellen mit Stellen.
Weiterhin besitzen Petrinetze eine ausführbare Semantik. Stellen können sogenannte Token enthalten. Falls alle Stellen im Vorbereich einer bestimmten Transition mit Token belegt sind, wird die Transition als aktiv bezeichnet und kann Schalten, d.h. dass alle Token von den Stellen im Vorbereich entfernt werden und alle Stellen im Nachbereich der Transition mit einem Token gefüllt werden.
Ein möglicher Typgraph für Petrinetze könnte z.B. wie in Abbildung 2.1 aussehen. Der Typgraph definiert die verschiedenen Knoten- und Kantentypen sowie ihre jeweiligen Attribute. Hervorzuheben ist, dass Kanten in Petrinetzen in diesem Typgraphen als Knoten definiert wurden.
Zu diesem Typgraphen können jetzt Graphen definiert werden. Die Graphen dürfen dabei nur die Typen verwenden, die im Typgraphen definiert wurden. Ein möglicher über dem Typgraphen getypter Graph ist in Abbildung 2.2 zu sehen. In der konkreten Darstellung von Petrinetzen kann man den Graphen in Abbildung 2.3 sehen. In ähnlicher Weise lassen sich auch andere Petrinetze als Graphen definieren, die über unserem Typgraphen getypt sind.

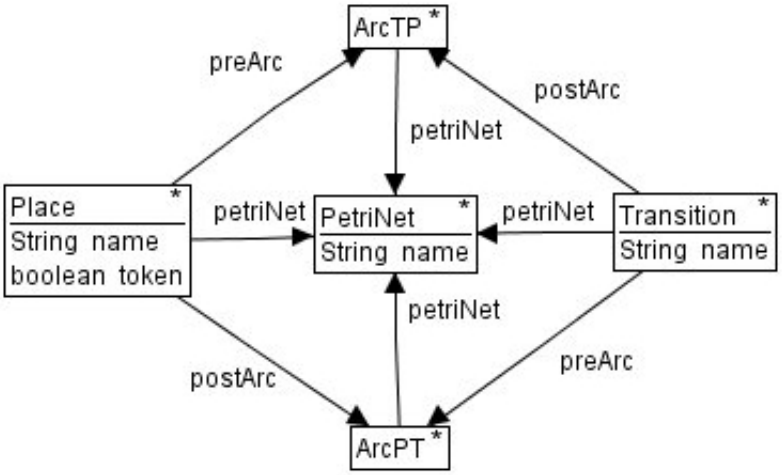

Abbildung 2.1: Petrinetz Typgraph

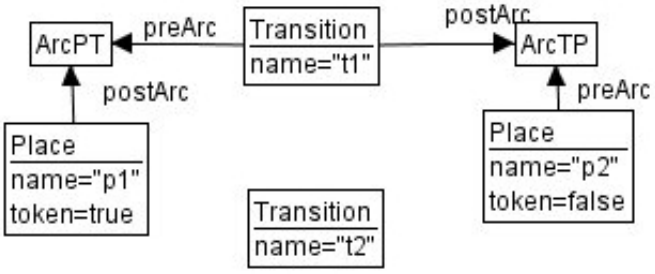

Abbildung 2.2: Graphdarstellung des Petrinetzes

Regeln

Mit Graphen hat man also die Möglichkeit bestimmte Systeme zu modellieren. Allerdings sind Graphen statische Konstrukte. Um Graphen zu verändern greift man deshalb auf Graphgrammatiken zurück.
Eine Regel p einer solchen Graphgrammatik, besteht aus mindestens zwei Graphen (L,R), welche linke und rechte Regelseite genannt werden. Beide Graphen sind über einen Morphismus miteinander verbunden.
Neben den beiden Graphen L und R können noch eine beliebige Menge an zusätzlichen NAC Graphen definiert werden. Diese sind ebenfalls über einen Morphismus mit L verbunden und stellen negative Anwendungsbedingungen

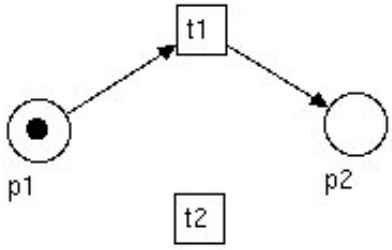

Abbildung 2.3: Petrinetz in konkreter Darstellung

dar, die dazu verwendet werden können die gefunden Matches einzuschränken. Eine Beispielregel für unser Petrinetzbeispiel findet sich in Abbildung 2.4.

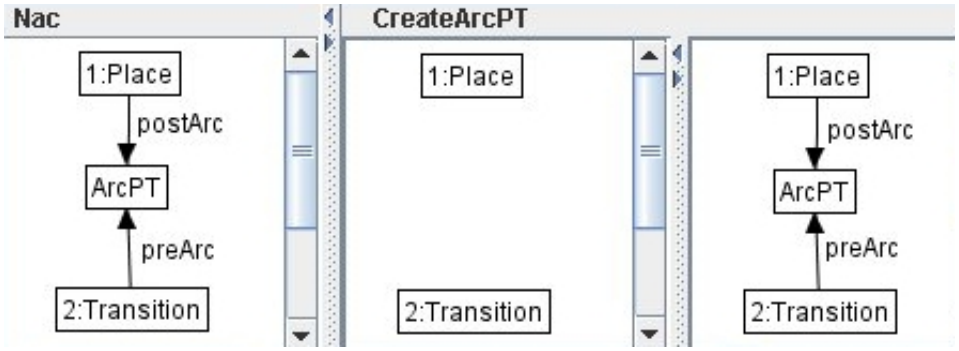

Abbildung 2.4: CreateArcPT Regel

Diese Regel besteht aus einer linken und rechten Seite sowie einer NAC. Gleiche Zahlen in verschiedenen Graphen besagen, dass der Morphismus zwischen den Graphen diese Objekte jeweils aufeinander abbildet. Falls man diese Regel auf einen Graphen anwendet, wird zwischen einer Stelle und einer Transition eine neue Kante eingefügt, falls noch keine Kante vorhanden war. Diese Regel kann beispielsweise Teil einer Grammatik sein, die Petrinetze generiert. In einer solchen Grammatik bräuchte man Regeln, die alle jeweiligen Komponenten eines Petrinetzes erstellen, z.B. eine CreatePlace und eine CreateTransition Regel.
Eine andere denkbare Grammatik könnte sich mit der Semantik von Petrinetzen, genauer gesagt dem Schaltverhalten von Transitionen,

beschäftigen. Beim Schalten einer Transition werden alle Token im Vorbereich der Transition gelöscht, und auf allen Stellen im Nachbereich der Transition wird ein Token hinzugefügt.

Match

Nach der Definition einer Regel, kann man diese nun auf einen bestimmten Graphen anwenden. Dazu betrachten wir als Arbeitsgraphen (der Graph auf den wir Regeln anwenden wollen) den Graphen aus Abbildung 2.2 und die CreateArcPT Regel aus Abbildung 2.4.
Die Regel erwartet mindestens eine Stelle und eine Transition. Im Arbeitsgraphen haben wir die beiden Stellen p1 und p2, sowie die Transitionen t1 und t2. Ein möglicher Match, also ein Morphismus zwischen der linken Regelseite und dem Graphen, könnte zum Beispiel so aussehen, dass die Stelle der Regel auf p1 und die Transition auf t2 abgebildet wird, siehe Abbildung 2.5.

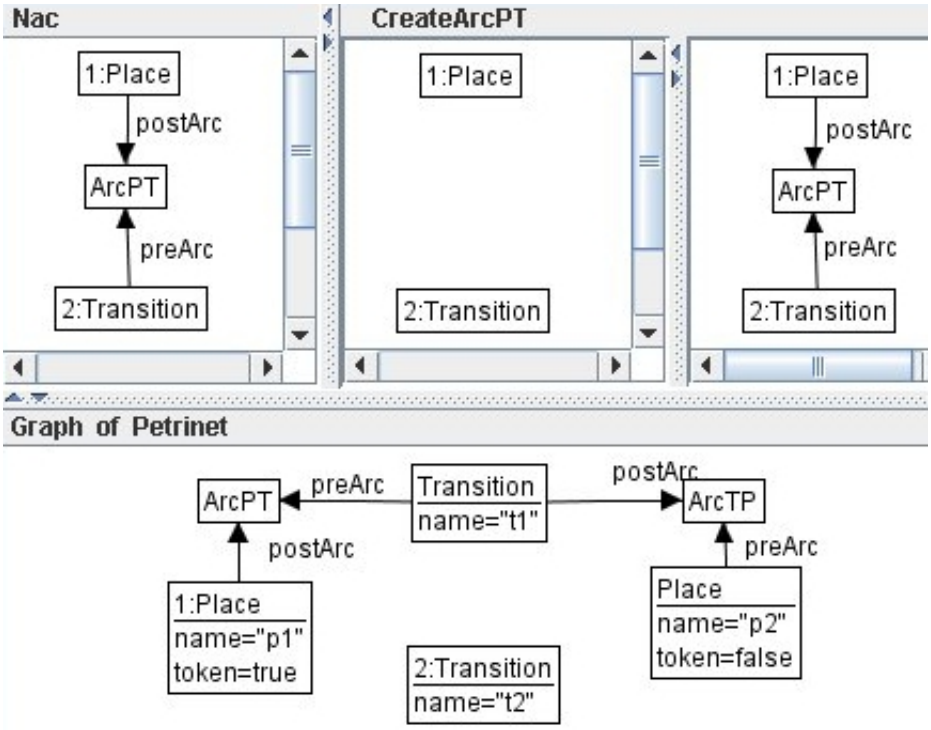

Abbildung 2.5: Match von CreateArcPT auf den Arbeitsgraph

Ein Morphismus auf p1 und t1 ist zum Beispiel nicht möglich, weil die NAC einen bereits vorhanden ArcPT Knoten zwischen p1 und t1 verbietet. Generell sollte man sich darüber im klaren sein, dass es mehrere mögliche Matches zwischen einer Regel und dem Arbeitsgraphen geben kann, d.h. die Anwendung der gleichen Regel auf den gleichen Graphen, kann zu einem anderen Ergebnis führen. Ebenfalls zu beachten ist, dass eine erzeugende Grammatik möglicherweise nicht terminiert, da auf den neuen Objekten ständig neue Matches gefunden werden können.

Regelanwendung

Falls ein Match gefunden wurde, der die NACs erfüllt, kann die Regel angewendet werden. Bei der Anwendung der Regel wird anschaulich die linke Regelseite durch die rechte Regelseite ersetzt. Im Fall von CreateArcPT bedeutet das, dass zwischen p1 und t2 eine ArcPT Kante eingefügt wird, wie in Abbildung 2.6 bzw. in konkreter Darstellung in Abbildung 2.7 zu sehen ist.

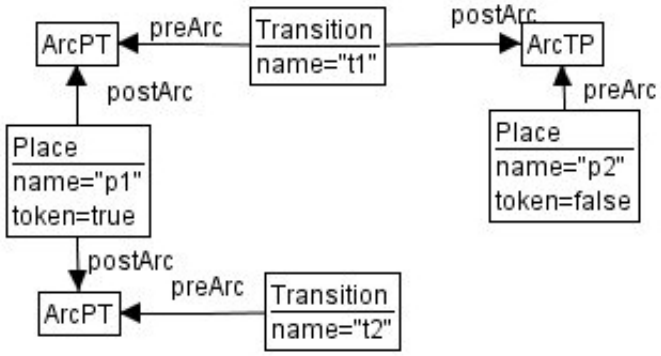

Abbildung 2.6: Arbeitsgraph nach Anwendung von CreateArcPT

Mit diesen grundlegenden Konzepten von Graphen und Graphtransformationen lassen sich verschiedenartige Änderungen beschreiben, sei es die konstruktive Definition einer Sprache oder die Beschreibung des Verhaltens eines Systems.

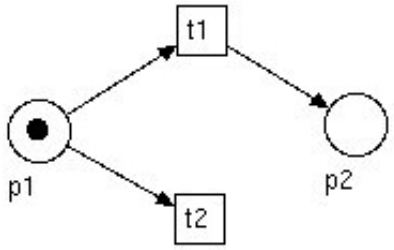

Abbildung 2.7: Petrinetz nach Anwendung von CreateArcPT

2.1.2 Vererbung

Ein gebräuchliches Konzept bei der Modellierung von Systemen ist die Vererbung, welche auch für Graphen defniert ist.
Wenn man unser Petrinetzbeispiel näher betrachtet, fällt zum Beispiel auf, dass *Place*, *Transition* und *PetriNet* jeweils das gleiche Attribut *String name* haben. Man könnte daher einen neuen Knotentyp *NamedElement* mit diesem Attribut einfügen, und die drei anderen Typen davon erben lassen, siehe Abbildung 2.8.

Abgesehen davon, dass man auf diese Weise sich wiederholende Strukturen vermeiden kann, bietet die Vererbung auch noch einen entscheidenden Vorteil für die Erstellung von Regeln.
Falls Regeln über einem allgemeinen Typ definiert wurden, kann die Regel dennoch auf speziellere Typen angewendet werden. Für das *NamedElement* lässt sich zum Beispiel eine Rename-Regel definieren, die einem *NamedElement* einen neuen Namen gibt. Diese eine Regel, lässt sich aber auf Places, Transition und PetriNets anwenden. Ohne Vererbung hätte man für die gleiche Funktionalität drei Regeln benötigt.

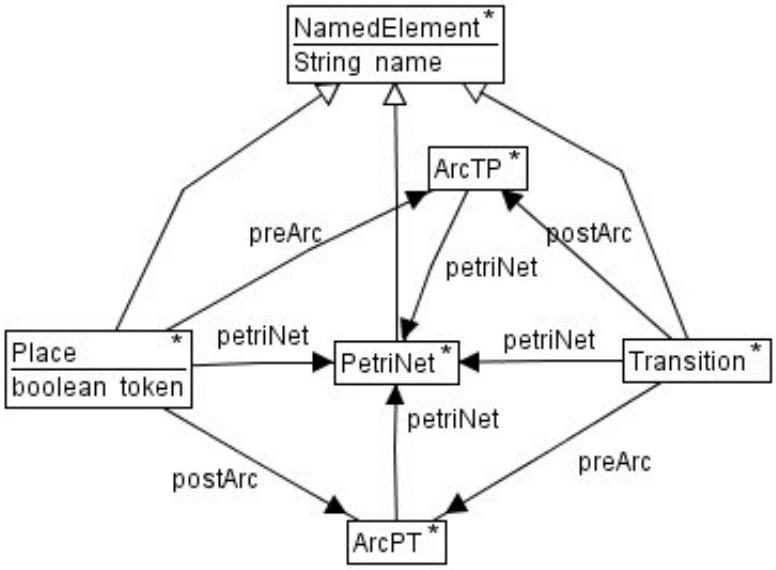

Abbildung 2.8: Typgraph mit Vererbung

2.2 Technische Grundlagen

2.2.1 Eclipse

Eclipse [2] ist zur Zeit eine der wichtigsten Entwicklungsumgebungen für verschiedene Progammiersprachen (Java, C++ und andere). Seit der Freigabe, 2001 durch IBM, wird es als Open-Source-Projekt unter Java weiterentwickelt. Es ist für die verbreitetsten Betriebssystem-Plattformen verfügbar und bietet verschiedene Perspektiven, welche bei der Software-Entwicklung sinnvoll sind. So zum Beispiel einen Editor für Java (mit Syntax-Highlighting und Anzeige von Syntax-Fehlern), eine Debug-Ansicht und eine Schnittstelle für CVS-Repositories.
Eclipse ist modular aufgebaut und bildet eine Plattform, die durch Plugins wie EMF, GEF oder selbst entwickelte Plugins erweitert werden kann. Es verwaltet dabei alle integrierten Plugins, regelt Abhängigkeiten zwischen ihnen und Interaktionen der Plugins untereinander.
Aus der normalen Entwicklungsumgebung kann eine neue Eclipse-Instanz als Laufzeitumgebung (Runtime-Workspace) gestartet werden, in welcher dann Plugins ausgeführt werden können.

2.2.2 EMF

Das Eclipse Modeling Framework (EMF) [6, 7] ist ein Java-Framework für Eclipse, mit dem man Modelle entwickeln und aus diesen Code generieren kann. Es bietet eine Schnittstelle zwischen den Modellierungsformaten UML und XML sowie einer Implementierung in Java. Jedes dieser Formate kann mit Hilfe von EMF in die jeweils anderen umgewandelt werden.

Modelle in EMF basieren auf dem Ecore-Metamodell, von dem wir einen Ausschnitt in Abbildung 2.10 zeigen. Das oberste Element eines Modells ist immer ein *EPackage*, das wiederum in mehrere Subpackages unterteilt sein kann. Klassen werden durch *EClasses*, primitive Datentypen durch

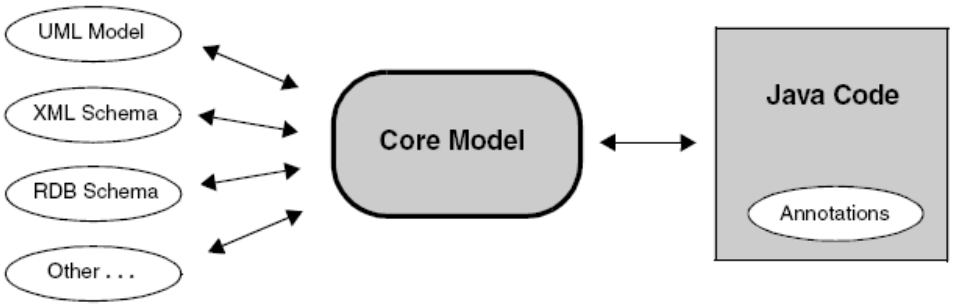

Abbildung 2.9: EMF als Interface zwischen verschiedenen Modellierungsstandards

EDataTypes repräsentiert. Beide sind unter dem abstrakten Typ $EClassifier$ zusammengefasst.
Attribute und Referenzen einer Klasse werden durch *EAttribute* bzw. $EReference$ dargestellt, die beide wiederum von dem abstrakten Typ $EStucturalFeature$ erben. Methoden einer Klasse und deren Parameter werden im Ecore-Metamodell durch *EOperation* bzw $EParameter$ repräsentiert. Zusammen mit $EStructuralFeature$ haben sie die gemeinsame Oberklasse *ETypedElement* - die Modell-Elemente die einen Typ haben, dargestellt durch die Referenz *eType*() auf einen $EClassifier$ - bei *EAttribute* ist dies immer ein *EDataType*, bei $EReference$ eine *EClass*, sonst beliebig.
Alle bisher genannten Elemente eines Modells haben einen Namen und sind daher Kinder der abstrakten Klasse $ENamedElement$. Ausser diesen gibt es noch Anmerkungen zu einem $EModelElement$ - sogenannte *EAnnotattions*, welche wiederum selbst ebenfalls als Obertyp $EModelElement$ haben. Die Oberklasse aller Objekte eines Ecore-Modells ist $EObject$.
Alle oben genannten Klassen stellen Teile eines Modells bzw dessen Gesamtheit ($EPackage$) dar - daneben gibt es zu jedem $EPackage$ noch jeweils eine eigene $EFactory$, die alle (nicht abstrakten) $EClasses$ eines Modells erzeugen kann.

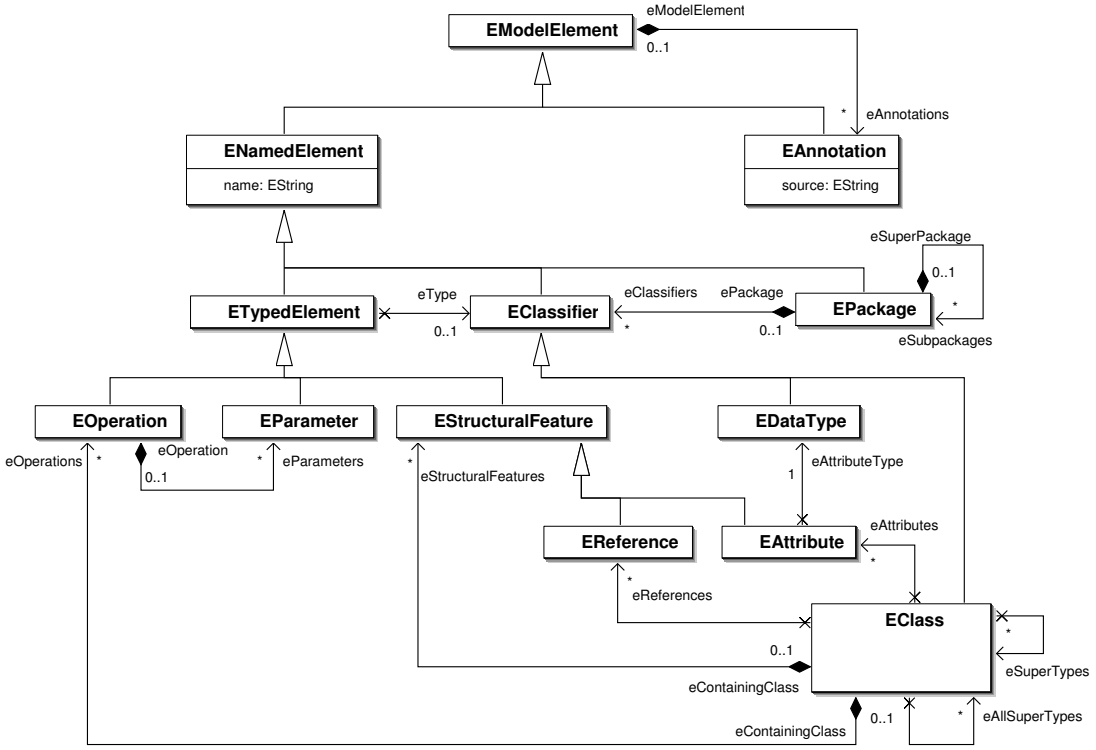

Abbildung 2.10: Ausschnitt des Ecore-Metamodells als EMF-Modell

Beispiel eines EMF-Modells

Abbildung 2.11 zeigt das EMF-Modell eines Petrinetzes. Die *EClasses* sind als Rechtecke dargestellt, welche den Namen der *EClass* und deren *EAttributes* enthalten - *Place*, *Transition* und *PetriNet* haben jeweils einen Namen. *Places* können entweder ein Token enthalten oder leer sein - dies wir durch das *EAttribute* 'token' gekennzeichnet. Die oberste *EClass*, in der wie in jedem EMF-Modell alle anderen über Containment-*EReferences* enthalten sind, ist in diesem Fall *Petrinet*. Weitere *EReferences* des Modells bestehen aus Referenzen der Stellen und Transitionen auf eingehende bzw. ausgehende Kanten. Das gesamte Modell stellt in EMF ein *EPackage* dar.

Aus diesem Modell wird zu jeder *EClass* jeweils ein Interface und eine

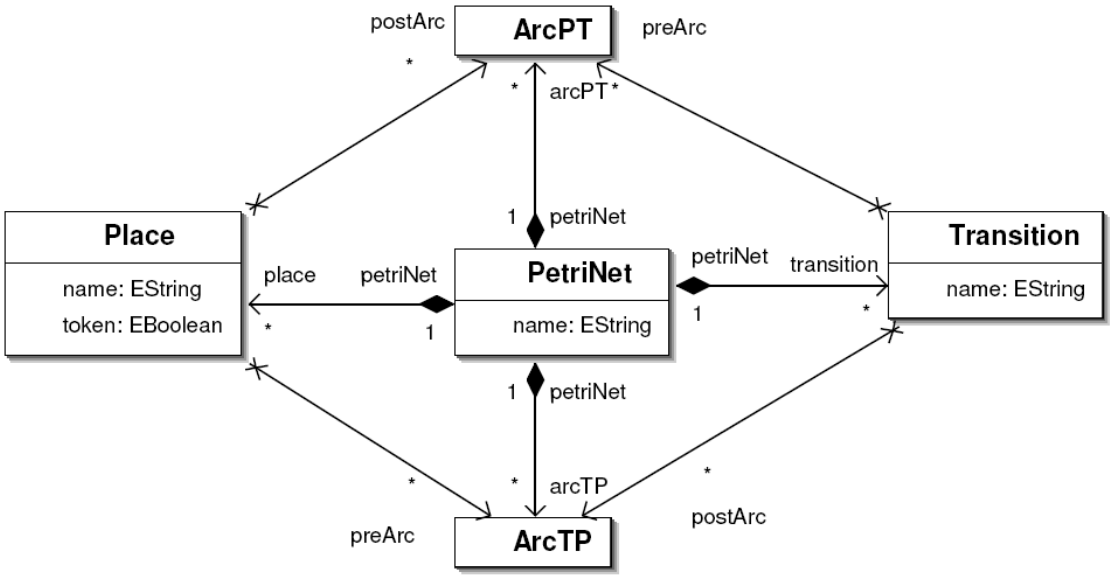

Abbildung 2.11: EMF-Modell eines Petrinetzes

Implementierungsklasse erzeugt - also zum Beispiel für *Place* das Interface *Place.java* und die Klasse *PlaceImpl.java*, in der die Methoden aus dem Interface implementiert werden. Diese umfassen jeweils einen Getter und einen Setter für jede (veränderbare) *EReference* bzw jedes *EAttribute*. Das Interface zu *Place* sieht dabei folgendermassen aus:

```
package petrimodel;

import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;

public interface Place extends EObject
{
  EList getPostArc();
  EList getPreArc();
```

```
  PetriNet getPetriNet();
  void setPetriNet(PetriNet value);

  String getName();
  void setName(String value);

  boolean isToken();
  void setToken(boolean value);
}
```

Zu den *EReferences preArc* und *postArc* wird hier kein Setter erzeugt, da diese jeweils eine Liste von Zielen haben. Mit *getPreArc*() wird zum Beispiel die Liste der *ArcTPs* zurückgegeben, aus der dann direkt Elemente entfernt oder hinzugefügt werden können.
Da alle generierten Klassen von *EObject* erben, haben sie auch aus dieser Klasse die generische Methoden wie *eClass*(), *eGet*() und *eSet*(). Über diese lassen sich alle Elemente einer Modellinstanz generisch verändern. Dies nutzen wir vor allem bei der Implementierung des Interpreters, um mit beliebigen EMF Modellen arbeiten zu können.
Da alle *EObjects* auch gleichzeitig das Interface *Notifier* implementieren, können über Adapter andere Objekte von Änderungen an einem Modell-Objekt erfahren. Wenn eine Veränderung geschieht, erhalten diese Adapter eine *Notification* von dem veränderten *EObject*. Dies ist zum Beispiel bei doppelseitigen Referenzen wichtig, um die Integrität des Modells zu bewahren - wenn eine Referenz verändert wird, dann auch automatisch die dazugehörige Referenz in umgekehrte Richtung. Weiterhin können auch Objekte ausserhalb des Modells, über Veränderungen informiert werden. Die Methode *setToken*() in der Implementierungsklasse *PlaceImpl.java* erzeugt gleichzeitig eine *Notification* wenn sie aufgerufen wird:

```
public void setToken(boolean newToken){
  boolean oldToken = token;
  token = newToken;
```

```
  if (eNotificationRequired())
    eNotify(new ENotificationImpl(this, Notification.SET,
    PetrimodelPackage.PLACE__TOKEN, oldToken, token));
  }
}
```

Mit dieser werden andere Objekte davon benachrichtigt, dass sich möglicherweise der Status des entsprechenden *Place* geändert hat - dadurch kann zum Beispiel in einem Editor die Figure, die dieses *Place* repräsentiert, anders dargestellt werden.
Zu jedem Modell wird ein $EPackage$ und eine $EFactory$ erzeugt. Die $EFactory$ hat für jede Klasse des Modells eine create-Methode, um Objekte mit diesem Typ zu erzeugen. Das $EPackage$ enthält eine Liste aller $EClassifier$ des Modells und kann zu einem bestimmten Namen die entsprechende $EClass$ zurückgeben.

2.2.3 JET

JET (Java Emitter Templates) [8] ist ein Tool innerhalb von EMF und dient dort zum Erzeugen von Java-Code zu einem Modell. Man kann es allerdings auch allgemeiner verwenden, um eigenen Code aus Templates und Datenobjekten zu generieren. Neben Java kann mit JET auch SQL oder XML erzeugt werden.
In den Templates befindet sich ein Skelett des zu generierenden Programms, Platzhalter, an denen Strings aus einem Datenobjekt eingefügt werden können, Variablen und Kontrollstrukturen wie Verzweigungen und Schleifen. Diese werden zunächst in ein Zwischenprojekt übersetzt, aus dem dann der eigentliche Code generiert wird. Wir benutzen JET beim Compiler, um zu Regeln Java-Code zu erzeugen. Dazu hier ein Beispiel wie das aussehen kann - zunächst der Code aus dem Template (*EcoreRule.javajet*):

```
<%@ jet package="Editor.rule" class="EcoreRule"
imports="de.tuberlin.emftransformation.generator.util.* java.util.*"%>
```

```
<%TransformationData data = (TransformationData) argument; %>
<%RuleData ruleData = data.getRule(); %>
...

...
<%for(Iterator it = ruleData.getLHSSymbols().iterator(); it.hasNext();){
SymbolData symbolData = (SymbolData) it.next();%>
/**
 * Getter for the LHS element <%=symbolData.getVariableName()%>.
 * @return current match for <%=symbolData.getVariableName()%>
 */
public <%=symbolData.getType()%> get<%=symbolData.getUpVariableName()%>(){
return this.<%=symbolData.getVariableName()%>;
}
<%}%>
```

Zuerst wird hier der Header eines Templates gezeigt: Dort ist festgelegt wie die Klasse zu diesem Template in dem Zwischenprojekt heisst, das JET bei der Codeerzeugung anlegt, und in welchem Package es sich dort befindet. Dann werden die Importe festgelegt, welche innerhalb des Platzhalter verwendet werden können. Dieses Template erhält als Argument ein Objekt vom Typ $TransformationData$ und bekommt aus diesem mit der Methode $getRule()$ ein $RuleData$, welches eine Regel darstellt.
Die Platzhalter befinden sich zwischen den $<\%\%>$ - Tags, alles was ausserhalb von diesen steht, wird unverändert in den Zielcode eingefügt. Weiterhin ist es möglich, Variablen zu definieren, die dann innerhalb der Tags lokal oder global gültig sind - wie in diesem Beispiel die Variable $ruleData$, welche global gültig ist, oder die Variable $symbolData$, deren Gültigkeit auf die for-Schleife beschränkt ist. Und schliesslich können Ausdrücke, die in Tags stehen, zu Strings ausgewertet werden, die dann in den zu generierten Code eingefügt werden. Dies ist überall der Fall wo zu Anfang eines getaggten Abschnittes ein Gleichheitszeichen steht. Der erzeugte Code kann dann folgendermassen aussehen - in diesem Fall wurde die for-Schleife zweimal durchlaufen und

dadurch zwei Getter generiert:

```
/**
* Getter for the LHS element eclassifier0.
* @return current match for eclassifier0
*/
public EClassifier getEclassifier0() {
return this.eclassifier0;
}

/**
* Getter for the LHS element epackage0.
* @return current match for epackage0
*/
public EPackage getEpackage0() {
return this.epackage0;
}
```

2.2.4 AGG

AGG [1] ist eine an der TU-Berlin entwickelte regelbasierte visuelle Sprache für Graphtransformationen. So ist es möglich mit dem grafischen Editor Graphtransformationen zu definieren und mit Hilfe der AGG-Engine anzuwenden.
AGG bietet unter anderem eine grafische Benutzeroberfläche, die in Abbildung 2.12 zu sehen ist.
Die Hauptkomponenten dieser Oberfläche sind eine Baumansicht der geladenen Graphtransformationen auf der linken Seite, ein dreiteiliges Panel mit der derzeit aktiven Regel, sowie den Hostgraphen auf den die Regeln angewendet werden sollen.
Die Hauptmerkmale von AGG sind:

1. Komplexe Datenstrukturen werden als Graphen modelliert, die über

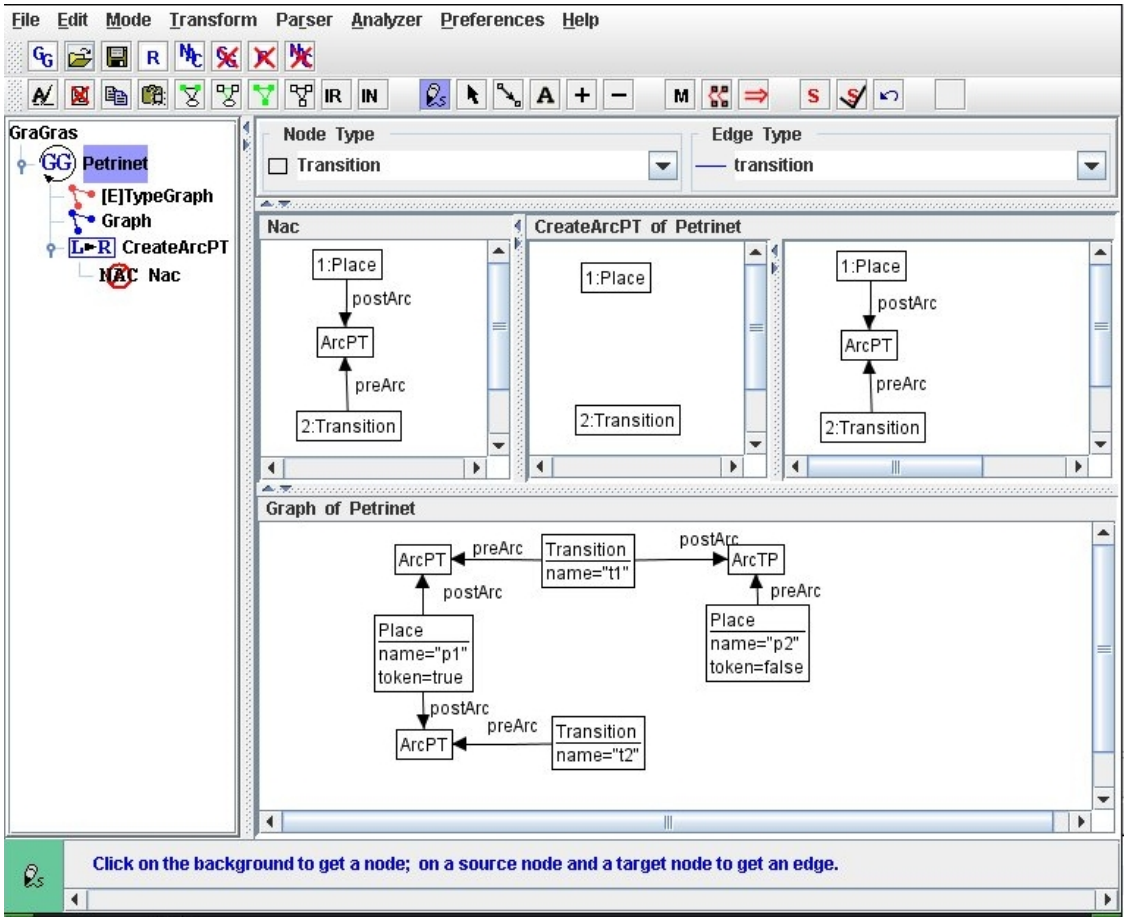

Abbildung 2.12: Agg Oberfläche

einem Typgraph getypt sein können.

2. AGG Graphen können durch Java Klassen attributiert werden.

3. Neue Java Klassen können hinzugefügt werden.

4. Das Verhalten des Systems wird von Graphgrammatikregeln beschrieben.

5. Regeln können neben ihrer linken und rechten Seite auch negative Anwendungsbedingungen beinhalten.

6. Die Anwendung einer Regel transformiert den Startgraph.

7. Die Anwendung mehrerer Regel zeigt einen möglichen Transformationsablauf.

8. Graphregeln können Java Methoden in den Attributen verwenden, die bei Regelanwendung ausgewertet werden.

9. Weiterhin können Attributbedingungen mit Java Ausdrücken verwendet werden.

Durch die Punkte 1-3 ist es mit AGG möglich Typgraphen äquivalent zu Klassendiagrammen zu modellieren, da Klassendiagramme zwar eine andere Terminologie verwenden, aber denoch den gleichen Zweck erfüllen.
Die Besonderheit von AGG ist aber, dass aufgrund des unterliegenden Graphtransformationsansatzes, dass Verhalten des Systems über Regeln beschrieben werden kann, wobei für die Änderung von Attributen auf bereits vorhande Java Methoden zurückgegriffen werden kann.
Neben diesen Punkten stehen dem Benutzer auch eine Reihe an Analysetechniken zur Verfügung, wie beispielsweise die kritische Paaranalyse, Termination des Systems und Konsistenzcheck. Für unsere Arbeit ist vor allem die Möglichkeit wichtig, die AGG Engine ohne die grafische Benutzeroberfläche anzusprechen. Hierzu kann man das Java Paket *agg.xt_basis* verwenden. Die Klassen innerhalb dieses Pakets erlauben es mit Hilfe von Methodenaufrufen eine Graphgrammatik zu definieren und anzuwenden. Beispielsweise definiert folgender Code zwei Typen und erstellt einen Graphen mit jeweils einem Knoten der beiden Typen:

```
TypeSet ts = new TypeSet();

Type placeType = ts.createType();
placeType.setStringRepr("Place");
Type transitionType = ts.createType();
transitionType.setStringRepr("Transition");

Graph g = new Graph(ts);
```

```
Node p1 = g.createNode(placeType);
Node t1 = g.createNode(transitionType);
```

Falls man sich auf diese Weise eine Grammatik aufgebaut hat, kann man ebenfalls über die Engine eine Regel anwenden. Hierzu erstellt man sich eine Graphtransformation *GraTra* und initialisiert sie mit der eigenen Grammatik.

```
GraTra gratra = new DefaultGraTraImpl();
gratra.setGraGra(myGrammar);
```

Anschliessend kann man die Transformation automatisch ablaufen lassen, wobei alle Regeln solange wie möglich angewendet werden, oder man wählt eine bestimmte Regel aus und versucht für diese einen Match zu finden.

```
Rule createArcPTRule = myGrammar.getRule("CreateArcPT");
Match m = aggGrammar.createMatch(rule);
if (m.nextCompletion()) {
  gratra.apply(m);
else
  System.out.println("Kein Match gefunden");
```

2.3 Bereits bestehende Ansätze zur Modelltransformation

Unser Tool reiht sich in bereits bestehende Ansätze ein, die wir hier kurz vorstellen wollen. Diese sind nur eine kleine Auswahl und führen Modelltransformationen in EMF oder mit Hilfe von Graphtransformation durch.

ATL (Atlas Transformation Language)

ATL [3, 4] wurde an der Universität Nantes in der Bretagne entwickelt und ist Teil des Eclipse-Projekts GMT (Generative Model Transformer). Es besteht im wesentlichen auf einer textuellen Definition der Transformationssprache,

die auf OCL basiert und überwiegend deklarative aber auch einige imperative Elemente enthält. Mit ADT (ATL Development Tools) gibt es ein Plugin für Eclipse, um Transformationen, die in ATL beschrieben wurden, auch auf EMF-Modellen durchzuführen. Mit diesem können sowohl exogene als auch endogene Modelltransformationen durchgeführt werden. Bei letzteren werden für alle Elemente der ursprünglichen Instanz, die nicht durch eine Regel gematcht werden, automatisch Kopien in der Ziel-Instanz angelegt.

TEFKAT (The Engine Formerly Known As TEFKAT)

TEFKAT [12] ist ein nicht-kommerzielles Eclipse-Plugin, das Modelltransformationen in EMF durchführt. Es wurde an der 'University of Queensland' in Australien entwickelt und unterstützt exogene Modelltransformation zwischen jeweils beliebig vielen Instanzen eines Quell- und eines Zielmodells. Eine Transformation wird über Regeln beschrieben, die sich stark am QVT-Ansatz [18] orientieren.

VMTS (Visual Modeling and Transformation System)

VMTS [13, 14] ist eine Modellierungsumgebung, die es erlaubt Modelle zu einem Metamodell zu entwickeln und zu verändern. Transformationen können dabei durch einzelne Schritte dargestellt werden, die eine bestimmte Reihenfolge haben oder, falls möglich, auch parallel angewendet werden können. Dies kann in der sogenannten VMTS Visual Control Flow Language (VCFL), ähnlich wie in einem Aktivity-Diagram festgelegt werden. Transformationen werden anhand von OCL-Constraints überprüft.

VIATRA2 (VIsual Automated model TRAnsformations)

VIATRA2 [9, 10] ist ein Framework das Unterstützung beim Entwickeln, Ausführen und Testen von Modelltransformationen bietet. Dabei können viele verschiedene Modelle importiert und Transformationen innerhalb des Frameworks durch einen Interpreter ausgeführt werden. Neben

Graphtransformation werden Abstract State Machines als theoretischer Hintergrund verwendet.

2.3.1 Klassifikation unseres Ansatzes und Vergleich

[17] Die Unterschiede unserere Arbeit zu anderen EMF-Tools besteht darin, dass wir eine graphische Repräsentation für unsere Regeln verwenden und mit dem Compiler Java-Code zu den Regeln generieren können. Ausserdem führen wir eine endogene Modelltransformation durch, während die meisten anderen Tools exogene Transformationen unterstützen.

Kapitel 3

Anforderungen

3.1 Funktionale Anforderungen

3.1.1 Transformation

Eines der Hauptaufgaben unserer Diplomarbeit ist die Entwicklung eines Transformationmodells für EMF-Modelle.
Weiterhin soll sich die Tranformation stark an Graphtransformationen orientieren, so sollen beispielsweise Transformationsregeln eine linke und rechte Seite besitzen, sowie mögliche negative Anwendungsbedingungen. Weitere Gemeinsamkeiten und Unterschiede werden später noch genauer beschrieben. Da für Graphtransformationen im grossen Umfang Theorie und Analysetechniken vorliegt, ist es so möglich bestimmte Eigenschaften wie Termination oder Konflikte einer Transformation nachzuweisen. Falls sich ein Benutzer eine Transformation definiert hat, kann er sie anschliessend auf ein geeignetes Modell anwenden. Hierfür sollen ihm sowohl ein Interpreter als auch ein Compiler zur Verfügung stehen. Beide sollen über mindestens ein gemeinsames Interface für die Eingabe der für eine Transformation nötigen Informationen verfügen.
Die Transformation selbst soll endogen durchgeführt werden, wobei sich exogene Modelltransformationen über die Zusammenführung beider EMF-Modelle simulieren lassen.

3.1.2 Interpreter

Der generelle Ablauf des Interpreters sieht die Übersetzung des Modells, der Modellinstanz und des Transformationsmodells in eine ausführbare Sprache für Graphtransformationen vor. Anschliessend werden die entsprechenden Änderungen in dieser Sprache durchgeführt und das Ergebnis zurückübersetzt. Durch die Anbindung an Graphtransformationen soll es weiterhin möglich sein bekannte Analysetechniken für Graphtransformationen auf EMF Transformationen zu übertragen.

3.1.3 Compiler

Nach der Implementierung des Interpreters soll ein Compiler für das Transformationsmodell entwickelt werden. Durch das Wegfallen des Interpretiervorgangs hoffen wir so eine höhere Effizienz zu erreichen und die Laufzeit der Transformation zu reduzieren.

3.2 Nichtfunktionale Anforderungen

3.2.1 Transformation

Das Transformationsmodell soll in EMF definiert werden und die Transformation selbst auf andere EMF-Modelle angewendet werden.

3.2.2 Interpreter

Der Interpreter soll die AGG-Transformationsengine verwenden. AGG bietet unter seiner grafischen Oberfläche eine Vielzahl von Analysetechniken für Graphtransformationen, wie zum Beispiel die kritische Paaranalyse für Regeln oder die Termination eines Transformationssystems.

3.2.3 Compiler

Aus dem Transformationsmodell soll ein Java Plugin Projekt erzeugt werden. Dieses Projekt soll Regelklassen enthalten, die Code aus einem EMF Projekt verwenden. Unter einem EMF Projekt verstehen wir den zu einem bestimmten EMF Modell von EMF automatisch erzeugten Code. Das Regelprojekt selbst wird mit Hilfe von JET aus dem Transformationsmodell erzeugt.

Kapitel 4

Modelltransformation

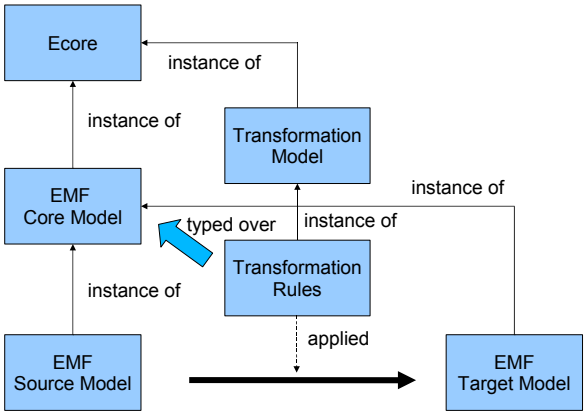

Abbildung 4.1: Transformationsübersicht

Grundsätzlich ist eine EMF Transformation eine regebasierte Veränderung eines EMF Modells zu einem Anderen. Sowohl das Quell- als das Zielmodell sind selbst wieder über dem EMF Core Modell getypt. Da wir nur endogene Modelltransformationen zulassen, fallen Quell- und Zielmodell zusammen.
Die Regeln selbst sind über dem Transformationsmodell getypt, siehe Abbildung 4.2. Das Transformationsmodell selbst ist ebenfalls über dem EMF Core Modell getypt.
Die Anwendungsgebiete von EMF Transformationen sind ähnlich denen von Graphtransformationen. Man kann sie zur Simulation des Verhaltens eines Systems verwenden, oder auch zur Erzeugung einer Sprache. Ein interessanter

Punkt bei der Spracherzeugung über EMF Modellen, ist die Existenz von GEF(Graphical Editing Framework) [5]. Mit Hilfe von GEF ist es möglich Editoren für visuelle Sprachen zu erstellen, deren unterliegendes Modell häufig EMF Modelle sind. So ist es möglich die Editieroperationen eines GEF Editors mit Hilfe von Regeln zu definieren und kann auf die manuelle Modifikation des unterliegenden Modells verzichten.

4.1 Transformationsmodell

4.1.1 Beschreibung der Klassenstruktur

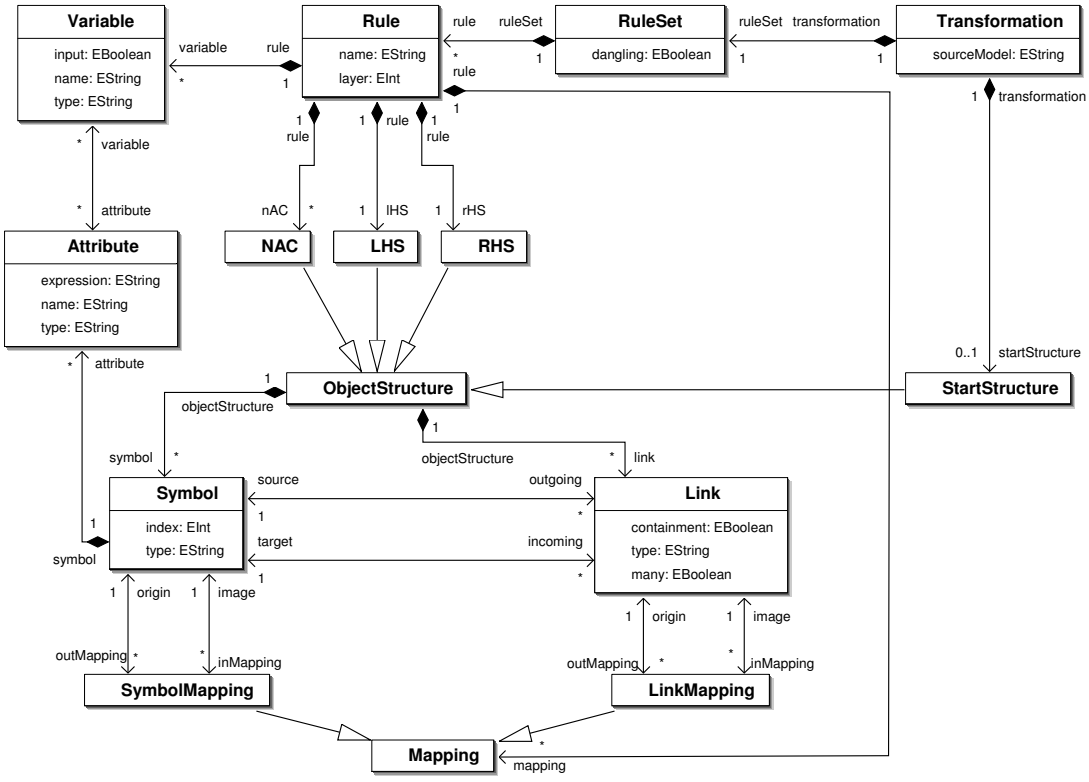

Abbildung 4.2: Transformationsmodell

Unser Transformationsmodell aus Abbildung 4.2 orientiert sich stark an den von Graphtransformationen eingeführten Konzepten der regelbasierten

Transformation.
Die oberste Klasse Transformation dient als Container für alle weiteren Klassen. Am Besten kann man sie mit einer Graphgrammatik auf Graphtransformationsseite vergleichen. Das in ihr befindliche Attribut sourceModel bezieht sich auf ein im Dateisystem vorhandenes Ecore Modell.
In einer Transformation enthalten sind eine Regelmenge, genannt RuleSet, und eine Startstruktur, die als Ausgangspunkt einer Transformation dienen kann. Das RuleSet enthält die Option dangling, mit der man entscheiden kann ob bei der Regelanwendung die Dangling Condition beachtet werden soll oder nicht.
Die Regeln, die im RuleSet enthalten sind, haben jeweils einen Namen und

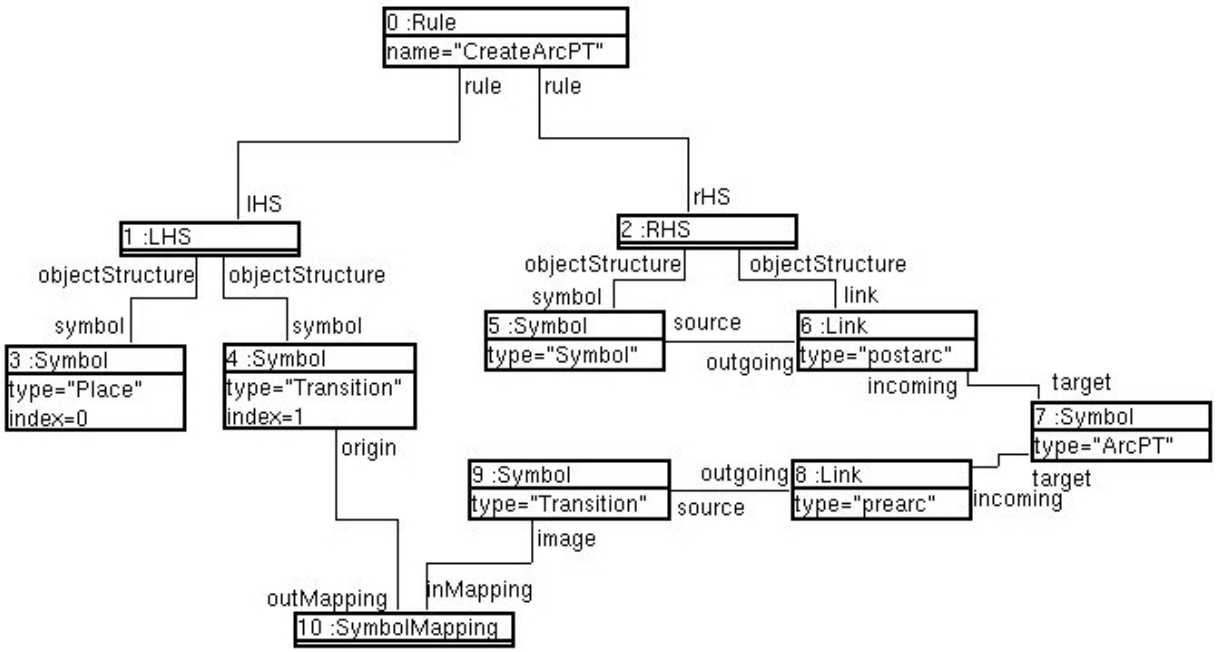

Abbildung 4.3: Modellinstanz zur Regel CreateArcPT

einen Layer. In Abbildung 4.3 ist ein Ausschnitt der Transformationsinstanz für der Regel CreateArcPT zu sehen. Einige Klassen, wie z.B. das Mapping zwischen den Places, wurden aufgrund der besseren Lesbarkeit weggelassen. Der Name in Rule ist ein eindeutiger Bezeichner über den die Regel identifiziert werden kann. Einen Layer kann man setzen, falls die Transformation geschichtet

ablaufen soll. Dabei werden Regeln mit dem kleinsten Layer als erstes ausgeführt und alle Regeln die im selben Layer liegen in beliebiger Reihenfolge. Im Fall von CreateArcPT wurde nur der Name der Regel gesetzt und kein Layer angegeben. Zusätzlich kann eine Regeln Variablen enthalten. Variablen haben einen bestimmten Namen, Typ und die Information ob sie Input Parameter für die Regel sind oder nicht. Variablen können mit Attributen verbunden werden und erlauben es den Attributen so sich in ihrer Expression auf die Variablen zu beziehen. Im Fall von CreateArcPT werden keine Variablen benötigt.
In einer Regel sind weiterhin verschiedene Objektstrukturen enthalten. Mindestens vorhanden sind eine LHS (linke Regelseite) und eine RHS (rechte Regelseite). Zusätzlich kann noch eine beliebige Menge an NAC Strukturen definiert werden.
All diese Objektstrukturen einer Regel, genauso wie die Startstruktur, enthalten Symbole und Links. Symbole und Links sollen Teile der Klassenstruktur widerspiegeln, daher stellen Links Beziehungen zwischen Symbolen dar, analog zur Beziehung von Klassen und Referenzen.
Im Fall von CreateArcPT haben wir sowohl einen Place als auch eine Transition auf der linken Regelseite. Auf der rechten Regelseite haben wir zusätzlich noch zwei Links und ein ArcPT Symbol.
Symbole enthalten einen Typ, der namentlich dem Typ der Klasse auf die es sich bezieht entspricht und einen Index, der eine eindeutige Initialisierungsreihenfolge unter den Symbolen festlegt. Da dies nur für Symbole der linken Seite eine Rolle spielt, muss der Index auch nur dort gesetzt werden. Bei CreateArcPT hat der Place den Index 0 und die Transition den Index 1 bekommen, daher würde beim Übergeben der EObjects an die Regel zuerst der Place erwartet werden. Beim Compiler kann man diese strenge Reihenfolge umgehen indem man die passenden Setter Methoden zum Setzen der Objekte verwendet, dennoch muss bei der Erstellung der Regel ein Index angegeben werden, da er für die Nummerierung von gleich getypten Objekten verwendet wird, z.B. Place0, Place1 usw..
Für jedes Symbol können Attribute definiert werden, die einen Namen, einen Typ und eine Expression haben. Falls ein bestimmtes Attribut mit Variablen verbunden ist, können innerhalb der Expression diese Variablen verwendet

werden.
Die Links besitzen ebenfalls einen Typ und zusätzlich bestimmte Informationen über ihren Status als Containment- oder Many-Kante.
Als letztes besteht noch die Möglichkeit verschiedene Symbole oder Links über SymbolMappings beziehungsweise LinkMappings zu verbinden. Dies ist vorallem in Regeln wichtig um zu zeigen, dass es sich bei bestimmten Symbolen aus LHS und NAC/RHS um das gleiche handelt.

4.1.2 Semantik des Transformationsmodells

Um die Funktionsweise und den Ablauf einer Transformation besser verstehen zu können, lohnt es sich einen genaueren Blick auf den Aufbau einer Regel zu machen und die Bedeutung der einzelnen Komponenten.
Generell lehnen sich die Regeln stark an solche der algebraischen Graphersetzung an.
Wie bereits im letzten Abschnitt beschrieben, besteht eine Regel aus einer linken und einer rechten Seite, sowie einer Menge von negativen Anwendungsbedingungen. Ähnlich zu Graphgrammatikregeln, beschreibt die linke Seite die Vorher-Situation. Die Anwendung einer Regel verlangt, dass die in der linken Seite beschriebene Struktur in der Instanz gefunden wird. Somit stellt die LHS eine Art positive Anwendungsbedingung dar.
Negative Anwendungsbedingungen können mithilfe von NACs formuliert werden. Eine Regel ist nur anwendbar wenn die Struktur der NAC nicht in der Instanz gefunden wurde. Falls alle Vorbedingungen erfüllt sind, kann die Regel angewendet werden. In diesem Fall beschreibt die rechte Regelseite die Nachher-Situation. Durch Mappings lassen sich jeweile Objekte der linken Regelseite mit Objekten der NAC bzw. RHS verbinden. Eine Verbindung zweier Objekte über ein Mapping entspricht der Gleichsetzung dieser.
Beispielsweise bedeutet eine Verbindung zweier Objekte aus LHS und RHS, dass dieses Objekt von der Regelanwendung bewahrt werden soll. Daraus folgt auch, dass RHS Objekte ohne Mapping erzeugt und LHS Objekte ohne Mapping gelöscht werden. Somit lassen sich über das Mapping und die einzelen Regelseiten strukturelle Änderungen beschreiben.

Inhaltliche Änderungen dagegen, können über Attribute und Variablen beschrieben werden. Die Attribute eines Symbols entsprechen dabei vom Namen und Typ dem Attribut der Klasse, auf die sich das Symbol bezieht. Man kann einem Attribut eine Expression zuweisen, allerdings ist diese konstant, sollte das Attribut nicht mit einer Variable verbunden sein. Für den Fall das sich das Ergebnis einer Regel aus ihrem jeweiligen Umfeld ergeben soll, kann man Variablen definieren und diese mit Attributen der linken Regelseite verbinden. Bei der Regelanwendung wird die Variable mit dem jeweiligen Attribut initialisert und die Variable kann dann in Attributen der rechten Regelseite für Berechnungen verwendet werden. Variablen können ebenfalls als Schnittstelle für den Benutzer dienen falls sie als Inputparameter definiert wurden. In diesem Fall wird verlangt, dass sie vor Anwendung der Regel manuell gesetzt werden.

4.1.3 Explizites vs. implizites Mapping

Eines der Hauptmerkmale von Regeln, egal ob bei Graphtransformationen oder EMF Transformationen, ist die Abbildung von Objekten der linken Regelseite zu Objekten aus NAC beziehungsweise RHS. Da wir bei unserer Diplomarbeit generell nur injektive Regeln zulassen wollen, bieten sich zwei Arten an das Mapping zu definieren, eine implizite und eine explizite. Bei der impliziten Darstellung wird, falls es für ein Objekt ein Mapping gibt, auch nur genau ein Objekt erstellt und mit den jeweiligen Objektstrukturen in denen es auftaucht verbunden. Dadurch ist implizit festgelegt, dass es sich um dasselbe Objekt handelt. Der Vorteil der impliziten Darstellung ist die geringere Menge an Klassen, da für jedes Objekt genau eine Klasse erzeugt wird. Bei der expliziten Darstellung werden für jede Objektstruktur und alle in ihr vorhandenen Objekte Klassen erzeugt. Die Gleichheit zwischen den Objekten verschiedener Objektstrukturen wird jeweils durch eine Mappingklasse angegeben, welche zwei Klassen verbindet. Der Vorteil der expliziten Darstellung ist die höhere Flexibilität bei der Definition von Beziehungen zwischen Objekten, die aber wegen der Einschränkung auf Injektivität nicht ausgenutzt wird. Der Grund warum wir uns dennoch für die explizite Darstellung entschieden haben, ist

die Modellierung von Vererbung. Falls Klassen über eine Vererbungshierarchie miteinander verbunden sind, wollen wir es zulassen, dass bestimmte Kindklassen von einer Transformationsregel ausgeschlossen werden können indem man diesen Typ in einer NAC angibt. Dass heisst es kann vorkommen, dass ein Objekt vom Elterntyp aus der linken Regelseite auf ein Objekt eines Kindtyps zeigt. Da dies bei der impliziten Darstellung nicht, oder nur umständlich über Attribute zu realisieren ist, verwendet unser Transformationsmodell ein explizites Mapping.

4.1.4 Vergleich von Graphtransformation und EMF-Transformation

Da sich die EMF-Transformation stark an Graphtransformationen anlehnt, stellt sich die Frage inwieweit sich die jeweiligen Konzepte entsprechen. Dies gilt vor allem für die Übersetzung von EMF Konstrukten in Graph Konstrukte. Beispielsweise wird die Typisierung eines Graphen durch dessen Typgraph angegeben, während die Typisierung auf EMF Seite durch das EMF-Modell beschrieben wird. Am Petrinetz-Beispiel lässt sich sehr gut die Ähnlichkeit des Typgraphen und des EMF-Modells sehen, siehe Abbildungen **??** und 2.11.
Der Graph selbst, auf den die Graphtransformation angewendet werden soll, ist über seinem Typgraph getypt, dass heisst es gibt einen Morphismus vom Graphen in seinen Typgraph. Das äquivalente Konstrukt auf EMF Seite ist in diesem Fall die Modellinstanz. Die Modellinstanz verwendet ähnlich einem Graphen die Typen des Modells. Ein Unterschied von EMF Instanzen zu Graph ist die Behandlung von Beziehungen zwischen Klassen. Die Beziehungen werden durch $EReferences$ beschrieben, die aber im Gegensatz zu Kanten nicht explizit definiert werden können, sondern einer Klasse gehören müssen. Damit ähneln sie mehr Attributen und beschreiben eher die interne Struktur einer Klasse wogegen Kanten die Struktur eines Graphen mitbestimmen und nicht einem bestimmtem Knoten zugeordnet sind.
Darüberhinaus besitzen EMF Instanzen eine explizite Baumstruktur, die durch die Containmentbeziehung zwischen den Klassen entsteht. Diese Beziehung beschreibt das Enthaltensein einer Klasse in einer anderen und für die daraus erzeugte Hierarchie innerhalb des EMF Modells gibt es auf Graphseite keine

Entsprechung.

4.2 Editieren von Transformationen

Um Instanzen des Transformationsmodells zu erstellen, stehen einem Benutzer verschiedene Möglichkeiten offen. Die erste, aber auch umständlichste, Art eine Instanz des Transformationsmodells zu erstellen, ist der automatisch zu jedem EMF Modell generierbare Editor. Mit diesem kann man in einer baumbasierten Ansicht, der der Struktur des Modells entspricht Instanzen, generieren. Als weitere Editoren stehen zum einen AGG und zum anderen der von Eduard Weiss entwickelte Regeleditor für das Transformationsmodell zur Verfügung.

4.2.1 Editieren mit dem EMF-Editor

Zusätzlich zur Erzeugung von Modellcode, liefert EMF zusätzlich einen baumbasierten Editor zum Erstellen dieses Modells.
Dieser stellt eine sehr simple und umständliche Möglichkeit dar Transformationen zu erstellen.

4.2.2 Editieren mit AGG

Da es möglich ist, AGG Grammatiken zu importieren beziehungsweise in Instanzen des Transformationsmodells umzuwandeln, steht mit der AGG Oberfläche ein grafischer Editor zur Verfügung.
Um Allerdings in AGG editieren zu können, muss zuerst einmal der Typgraph aus dem EMF Modell erzeugt werden.

4.2.3 Editieren mit dem Regeleditor

Der von Eduard Weiss entwickelte Regeleditor, siehe Abbildung 4.4, stellt direkt eine grafische Oberfläche zum Erstellen von Transformationen bereit. Er importiert ein beliebiges EMF Modell und passt die Arbeitsumgebung entsprechend an, um Regeln für dieses Modell zu erstellen.

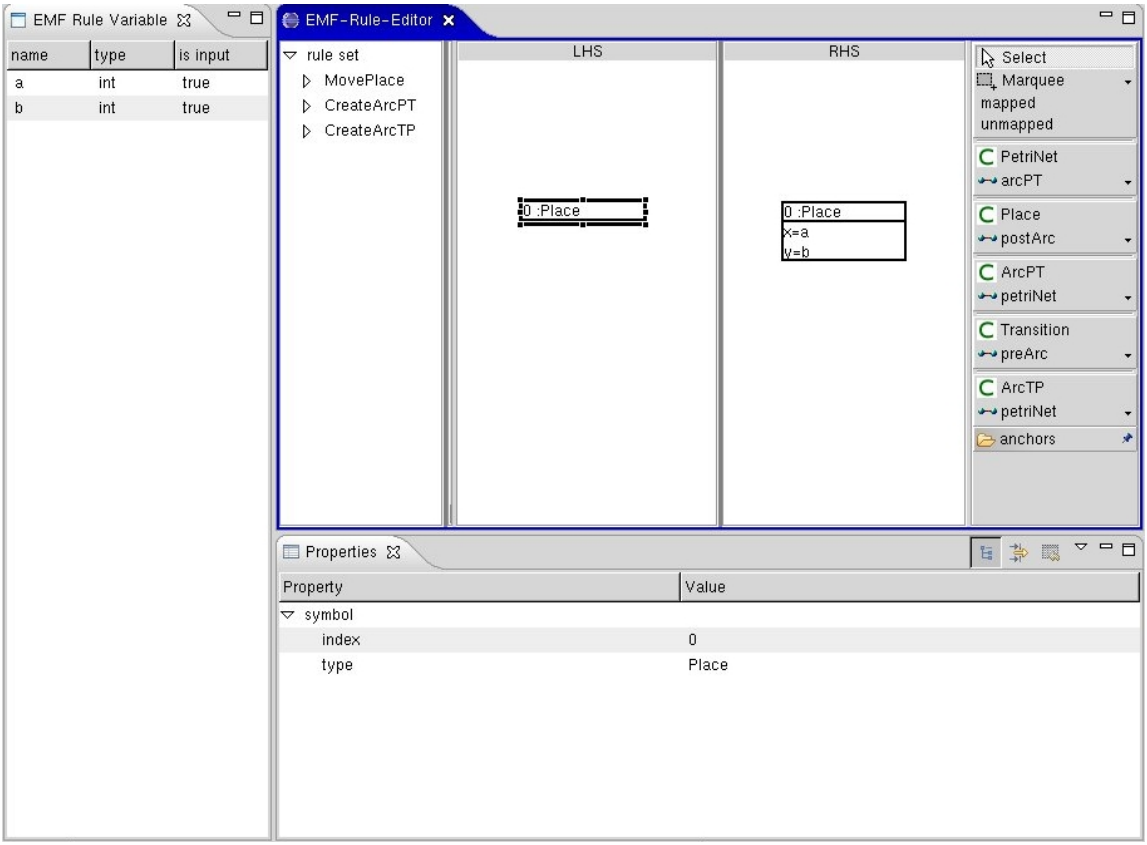

Abbildung 4.4: Regeleditor

Beispielsweise für die Erstellung der *CreateArcPT* Regel unseres Petrinetzbeispiels sind folgende Arbeitsschritte notwendig:

1. Wechseln in die EMF Rule Perspektive
2. Erstellen eines neuen Projekts
3. New→EMF Transformation
4. Im Wizard siehe Abbildung 7.8 die Petrinet ecore Datei auswählen
5. Über das Contextmenu von *ruleset* den Punkt *create rule* auswählen und den Namen *CreateArcPT* für die Regel vergeben

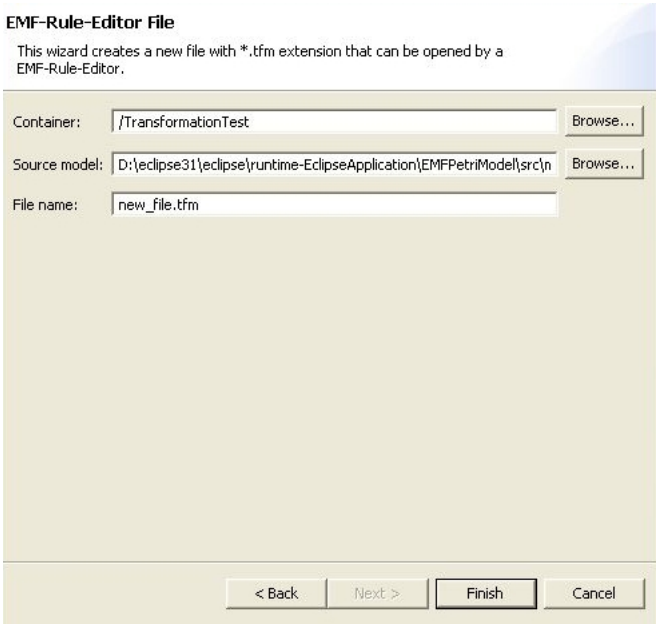

Abbildung 4.5: Wizard für eine neue Transformation

Nach diesen Schritten kann man aus der Baumansicht die Regel auswählen und im Hauptfenster editieren.
Als nächstes kann man die verschiedenen Klassen, die man für die Regel braucht, aus der Palette auswählen und in den jeweiligen Regelseiten platzieren, ähnlich wie in Abbildung 4.6. Am Ende müssen noch Mappings zwischen einzelnen Objekten definiert werden, in dem aus der Palette *mapped* ausgewählt wird und dann auf die zwei Objekte geklickt wird, die aufeinander abgebildet werden sollen.

4.3 Ausführung der Transformation

4.3.1 Semantik der Transformation

Eine definierte Transformation soll auf eine Modellinstanz, die gleich getypt ist, anwendbar sein. Die Anwendung einer Regel auf eine Modellinstanz setzt sich aus drei Schritten zusammen:

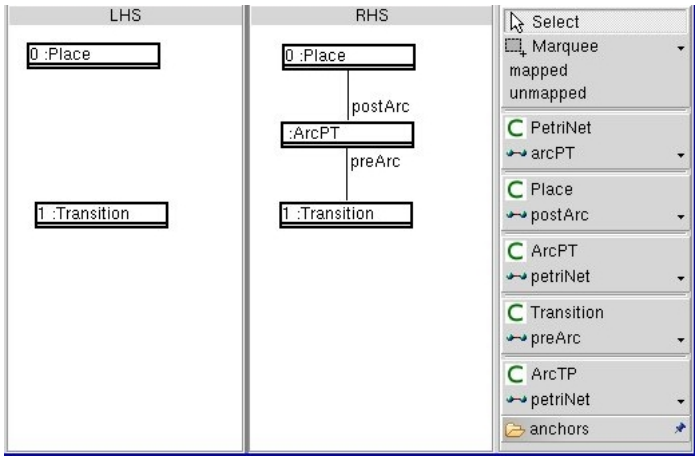

Abbildung 4.6: CreateArcPT Regel im Editor

- Matchsuche
- Regelanwendung
- Bereinigung

Matchsuche

Eine Regel besteht aus Objekten, die bestimmten EMF Typen entsprechen. Ein Match an einer bestimmten Stelle in der Modellinstanz kann dann zustande kommen, wenn alle Objekte der Regel auf EMF Klassen des Typs abgebildet werden können, dem sie entsprechen. Zusätzlich darf es keine vollständige Abbildung einer NAC Struktur in die Modellinstanz geben. Falls ein solches strukturelles Abbild der Regel in der Modellinstanz gefunden wurde, müssen noch die Attribute der einzelnen Objekte der Regel überprüft werden. Für alle Attribute, die nicht in der Regel definiert wurden, aber aufgrund des Modells in Klassen der Modellinstanz vorkommen, werden nicht weiter beachtet und stören die Transformation somit nicht. Letztendlich müssen alle Variablen, die als Inputparameter definiert sind, mit Werten belegt sein.

Regelanwendung

Nachdem ein Match an einer bestimmten Stelle in der Modellinstanz gefunden wurde, kann die Regel angewendet werden. Als erstes werden alle Variablen, die nicht als Inputparameter übergeben wurden, durch den konkreten Match initialisiert. Das heisst die Variablen bekommen den Wert des Attributs, auf das sie sich beziehen. Anschliessend werden die von der Regel definierten strukturellen Änderungen durchgeführt. Klassen für LHS Objekte ohne Mapping werden gelöscht, Klassen für RHS Objekte ohne Mapping erzeugt. Am Ende werden die Attributänderungen durchgeführt. Falls für eine der neu erzeugten Klassen einige Attribute nicht durch die Regel initialisiert werden, werden sie auf undefiniert gesetzt.

Bereinigung und Konsistenz einer EMF-Instanz

Da EMF-Modell im Gegensatz zu Graphen zusätzlich zu gewöhnlichen Kanten auch Containment Kanten enthalten können, stellt sich die Frage, was mit dem Modell passiert sollte eine Klasse aus dem Containment-Baum mit einer Klasse verbunden sein, die sich nicht im Baum befindet. Für uns ist das Modell in diesem Fall inkonsistent, da u.a. EMF ein solches Modell auch nicht mehr abspeichern kann. Wenn man das Modell in Abbildung 4.7 betrachtet, könnte eine mögliche Instanz dieses Modells aussehen wie in Abbildung 4.8.

Falls jetzt in dieser Instanz die Containment-Beziehung zwischen *Graph* und *Subgraph* gelöscht wird, sind sowohl *Subgraph* als auch *Node* nicht mehr in *Graph* enthalten, dennoch bleibt die Beziehung *node* zwischen *Graph* und *Node* bestehen, siehe Abbildung 4.9. In diesem Fall bezeichnen wir das Modell als inkonsistent, da *node* auf eine dem Modell unbekannte Klasse verweist.

Neben dieser Art von Inkonsistenzen, die beim Löschen von auftreten kann, gibt es noch die Möglichkeit, Inkonsistenzen beim Erzeugen von Klassen zu erhalten, z.B. wenn man eine Klasse einfügt, die nicht über eine Containmentbeziehung, aber über eine gewöhnliche mit dem Baum verbunden ist.

Um solche Inkonsistenzen zu vermeiden, bleiben uns jetzt zwei Möglichkeiten. Wir können zum einen die Anwendung solcher Regeln verbieten, die eine

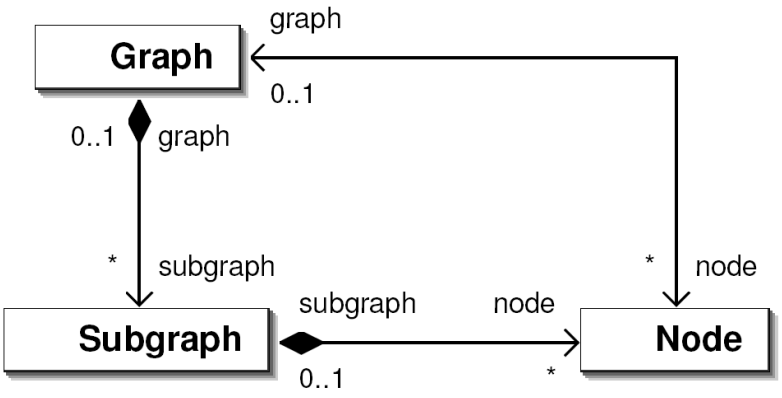

Abbildung 4.7: Beispielmodell für Konsistenz

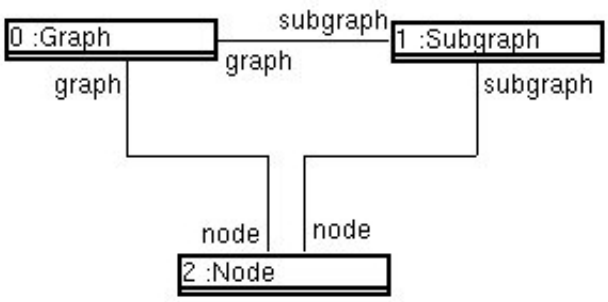

Abbildung 4.8: Beispielinstanz für Konsistenz

Inkonsistenz verursachen, oder wir passen das Modell so an, dass es wieder konsistent wird. Das Problem mit dem Verbot von Regeln ist, dass man bei Inkonsistenzen durch Löschen meist erst bei der Regelanwendung die Inkonsistenz erkennen kann, während die Inkonsistenzen durch Erzeugen statisch ermittelt werden können. Die zweite Möglichkeit, das Bereinigen von Inkonsistenzen, lässt sich dagegen auf jeden Fall anwenden. Bei der Bereinigung werden alle Nicht-Containmentkanten zwischen dem Containmentbaum und den restlichen Klassen zusätzlich gelöscht. Dadurch wird sichergestellt, dass das Modell konsistent bleibt, allerdings stellt es eine Abweichung von der Semantik von Graphtransformationen dar.

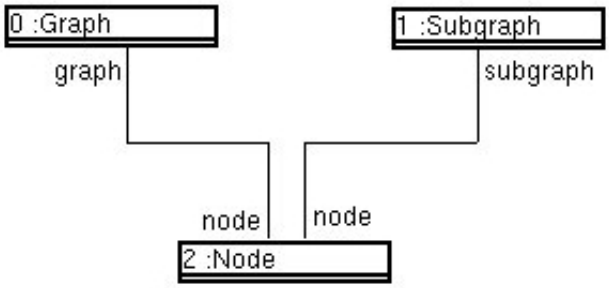

Abbildung 4.9: Inkonsistente Instanz

4.3.2 Ausführung mit Interpreter

Um eine Transformation mit dem Interpreter durchzuführen, muss dieser ersteinmal initialisiert werden. Dies geschieht beispielsweise über den Aufruf:

```
Interpreter interpreter = new Interpreter(petrinet1);
```

*petrinet*1 ist dabei eine beliebige Klasse aus der Modellinstanz, auf die der Interpreter angewendet werden soll.
Falls der Interpreter auf diese Weise initialisiert wurde, kann man eine Regel über $applyRule(Regelname, Mapping, Parameter)$ anwenden. Der Regelname ist dabei der Name der Regel als String, Mapping ist ein Vektor von EObjects und beschreibt auf welche Objekte der Instanz die Regelsymbole abgebildet werden sollen. Parameter wird schliesslich dazu verwendet, die Inputparameter der Regel zu setzen.

4.3.3 Ausführung mit Compiler

Mit dem Compiler muss man zuerst ein Regelprojekt aus seiner EMF Transformation erzeugen, genaueres dazu findet sich in Kapitel 6. Falls man ein solches Regelprojekt hat, kann man sich eine bestimmte Instanz einer Regelklasse erzeugen, beispielsweise

```
CreateArcPTRule createArcPTRule = new CreateArcPTRule(petrinet1);
```

Wie beim Interpreter ist *petrinet*1 ein beliebiges Instanzobjekt. Zum Setzen der Mappingobjekte und Inputparameter stehen dem Benutzer jetzt verschiedene Möglichkeiten offen. Zum einen gibt es die Methode $setLHS(Mapping)$, die ähnlich der Methode *applyRule* des Interpreters einen Vektor von EObjects erwartet. Zum Setzen von Inputparametern gibt es beim Compiler allerdings feste Setter Methoden, zum Beispiel würde für einen Inputparameter *String a* die Methode $SetA(Stringnewa)$ erstellt werden.
Nach dem Setzen der Mappingobjekte und Inputparameter kann man die Regel dann über die *execute*() Methode aufrufen:

```
createArcPTRule.execute();
```

Kapitel 5

Interpreter

5.1 Arbeitsweise

Beim Interpreter wird eine Regel nicht direkt auf eine EMF-Instanz angewendet, sondern auf einen AGG-Graphen, der die Struktur der Instanz hat. Dazu muss zunächst eine Instanz in einen solchen äquivalenten Graphen umgewandelt werden. Die Veränderungen an dem AGG-Graphen, die durch die Anwendung der Regel verursacht wurden, müssen dann noch interpretiert und zurück auf den EMF-Graphen übertragen werden. Um mit allem möglich EMF-Modellen operieren zu können, arbeitet der Interpreter mit den generischen Methoden wie $eGet()$, $eSet()$ und $eClass()$, die alle Objekte aus EMF-Modellen gemeinsam haben. Dafür benötigt er nur die Namen von Typen, Referenzen oder Attributen, die er aus dem $EPackage$ bekommt. Aus dem $EPackage$ kann ebenfalls der Typgraph für die AGG-Grammatik erzeugt werden.

Weil ein großer Teil der Funktionalität durch AGG übernommen wird, ist der eigentliche Interpreter recht kompakt und besteht nur aus wenigen Klassen. Da es für den Interpreter nötig ist, die Transformation von EMF nach AGG und umgekehrt durchzuführen, ergibt sich dadurch ein höherer Rechenaufwand als wenn man die Regel, wie beim Compiler, direkt in EMF anwenden würde. Die generischen Methoden von EMF sind auch etwas aufwändiger als direkt die Getter und Setter im erzeugten EMF-Code zu benutzen.

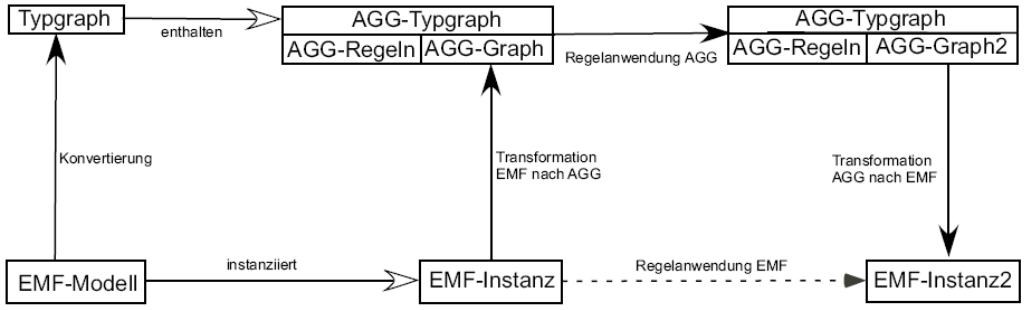

Abbildung 5.1: Interpreter

5.1.1 Initialisierung

Der Interpreter kann Regeln verarbeiten, die er in Form einer AGG-Grammatik oder einer Instanz unseres Regelmodells bekommt. Im zweiten Fall muss diese zunächst in eine AGG-Grammatik umgewandelt werden. In dieser werden dann später die Regeln angewendet.
Der Typgraph der AGG-Grammatik kann aus einem EPackage erzeugt werden. Dazu muss die Methode $createTypeGraphFromEPackage(EPackage\ ePackage)$ aufgerufen werden. Diese sucht aus einem $EPackage$ zunächst alle $EClasses$ heraus und untersucht anschliessend deren $EReferences$ und $EAttributes$.
Eine Referenz auf die Modellinstanz, auf der der Interpreter operieren soll, bekommt er durch einen beliebigen Baumknoten aus dieser. Die Regeln werden nur auf alle Knoten unterhalb, des übergebenen angewendet - falls es das oberste Element in der EMF-Baumstruktur ist, also auf die gesamte Modellinstanz.

5.1.2 Transformation nach AGG

In AGG befindet sich durch die Initialisierung bereits eine Grammatik, welche die Regeln sowie einen Typgraphen enthält, der zum Modell der Instanz passen sollte. Der AGG-Graph zu dieser Grammatik, auf den die Regeln angewendet werden sollen, wird nun aus der Instanz generiert, indem zu Objekten und

Referenzen aus EMF in einem AGG-Graph Knoten und Kanten mit den jeweils entsprechenden Typen aus dem AGG-Typgraph erzeugt werden. Die source- und target-Beziehungen der Kanten in AGG bestehen zu den Knoten, die aus der EMF-Instanz zu den source- und target-Objekten der entsprechenden Referenz erzeugt wurden. Attribute werden in AGG ebenso gesetzt wie in EMF.

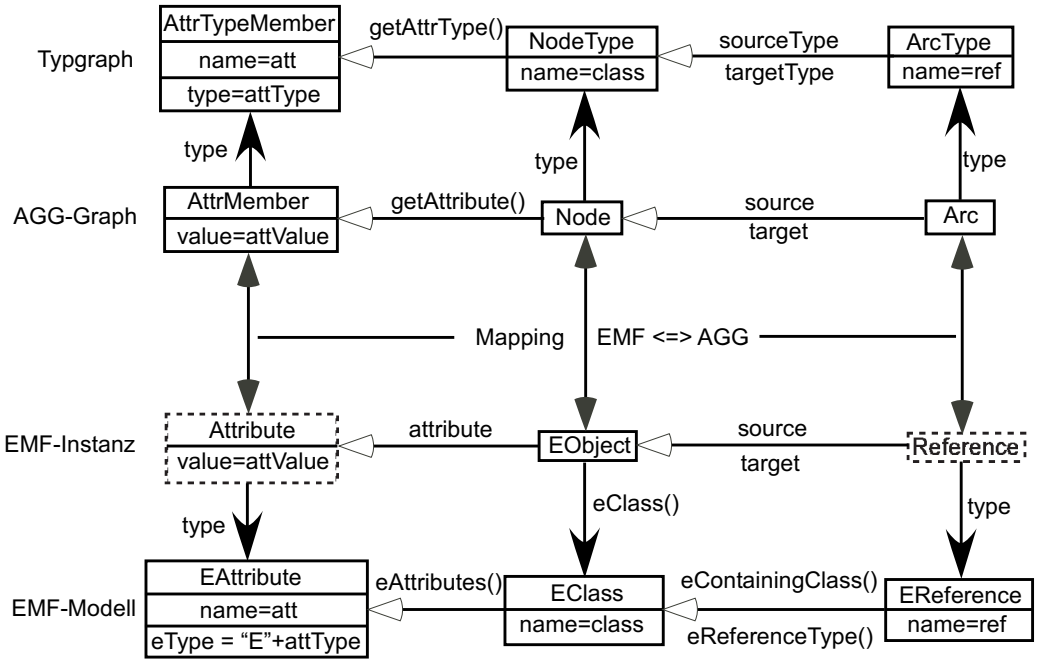

Abbildung 5.2: Mapping zwischen EMF und AGG

Gleichzeitig werden bei dieser Transformation zwei Hashmaps gefüllt, um die Beziehungen zwischen den Elementen aus dem AGG-Graphen und den entsprechenden Objekten und Referenzen auf der EMF-Seite herzustellen. Dies wird dann für die Rücktransformation benötigt, um festzustellen, welche Knoten oder Kanten in AGG neu entstanden sind oder gelöscht wurden. Dies ist der dazu entsprechende Code aus der Methode *applyRule*():

```
//transformation EMF-AGG
```

```
aggToEmf = new HashMap<GraphObject, EObject>(5);
emfToAgg = new HashMap<EObject, GraphObject>(5);

aggGrammar.getGraph().clear();
convertClasses2Nodes(root);
convertReferences2Arcs(root);
```

5.1.3 Anwendung der Regel

Die eigentliche Regelanwendung erfolgt in AGG. Da sowohl der Interpreter als auch AGG in Java implementiert sind, importiert dieser einfach die nötigen AGG-Klassen und ruft die entsprechenden Methoden aus diesen auf. Hierbei muß nur zwingend der Regelname angegeben werden. Input-Parameter und Vorgaben für den Match können beliebig angegeben oder leer gelassen werden, sofern die Regel in AGG trotzdem anwendbar bleibt. Dies ist der Code, mit dem wir innerhalb der Methode *applyRule*() eine Regel in AGG ausführen lassen, nachdem wir Input-Parameter und ein partielles Match vorgeben:

```
//rule execution in AGG
GraTra gratra = new DefaultGraTraImpl();
gratra.setGraGra(aggGrammar);

agg.xt_basis.Rule rule = aggGrammar.getRule(rulename);
rule.setInputParameters(parameters.getParameterMap());
Match m = aggGrammar.createMatch(rule);

//give (partial) mapping
...

//apply rule if match is found
if (m.nextCompletion()) {
gratra.apply(m);
```

```
result = true;
} else result = false;
```

5.1.4 Transformation von AGG nach EMF

Wenn die Regelanwendung erfolgreich war, hat sich der Graph in AGG auf die gewünschte Weise verändert. Anhand der Hashmaps, die bei der ersten Transformation von EMF nach AGG angelegt wurden, lässt sich feststellen welche Knoten und Kanten in AGG dabei neu entstanden oder entfernt wurden. Durch die Regel erzeugte Knoten werden auch in EMF neu erzeugt, gelöschte werden aus dem EMF-Baum entfernt. Die Referenzen in EMF werden zunächst alle gelöscht und nur diejenigen, zu denen eine Kante in AGG existiert, werden wieder neu gesetzt. Die Attribute in EMF werden einfach mit den Werte aus AGG überschrieben.
Wenn dieser Schritt abgeschlossen ist, sollten sich der AGG-Graph und die EMF-Instanz wieder so weit entsprechen wie dies in Abb. 5.2 gezeigt wurde. Allerdings werden beim Löschen eines *EObjects* aus der EMF-Instanz auch alle anderen Knoten, die sich in der Baumstruktur unterhalb von diesem befinden, automatisch mit gelöscht, was in AGG nicht der Fall ist.
Die Rücktransformation wird mit den folgenden Methodenaufrufe aus *Grammar.applyRule*() durchgeführt:

```
//create objects and set attributes
convertNodes2Classes(root);
//delete all references
clearRefs(root);
//restore references according to AGG
buildRefs(root);
//delete objects
deleteClasses(root);
```

5.1.5 Wiederherstellung der Konsistenz

Wie schon bereits erwähnt, hat der EMF-Graph eine Baumstruktur und darf keine Referenzen auf Knoten enthalten, die sich außerhalb von dieser befinden. Dieses kann aber nach einer Regelanwendung nicht immer gewährleistet werden. So können immer noch Referenzen auf gelöschte Knoten oder Knoten unterhalb eines solchen existieren.
Dies geschieht in der Klasse *ConsistenceRestorer*: Da das oben beschriebene ein bekanntes Problem in EMF ist, gibt es in der Klasse $EcoreUtil.UsageCrossReferencer$ die Methode $findAll()$, mit deren Hilfe sich solche Referenzen finden lassen. Wir übergeben dieser Methode eine Liste der zu löschenden Knoten und deren Unterknoten sowie die *Resource*, in der diese sich befinden. Daraufhin bekommen wir eine Map zurück, in der sich zu jedem dieser Knoten eine Liste von Settings befindet:

```
Map test = EcoreUtil.UsageCrossReferencer.findAll
  (col, this.deleted.eResource());
```

Jedes Setting enthält den Ausgangspunkt sowie die $EReference$, die auf den gelöschten Knoten zeigt. Da damit der Ursprung, die $EReference$ und der (gelöschte) Zielknoten bekannt sind, kann auch die Referenz einfach gelöscht werden:

```
public void restore(){

  Map refMap = this.getReferringObjects();

  //iterate over all deleted objects
  for(Iterator mit = col.iterator();mit.hasNext();){
    //deleted object - target of reference
    EObject del = (EObject)mit.next();
    //list of referece settings for current object
    Collection refs = (Collection) refMap.get(del);
```

```
if(refs!=null){

  //iterate over reference settings
  for(Iterator it = refs.iterator(); it.hasNext();){
    EStructuralFeature.Setting set =
      (EStructuralFeature.Setting)it.next();

    //source of reference
    EObject refSource = set.getEObject();
    //reference
    EReference ref =
      (EReference)set.getEStructuralFeature();

    //unset reference
    if(ref.isMany()){
      ((EList)refSource.eGet(ref)).remove(del);
    }
    else refSource.eUnset(ref);
    ...
```

Weiterhin kann es sein, dass durch eine Regel ein Objekt erzeugt wird, ohne dass es durch eine Containment-Kante in die EMF-Baumstruktur integriert wird. Werden dabei zusätzlich noch Referenzen von anderen Objekten auf dieses gesetzt, kann auch dies eine Inkonsistenz verursachen. Daher prüfen wir, bevor wir in der Instanz ein Objekt erzeugen, ob gleichzeitig auch eine Containment-Kante gesetzt wird - falls nicht, wird das Objekt gar nicht erst generiert:

```
//Typ des neuen EObjects
EClass newClass =
  (EClass) emfModel.getEClassifier(node.getType().getName());
boolean isContained = false;
```

```
//Kanten in AGG - potentielle Containments
Enumeration incomingArcs = node.getIncomingArcs();
while (incomingArcs.hasMoreElements()) {
  Arc current = (Arc) incomingArcs.nextElement();
  Node source = (Node)current.getSource();
  if(aggToEmf.containsKey(source)){
    EObject emfSource = (EObject)aggToEmf.get(source);
    EList refs = emfSource.eClass().getEAllReferences();
    for(Iterator it = refs.iterator();it.hasNext();){
      EReference ref = (EReference)it.next();

      //Test ob Kante einer Containment-Referenz entspricht
      if(ref.isContainment() && ref.getName().equals
        (current.getType().getStringRepr())){
        isContained = true;
      }
    }
  }
  //vorzeitiger Abbruch der Suche falls Containment gefunden
  if (isContained)
    break;
  }
}
```

5.2 Erweiterte Funktionalität

Neben den Methoden, die direkt zur Ausführung einer Regel oder einer Transformation benötigt werden, gibt es noch weitere, die nicht zur Kernfunktionalität des *Interpreters* gehören, diesen aber in wichtigen Aufgaben unterstützen. Dazu gehören Methoden zur Konvertierung, sowie zum Laden und Speichern.

5.2.1 Konverter EPackage → Typgraph

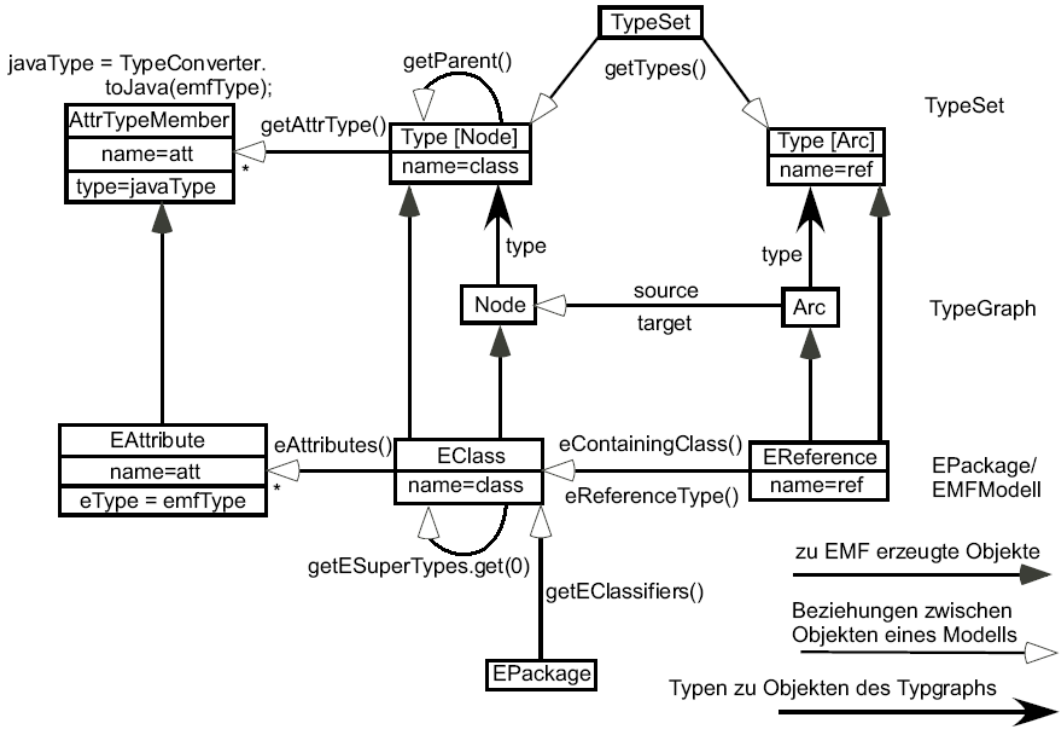

Abbildung 5.3: Mapping von EPackage auf einen AGG-Typgraphen

Durch diesen Konverter kann aus einem *EPackage* eine Grammatik erzeugt werden, die zunächst nur einen Typgraphen enthält. In dieser können dann Regeln erstellt werden , die später auf eine Modellinstanz angewendet werden sollen.
Dies geschieht in der Methode *createTypeGraphFromEPackage*(). Zuerst wird mit *getEClassifiers*() eine Liste aller *EClassifier* (*EClass* und *EDataType*) aus einem *EPackage* gelesen:

```
List l = ePackage.getEClassifiers();

Type newType = ts.createType();
newType.setStringRepr(currentClass.getName());
Node n = tg.createNode(newType);
```

```
nodes.put(currentClass, n);

AttrType attr = (AttrType) newType.getAttrType();
List l3 = currentClass.getEAttributes();
...
EAttribute currentAttr = (EAttribute) l3.get(j);
String javaType = TypeConverter.toJava(currentAttr
  .getEType().getName());
attr.addMember(handler, javaType, currentAttr.getName());
```

Zu jedem Element dieser Liste wird dann ein Typ mit dem selben Namen sowie ein Knoten im AGG-Typgraphen erzeugt. Da später auf den Knoten zu einer bestimmten *EClass* zugegriffen werden muss, wird die Beziehung zwischen beiden in der Hashmap *nodes* gespeichert. Weiterhin werden in dem neuen Typ für alle *EAttributes* gleichnamige *AttributeMember* angelegt. Dabei wandelt die Klasse *TypeConverter* die Typen der Attribute in EMF in entsprechende Java-Klassen um.
Um Vererbungskanten in den Typgraph einzufügen, muss die Liste ein zweites Mal durchlaufen werden und zu jeder *EClass* deren Obertyp herausgefunden werden:

```
//direkte Oberklasse
EClass parent = currentClass.getESuperTypes().get(0);

Node parentNode = (Node) nodes.get(parent);
Node childNode = (Node) nodes.get(currentClass);

ts.addInheritanceRelation(childNode.getType(),
  parentNode.getType());
```

Im AGG-Typgraph wird dann zwischen den ensprechenden Typen eine Vererbungskante eingefügt. Gleichzeitig werden dabei alle *EClasses* auf ihre *EReferences* untersucht und zu jeder ein Kantentyp mit dem selben Namen

angelegt. Im Typgraph werden dann *Arc* diesen Typs zwischen den *Nodes* erstellt, welche Ursprung und Ziel der *EReference* darstellen:

```
List l2 = currentClass.getEReferences();
String typeName = currentRef.getName();
//EClass die Ziel der EReference ist
EClass target = (EClass) currentRef.getEType();

Type newType = ts.createType();
newType.setStringRepr(typeName);
Node src = (Node) nodes.get(currentClass);
Node tar = (Node) nodes.get(target);
Arc typeArc = tg.createArc(newType, src, tar);
```

5.2.2 Konverter Grammar → Transformation

Hierbei wandeln wir die Regeln einer AGG-Grammatik in eine äquivalente Instanz unseres Transformationsmodells um. Die Informationen, die im Typgraph enthalten sind, gehen dabei verloren, solange sie nicht direkt die Regeln betreffen. Diese Instanz kann dann im XML-Format abgespeichert werden und zum Beispiel als Input für den Compiler verwendet werden.

5.2.3 Konverter Transformation → Grammar

Umgekehrt können aus einer Instanz des Transformationsmodells wieder die Regeln einer AGG-Grammatik erzeugt werden. Der Typgraph zu dieser Grammatik muss aus dem *EPackage* des Modells erzeugt werden, da eine Transformationsinstanz nicht die nötigen Informationen enthält. Wenn Regeln und Typgraph zueinander passen, kann dann mit dieser Grammtik der *Interpreter* aufgerufen werden.

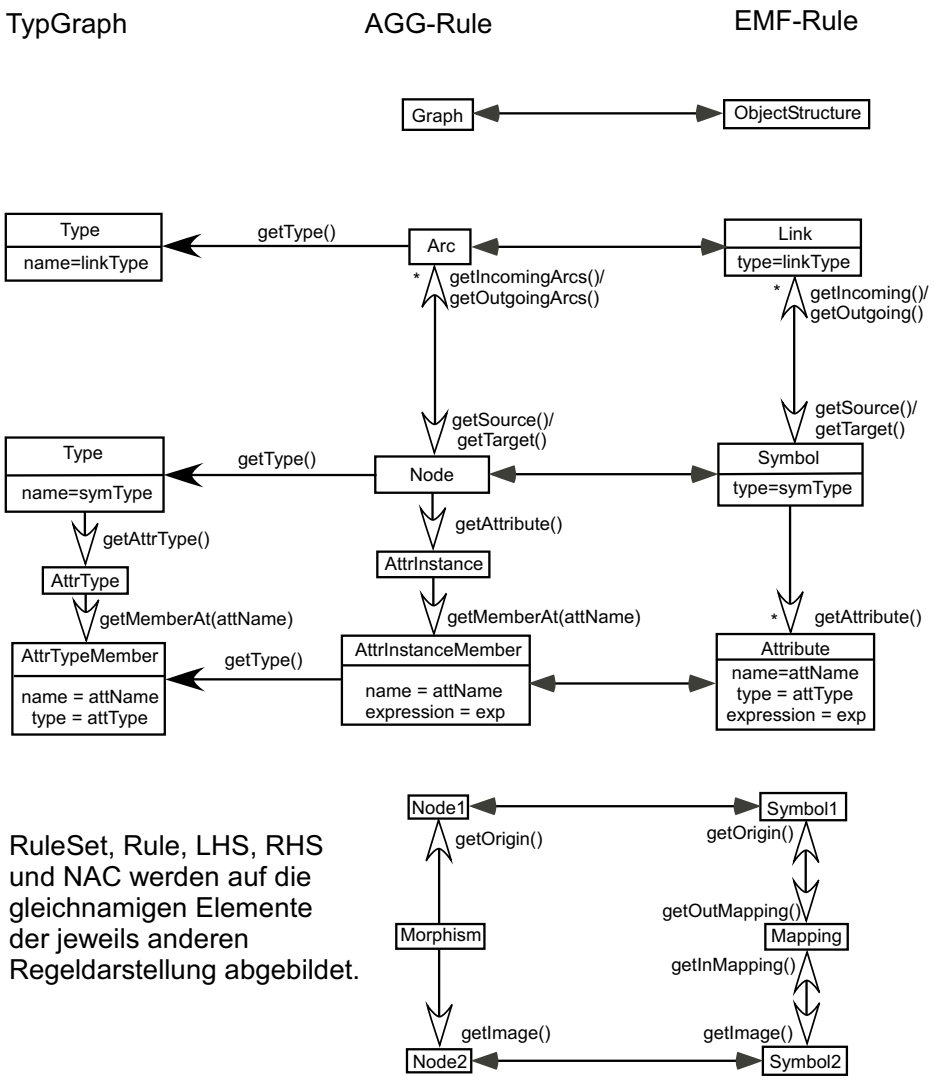

Abbildung 5.4: Mapping zwischen AGG-Regeln und einer Transformationsinstanz

5.3 Wichtige Klassen

Interpreter
Diese Klasse bietet hauptsächlich Interface-Funktionalität nach aussen und enthält eine Instanz von *Grammar*, auf die Regeln angewendet werden können. Im Konstruktor muss eine *EClass* aus dem Modell angegeben werden, damit die Typinformationen daraus gewonnen und eine passende *EFactory* dazu erzeugt werden kann. Die Variante des Konstruktors, die einen Pfadnamen auf eine *.ecore-Datei bekommt, funktioniert leider (noch) nicht, da daraus keine passende *EFactory* erzeugt werden kann, die *EObjects* mit den selben Typen wie in der Modellinstanz generiert.
In dieser Klasse können Regeln sowohl direkt als AGG-Grammatiken oder auch aus einer Transformationsinstanz geladen werden. Ebenso gibt es die Möglichkeit, in beiden Formaten abzuspeichern. Mit *setRoot*() kann das oberste Element einer Modell-Instanz angegeben werden - Regeln werden nur auf alle Elemente unterhalb von diesem angewendet. Und schliesslich erlaubt es die Methode *applyRule*(), eine Regel auszuführen.

Grammar
Während Interpreter Funktionalität nach aussen hin zur Verfügung stellt, wird diese hier implementiert. Die Methode *applyRule*() erlaubt das Ausführen einer Regel während $transform()$ so lange eine Transformation mit allen verfügbaren Regeln durchführt, bis keine Regel mehr anwendbar ist. Weiterhin befinden sich hier alle Methoden, um die Konvertierungen durchzuführen, die im Abschnitt 5.2 erwähnt wurden. Alle weiteren Methoden sind entweder Hilfmethoden bei einer Regelanwendung oder dienen zum Laden bzw Speichern.

Parameter
Diese Klasse bildet eine Datenstruktur, in der Name, Typ und Wert von Input-Parametern in einem Format gespeichert werden, das dem von AGG entspricht und daher leicht übergeben werden kann.

ConsistenceRestorer

Diese Klasse übernimmt die Wiederherstellung der Konsistenz einer EMF-Instanz. Gelöschte Knoten werden mit ihren Unterknoten zunächst einmal gesammelt. Anschliessend werden Referenzen auf diese gesucht und gelöscht. Dies geschieht in der Methode *restore*().

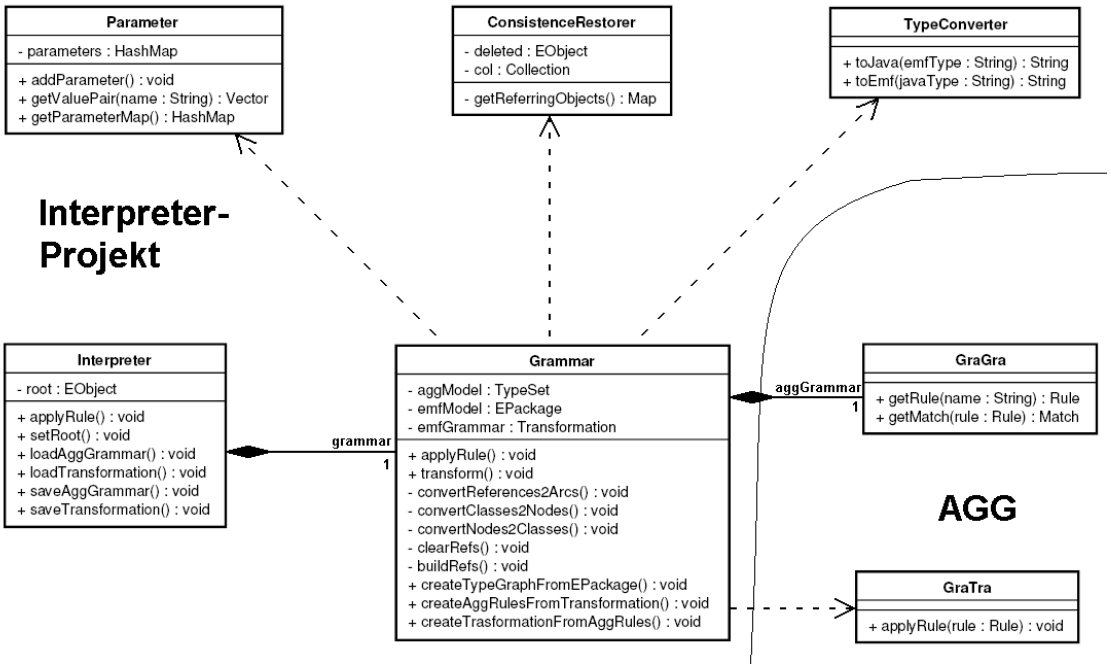

Abbildung 5.5: Klassenstruktur Interpreter

Kapitel 6

Compiler

6.1 Arbeitsweise

Im Gegensatz zum Interpreter wird beim Compiler Java-Code erzeugt, der die Veränderungen direkt auf der Modellinstanz vornimmt. Dadurch fallen die Transformationen von und nach AGG weg, aber AGG-Funktionalität, wie das Finden einen Matches für die LHS, die vom Interpreter verwendet wurde, muss direkt im Compiler implementiert werden.
Da der Compiler in den meisten Fällen mit den konkreten Typen aus dem Modell operiert, müssen dazu diese mit ihren Getter- und Setter-Methoden bekannt sein. Bei der Verwendung des Compilers können zwei Phasen unterschieden werden: die Erzeugung von Code aus den Templates und Data-Klassen mit Hilfe von JET und die Anwendung dieses Codes zur Laufzeit.

6.1.1 Codeerzeugung

Um Java-Code zu unserem Transformationmodell zu erzeugen, benutzen wir JET in Verbindung mit einigen JET-Templates und einer Datenstruktur, welche die Informationen aus der Transformationsinstanz enthält. Für jede der wichtigeren Klassen unseres Transformationsmodells gibt es in dieser eine entsprechende Klasse, die einfachen Zugriff auf deren benötigten Informationen liefert.

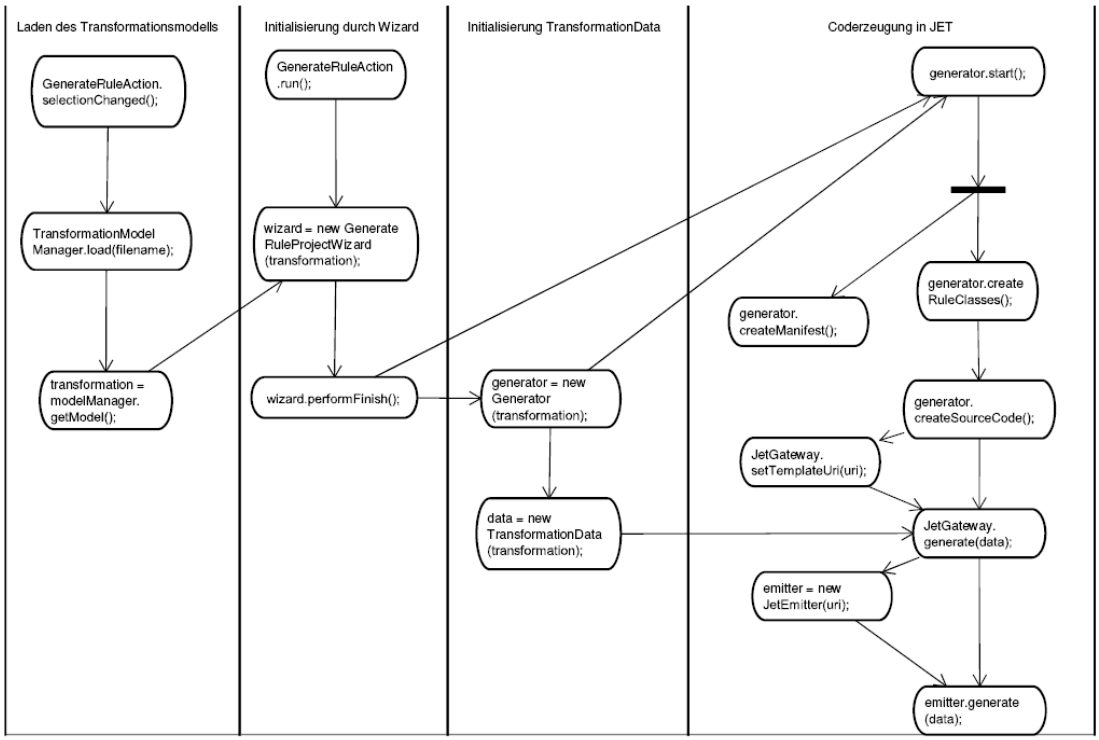

Abbildung 6.1: Code-Erzeugung mit JET

Zunächst wird in *GenerateRuleAction* aus einer entsprechenden *.tfm-Datei eine Instanz unseres Transformationsmodells geladen und gleichzeitig ein $RuleProjectWizard$ gestartet. Dieser fragt in einem Dialog das Projekt ab, welches das zu bearbeitende Modell enthält, sowie nach dem Namen des neuen Projekts, in dem die Regeln erstellt werden sollen. Nachdem der Dialog beendet wurde, wird in der Methode $performFinish()$ der Konstruktor von *Generator* mit der Transformationsinstanz aufgerufen. Dort wird aus dieser eine Instanz unsere Datenstruktur $TransformationData$ mit allen in ihr enthaltenen Klassen initialisiert. Anschliessend wird der *Generator* mit der Methode *start*() gestartet. Dabei erzeugt er in der Workbench ein neues Projekt ($createManifest()$) und generiert anschliessend aus den Templates und dem $TransformationData$ Java-Code. Dies geschieht durch den Aufruf von $generateRuleClasses()$ bzw $createSourceCode()$, und mit Hilfe der Klassen $JetGateway$ und $JetEmitter$. Verweise auf die Templates werden

über deren URIs gegeben, die Datenstruktur $TransformationData$ wird als Parameter an die Methoden $generate(data)$ aus $JetGateway$ bzw $JetEmitter$ weitergegeben. In den Templates befinden sich, neben festen Code-Fragmenten, noch Platzhalter, an denen dann Informationen aus der Datenstruktur eingefügt werden.

6.1.2 Verhalten zur Laufzeit

Wenn die Coderzeugung erfolgreich war, befindet sich in der Runtime-Workbench ein neues Projekt, das den erzeugten Code enthält. Dieser umfasst jeweils eine Regel- und eine Wrapper-Klasse für jede Regel des Transformationsmodells.
Ausserdem werden für den Algorithmus zum Suchen von Matches noch einige zusätzliche Klassen erzeugt. Vor einer Regelanwendung muss zunächst einmal eine Instanz der entsprechenden Regelklasse erzeugt werden. Diese enthält Getter und Setter, um Input-Parameter und Matches vorzugeben.
Wenn man dann eine Regel ausführt, wird zunächst einmal der Match vervollständigt und dabei die NACs überprüft. Hierzu werden dem Matchfinder in der Wrapper-Klasse, die zur Regel gehört, die nötigen Informationen über die Struktur der linken Regelseite gegeben. Wenn ein Match gefunden wurde, wird die Regel angewendet, indem direkt auf dessen EMF-Objekten die entsprechenden Getter- und Setter-Methoden aufgerufen werden.

6.2 Designentscheidungen

Die Verwendung der Getter und Setter aus den Modellklassen bringt neben Vorteilen bei der Laufzeit einige Nachteile: So sind zum Beispiel nicht immer alle Referenzen eines Modells direkt veränderbar - das heisst, zu einer Referenz wird nicht in jedem Fall auch ein Setter erzeugt. Wenn man den Wert einer solchen Referenz ändern will, ist es zwar möglich im Editor eine Regel zu erzeugen, die dies tut, es wird jedoch zu ihr fehlerhafter Java-Code erzeugt.
Eine Alternative wäre die Verwendung von generischen Methoden, wie in der

Variante des Compilers für das Ecore-Modell. Dies hat jedoch die Nachteil, dass die Regel zwar ausgeführt, die Referenz aber trotzdem nicht gesetzt wird und eine leicht höhere Laufzeit des erzeugten Codes. Oder man könnte die Verwendung solcher Referenzen im Regeleditor generell verbieten - der Nachteil hiervon ist ein leicht höherer Aufwand bei der Matchsuche, da diese Referenzen oft weniger Ziele haben als ihre allgemeineren, veränderbaren Varianten.

Bisher wird in den Wrapper-Klassen zu regulären Modellen die Klasseninformationen der jeweiligen Symbols aus der *EFactory* des Modells geholt:

```
private [ModelName]Factory factory = new [ModelName]FactoryImpl();
...
EObject source = factory.create[symbolType]();
Vector<EObject> domain = typeToDomain.get(source.eClass());
```

Da dies für abstrakte *EClasses* nicht funktioniert, ist die Variante wie in der Wrapper-Klasse zum Ecore-Model vorteilhafter und wird vermutlich auch demnächst als Lösung für dieses Problem gewählt:

```
public [RuleName]Wrapper(EObject root) {
[ModelName]Package pack =
([ModelName]Package)root.eClass().getEPackage();
...
EClass source = (EClass)pack.getEClassifier("[symbolType]");
Vector<EObject>domain = typeToDomain.get(source);
```

6.3 Wichtige Komponenten

6.3.1 Generator-Klassen

Um den Compiler zu starten, brauchen wir zunächst einige Klassen, welche die Generiereung von Java-Code mit Hilfe von JET steuern. Im folgenden werden die wichtigsten Klassen mit ihren Methoden kurz erklärt:

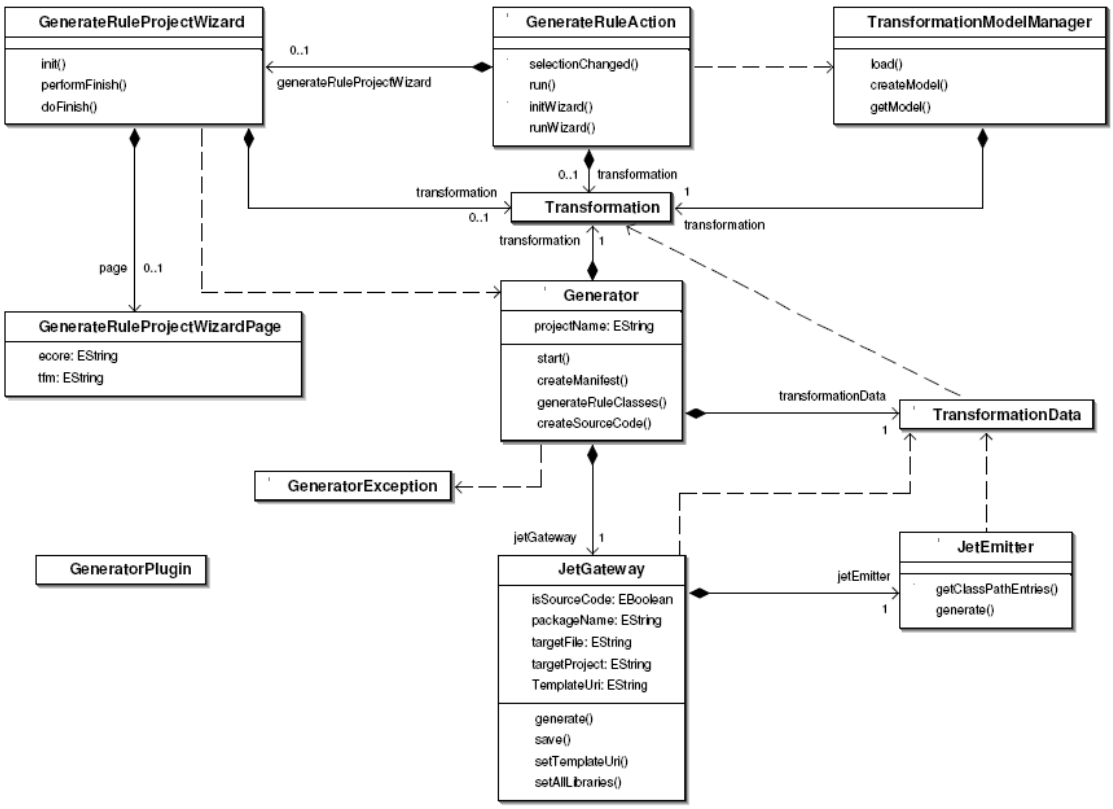

Abbildung 6.2: Klassendiagramm Generator und Zusammenhang mit Transformation

GenerateRuleAction

Diese Action wird ausgelöst, wenn in der Runtime-Workbench eine *.tfm-Datei ausgewählt und im Menü 'Generate Rule Classes' angeklickt wird. Daraufhin

werden die Methoden *selectionChanged*() und *run*() gestartet. Erstere lädt mithilfe des $TransformationModelManagers$ eine $Transformation$. Dazu werden in jener Klasse mit der Methode *load*() aus einer Datei eine $Transformation$ geladen und diese mit $getModel()$ zurückgegeben.
Die Methode *run*() initialisiert mit $initWizard$ einen $GenerateRuleProjectWizard$, wobei diesem die $Transformation$ übergeben wird. Durch den Aufruf von $runWizard$ wird dieser dann anschliessend gestartet.

GenerateRuleProjectWizard

Der $GenerateRuleProjectWizard$ startet zunächst einen Dialog - eine Instanz von $GenerateRuleProjectWizardPage$. Sobald dieser durch einen Klick auf den Finish-Button beendet wurde, wird dadurch die Methode $performFinish()$, und durch diese $doFinish()$, aufgerufen. In $doFinish()$ wird ein *Generator* erzeugt und gestartet.

Generator

In diese Klasse beginnt die Codeerzeugung, nachdem zuvor alle nötigen Informationen zusammengetragen wurden. Im Konstruktor wird zuerst aus der Transformationsinstanz ein *JetGateway* initialisiert und ein $TransformationData$ erzeugt - unsere Datenstruktur, die an JET als Parameter bei der Codegenerung übergeben wird.
Die Methode *start*() ruft $createManifest()$ und $generateRuleClasses()$ auf. Die erste erzeugt ein neues Projekt mit den nötigen Importen für den generierten Code, imdem sie die Templates *build.properties.xmljet* und *plugin.xml.xmljet* übersetzt. Mit $generateRuleClasses()$ werden die Klassen zu einer $Transformation$ erzeugt - dazu wird $createSourceCode()$ für alle Templates aufgerufen. Diese übergibt die nötigen Informationen an das *JetGateway* und ruft dann $generate(transformationData)$ auf diesem auf.

JetGateway

Diese Klasse steht direkt mit der JET-Klasse *JetEmitter* in Verbindung. Im Konstruktor werden durch die Methode *setAllLibraries*() alle Libraries festgelegt, die bei der Codeerzeugung verwendet werden können. Durch *setTemplateUri*() wird der Pfad des verwendeten Templates festgelegt. Die Methode *generate*(*transformationData*) erzeugt einen *JetEmitter* und übergibt diesem den Pfad zum Template. Dann wird *JetEmitter.generate*(*transformationData*) aufgerufen und der Code, der dadurch erzeugt wurde, mithilfe der Methode *save*() unter dem gewünschten Klassennamen abgespeichert.

6.3.2 Templates

Diese Dateien bilden die Vorlagen für die zu erzeugenden Klassen, indem sie die Compilierung steuern, Code-Fragmente liefern und festlegen an welchen Stellen bestimmte Informationen aus den Data-Klassen eingefügt werden sollen. Sie befinden sich in den Packages *templates* (Templates zum Erzeugen des Regelprojekts) beziehungsweise *templates.rules* (Templates für Sourcecode).
Die Regel- und Wrapper-Templates werden werden so oft angewendet, dass zu jeder Regel jeweils eine Regel- und eine Wrapper-Klasse erzeugt wird. Dabei bekommen sie aus *TransformationData* mit *getRule*() jeweils das *RuleData*, das der aktuellen Regel entspricht.
Alle anderen Templates werden ingesamt nur einmal benutzt, um Code zu erzeugen. Die aus den Templates generierten Java-Klassen haben im wesentlichen den selben Namen wie diese, nur bei Regel- und Wrapper-Klassen besteht der erste Teil des Klassennamens aus dem Namen der Regel. Die Templates *EcoreRule.javajet* und *EcoreWrapper.javajet* sind speziell an die Besonderheiten des EcoreModells angepasst und müssen anstelle der normalen Templates verwendet werden, wenn Regeln zum Ecore-Modell compiliert werden sollen.

Übersicht Templates

- *build.properties.xmljet* → *build.properties* legt Eigenschaften des neuen Regelprojekts fest
- *plugin.xml.xmljet* → *plugin.xml* legt Eigenschaften des neuen Regelprojekts fest
- $TransformationInterface.javajet \rightarrow TransformationInterface$
- $Parameter.javajet \rightarrow Parameter$: Sammlung aller Input-Parameter (identisch zu gleichnamiger Klasse beim Interpreter)
- *AbstractRule.javajet* → *AbstractRule* abstrakte Oberklasse aller Regeln, enthält Methoden zum Löschen von *EObjects* und Konsistenzwiederherstellung
- *Rule.javajet* → [Regelname]*Rule* Template zum Erzeugen der Regelklassen (für reguläres EMF-Modell)
- *EcoreRule.javajet* → [Regelname]*Rule* Template zum Erzeugen der Regelklassen (für Ecore-Modell)
- $Wrapper.javajet \rightarrow$ [Regelname]$Wapper$ jeweils ein Wrapper für jede Regelklasse (für reguläres EMF-Modell)
- $EcoreWrapper.javajet \rightarrow$ [Regelname]$Wapper$ jeweils ein Wrapper für jede Regelklasse (für Ecore-Modell)
- $Matchfinder.javajet \rightarrow Matchfinder$ zentrale Klasse des Matchsuch-Algorithmus
- $Variable.javajet \rightarrow Variable$
- *Query.java* → *Query* gemeinsame Oberklasse aller anderen Query-Klassen
- $TargetQuery.javajet \rightarrow TargetQuery$, überprüft das Ziel einer Referenz in der Instanz

- *SourceQuery.javajet* → *SourceQuery*, überprüft die Quelle einer Referenz in der Instanz
- *InjectivityQuery.javajet* → *InjectivityQuery*, verhindert die gleiche Belegung von mehreren *Variablen* (Injektivität)
- *TypeQuery.javajet* → *TypeQuery*, stellt sicher, dass Belegung einen bestimmten Untertyp hat (nur für NACs)
- *VariableQuery.javajet* → *VariableQuery*, fordert die gleiche Belegung zweier Attribute

6.3.3 Data-Klassen

Diese Klassen befinden sich im Package *util* und bilden das Pendant zu den wichtigsten Klassen des Transformationsmodells. Sie werden anhand einer Instanz dieses Modells initialisiert, wobei jedes Objekt einer Data-Klasse (mindestens) eine Referenz auf ein Objekt der Transformations-Instanz hat. Weiterhin stellt eine Data-Klasse noch die Methoden zur Verfügung, um die benötigten Informationen aus dieser zu erhalten. Im wesentlichen sind dies Getter, um die Attribute und Referenzen des entsprechenden Elements aus dem Transformationsmodell abzufragen und Methoden, die bei der Initialisierung aufgerufen werden, damit die Data-Klassen die selben Referenzen aufeinander haben wie in der Transformationsinstanz. Ausserdem gibt es noch Methoden um Regeln zu analysieren, zum Beispiel um festzustellen welche Symbols durch die Regel erzeugt oder gelöscht werden.

Im Folgenden gehen wir auf die Initialisierung der Data-Klassen und auf deren zusätzlichen Methoden zur Regelanalyse ein.
Die Initialisierung beginnt im Konstruktor von *Generator.java* - der Parameter *transformation*, das oberste Element einer Transformationsinstanz, wird benutzt um die Data-Klassen zu initialisieren:

```
this.transformationData = new TransformationData(transformation);
```

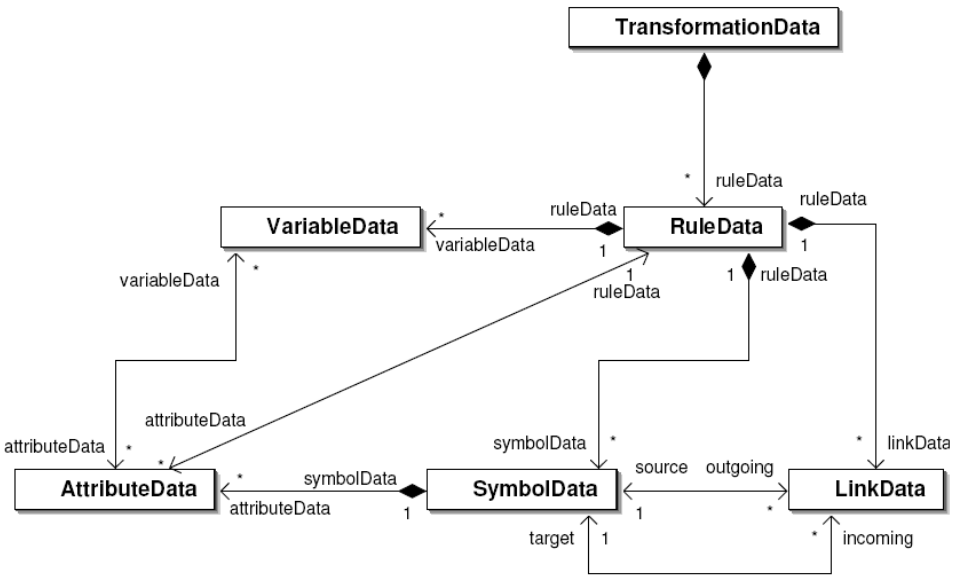

Abbildung 6.3: Klassendiagramm der Data-Klassen

TransformationData

Im Konstruktor von *TransformationData* wird die Transformationsinstanz als Attribut des neuen Objekts gespeichert und anschliessend werden in *initRules*() nacheinander für jede Regel ein Objekt vom Typ *RuleData* erzeugt.

```
public TransformationData(Transformation transformation){
  this.transformation = transformation;
  this.initRules();
}
```

Im Attribut *ruleIndex* wird der Index der aktuellen Regel abgespeichert, diese kann mit *getRule*() ausgegeben werden. Dies ist für die Regel- und Wrapper-Templates wichtig, damit diese jedes mal das aktuelle *RuleData*

bekommen. Ausserdem gibt es noch weitere Hilfsmethoden: *getRuleCount*() liefert die Anzahl der der Regeln, *getMaxLayer*() die Anzahl der Layer und *getLayerRule*() gibt, mit einem Parameter aufgerufen, alle Regeln eines Layers oder mit zweien eine bestimmte Regel aus einem Layer zurück.

RuleData

Im Konstruktor von *RuleData* werden auch gleichzeitig *VariableData* und *SymbolData* initialisiert. Für jedes Symbol der LHS und alle davon gemappten Symbols aus der RHS und den NACs wird jeweils nur ein *SymbolData* erzeugt. Dies geschieht durch die Methode *initLHSSymbols*(). Weiterhin wird zu jedem Symbol aus der RHS oder den NACs, das nicht von der LHS aus gemappt wird, mit *initNonLHSSymbols*() ebenfalls ein *SymbolData* erzeugt. Wenn alle *SymbolData* initialisiert wurden, werden noch die target- bzw. icoming-Referenzen zwischen den *SymbolData* und *LinkData* gesetzt. Weiterhin werden aus den *SymbolData* alle *AttributeData* gesammelt und unter *attributes* in *RuleData* gespeichert.
Mit dem Methoden *getCreatedSymbols*() und *getDeletedSymbols*() lassen sich alle erzeugten beziehungsweise gelöschten Symbols einer Regel ausgeben. Um die Links zu analysieren, gibt es analoge Methoden. Weiterhin lassen sich alle Symbols, Links oder Attribute einer bestimmten Regelseite oder NAC zurückgeben. Eine Liste aller Symbols der LHS, sortiert nach Gewicht, liefert die Methode *getSortedLHSSymbols*(), welche durch die Hilfsmethode *sortSymbols*() unterstützt wird. Mit *getNACCount*() bekommt man die Anzahl der NACs einer Regel, mit *getParams*() deren Input-Parameter.

SymbolData

Bei der Erzeugung eines *SymbolData* werden gleichzeitig dazu alle *AttributeData* initialisiert und für alle Links, die von diesem ausgehen, *LinkData*-Objekte erzeugt. Da nicht sicher ist, ob zu dem Symbol, auf das ein Link zeigt, schon ein *SymbolData* erzeugt wurde, kann zunächst nur die source- bzw outgoing-Referenz zwischen *SymbolData* und *LinkData* gesetzt

werden.
Ein *SymbolData* kann mehrere Symbols repräsentieren - ein Symbol aus der LHS und mehrere Symbols aus der RHS oder den NACs, die von diesem gemappt werden. Das erste Symbol wird dabei in dem Attribut *symbol*, alle weiteren unter *images* gespeichert.
Mit *createVariableName*() wird jedem *SymbolData* ein eindeutiger String zugeordnet - der Name der Variablen, die diesem *SymbolData* in der generierten Regelklasse entspricht. Dieser kann zum Beispiel von den Templates aus mit *getVariableName*() abgefragt werden. Die Methode *isConsistent*() überprüft für erzeugte Symbols, ob diese auch in der Containment-Struktur des EMF-Baums integriert sind. Falls dies nicht der Fall ist, wird im generierten Regel-Code das Objekt zu diesem Symbol gar nicht erst erzeugt. Weiterhin kann man mit *isCreated*() oder *isDeleted*() zurückgeben lassen, ob das Symbol durch die Regel erzeugt oder gelöscht wird. Ausserdem lässt sich überprüfen, ob das Symbol oder ein von diesem gemapptes in einer bestimmten Regelseite oder NAC vorkommt.

LinkData

Bei der Erzeugung eines *LinkData* müssen keine weiteren Objekte mehr initialisiert werden. Lediglich die target- beziehungsweise incoming-Referenz muss noch gesetzt werden. Dazu wird in *getTarget*() zunächst einmal überprüft, ob die Referenz schon gesetzt wurde. Falls dies noch nicht geschehen ist, wird mit Hilfe der Methode *findTarget*() aus *RuleData* das passende *SymbolData* gesucht und die Referenzen hergestellt. Die Methode *getTarget*() wird nach der Initialisierung aller *SymbolData* für jedes *LinkData* einmal aufgerufen. Dies geschieht in der Methode *initSymbols* von *RuleData*.

```
public SymbolData getTarget(){
  if(this.target == null){
    this.target = this.rule.findTarget(link.getTarget());
      if(this.target!=null){
```

```
        this.target.addIncoming(this);
      }
    }
  return this.target;
}
```

Wie auch schon bei *SymbolData* gibt es auch in dieser Klasse Methoden um festzustellen ob ein Link durch die Regel erzeugt beziehungsweise gelöscht wird und ob er in einer bestimmten Regelseite oder NAC vorkommt.

AttributeData

Bei der Initialisierung eines *AttributeData* müssen nur noch die Referenzen zwischen diesem und den $VariableData$ gesetzt werden. Dies geschieht in der Methode $initVars()$, wobei die passende $VariableData$ über deren eindeutigen Namen identifiziert wird.
Auch hier gibt es Methoden, um festzustellen, ob ein *AttributeData* in einer bestimmten Regelseite oder NAC vorkommt.

6.4 Erzeugter Code

6.4.1 TransformationInterface

Diese Klasse stellt nach aussen hin die selbe Funktionalität wie der Interpreter zur Verfügung. So gibt es auch hier eine Methode $applyRule()$, die mit den selben Parametern wie die gleichnamige Methode des $Interpreters$ aufgerufen wird und ein vergleichbares Ergebnis unter Anwendung des Compilers liefert. Die Methode $transform()$ wendet alle Regeln so lange an, bis keine Regel mehr anwendbar ist. Dabei werden, wie auch beim Interpreter, die Layer der Regeln mit berücksichtigt.

6.4.2 Regel-Klassen

Für jede Regel aus dem Transformation-Modell wird eine eigene Regel-Klasse erzeugt. Diese importiert die verwendeten Modell-Klassen und erlaubt das Ausführen[execute()], Rückgängig-Machen [*undo*()] sowie danach die erneute Anwendung [*redo*()] einer Regel. In einer Instanz der Regelklasse kann *execute*() nur enmal aufgerufen werden, die Regel also nur einmal angewendet werden. *undo*() und *redo*() bedingen sich jeweils gegenseitig und können beliebig oft immer abwechselnd ausgerufen werden.

In einer Regelklasse können über Setter Input-Parameter gesetzt oder Match-Vorgaben an die Regel gemacht werden. Die Anwendung einer Regel erfolgt schliesslich über einen Aufruf der Methode *execute*:

```
public boolean execute() {

  //initialisation wrapper
  DeleteAnnotationWrapper wrapper =
    new DeleteAnnotationWrapper(this.root);
  //set (partial) match
  wrapper.instanciateVariables(getLHS());
  //set and apply input parameters
  wrapper.setC(this.c);
  wrapper.reduceDomains();

  //get match
  Vector<EObject> match = wrapper.getSolution(-1);
  if (match == null){//no match found
    return false;
  }else{//match found
    setLHS(match);
  }

  //check if executable and ensure
```

```
  //that rule is applied at most once
  if (canExecute() && !isExecuted) {

    //init variables from LHS
    initVariables();

    //objects
    createNewObjects();
    deleteOldObjects();

    //references
    deleteOldReferences();
    setNewReferences();

    //attributes
    setExpressions();

    //set execution flag
    isExecuted = true;
    return true;
  }

  return false;
}
```

Diese ist in jeder Regelklasse fast identisch - Unterschiede bestehen nur beim Typ des zugehörigen Wrappers sowie bei der Anzahl und den Namen der Input-Parameter, die an diesen übergeben werden.
Vor der Regelanwendung wird zunächst mit Hilfe der Wrapperklasse und des Matchfinders der Match vervollständigt. Dann wird durch die Methode *canExecute*() geprüft, ob die Regel anwendbar ist (zusätzlich nur noch Dangling Condition).
Falls dies der Fall ist, werden die Methoden aufgerufen, die die Regelanwendung

auf der EMF-Instanz simulieren. Deren Inhalt ist für jede Regel unterschiedlich - wenn eine Regel eine bestimmte Art der Veränderung nicht durchführt, ist die entsprechende Methode leer. Zuerst werden dabei aus dem verwendeten Match alle Variablen, die in der LHS vorkommen mit den dazugehörigen Attributwerten initialisiert. Anschliessend werden Objekte, wie in der Regel beschrieben, erzeugt bzw. gelöscht. Nach dem Löschen wird die Konsistenz innerhalb der Modellinstanz wiederhergestellt, indem alle Referenzen auf gelöschte Objekte und deren Unterknoten gelöscht werden. Dies geschieht mithilfe der Methode $deleteTreeRefs()$ aus der Klasse $AbstractRule$. Danach werden die Referenzen wie in der Regel gelöscht bzw neu gesetzt. Zum Schluss werden durch die Methode $setExpresssions()$ Attributveränderungen durchgeführt.

Besonderheiten bei der Verwendung des Ecore-Modells

Im Gegensatz zu normalen Regelklassen werden Referenzen nicht mit den Gettern und Settern aus dem Ecore-Modell gesetzt oder gelöscht sondern mit Hilfe des generischen Codes der *EObjects*. Der Grund hierfür war, dass es im Ecore-Modell viele unveränderbare Referenzen gibt und diese, wenn sie in einer Regel auftauchen, Syntax-Fehler im generierten Code verursachen.

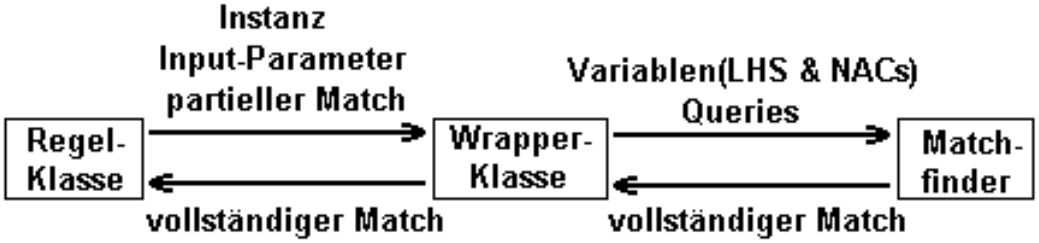

Abbildung 6.4: Klassen im Laufzeit-Code

6.4.3 Wrapper-Klassen

Zu jeder Regel-Klasse gibt es noch eine Wrapper-Klasse, die als Interface zwischen Regel-Klasse und Matchfinder fungiert. Zunächst wird in der Methode $fillTypeMap()$ die Instanz durchlaufen und in einer HashMap jedem gefunden Typen ein Vector aller *EObjects* diesen Typs zugeordet. Jedes *EObject* kommt dabei auch gleichzeitig in allen Listen seiner Obertypen vor.
Anschliessend werden zu jedem Symbol der LHS und der NACs eine $Variable$ und die dazugehörigen *Queries* erzeugt. Die *domain* (Liste der möglichen Belegungen) dieser $Variablen$ enthält zunächst alle Objekte aus der Instanz, die den Typ haben, der beim Symbol angegeben wurde (oder einen Untertyp davon). Dazu wird die HashMap, die bei der Methode $fillTypeMap()$ angelegt wurde, genutzt, um alle *EObjects* eines bestimmten Typs zu finden. Anhand der Attributbelegungen, die von der Regel gefordert werden, sowie durch die vorgegebenen Input-Parameter wird die *domain* dann in der Methode $reduceDomain()$ weiter eingeschränkt.
Die *EObjects*, die sich als Variablenbelegungen in den *domains* befinden, sind in der Reihenfolge, wie sie von $fillTypeMap()$ gefunden werden, geordnet - also *EObjects*, die sich weiter oben im EMF-Baum befinden, sind auch in den *domains* eher weiter vorn.
Der Matchfinder wird dann in der Methode $getSolution()$ schliesslich mit diesen $Variablen$ initialisiert und gestartet. Für die LHS sind die $Variablen$ nach Gewicht geordnet, wobei sich dieses aus $Anzahl\_ausgehender\_Links - Anzahl\_eingehender\_Links$ berechnet. Der Grund hierfür ist, dass das Ziel einer Referenz wesentlich leichter zu überprüfen ist als ihre Quelle. Ausserdem ist für $Variablen$ mit vielen *Queries* die Wahrscheinlichkeit grösser, dass Konflikte schon direkt bei der Belegung festgestellt werden und weniger Backtracking nötig ist. Bei den NAC-$Variablen$ kommen zuerst diejenigen, die schon in der LHS vorkommen, da diese eindeutig mit Werten aus den gefundenen Matches belegt werden können. Anschliessend diejenigen, die nur in der NAC vorkommen, in der selben Reihenfolge wie dort angegeben.

Besonderheiten bei der Verwendung des Ecore-Modells

In den Wrapper-Klassen zum Ecore-Modell werden Typinformationen anstelle aus einer Factory direkt aus dem Package geholt. Für abstrakte *EClasses* ist dies nicht anders möglich, da es in einer Factory keine create-Methoden für diese gibt.

6.4.4 Variable

Variablen haben neben dem Attribut *domain* eine weitere Liste, in der die Werte aus der *domain*-Liste enthalten sind, die anhand der bisher bekannten Belegung noch möglich sind - das Attribut *dynamicDomain*.
Die Methode *instanciate*() belegt die *Variable* mit dem ersten gültigen Wert aus der *dynamicDomain*. Dabei werden ebenfalls die *Queries*, die in dieser *Variablen* enthalten sind, überprüft. Falls dabei durch die aktuelle Belegung ein Konflikt (für eine andere Variable ist keine Belegung mehr möglich) verursacht wird, wird diese aufgehoben und durch die Methode *nextInstance*() die *Variable* mit dem nächsten Wert aus der *dynamicDomain* belegt.

6.4.5 Queries

Queries überprüfen die Constraints, die mit einer *Variablen* zusammenhängen und bestehen in der Regel nur zwischen zwei *Variablen* - die gerade belegte (Referenz *creator*) und eine abhängige zu überprüfende *Variable* (Referenz *target*), die sich in der Reihenfolge der *Variablen* weiter hinten befindet.
Durch die *Query* wird die *dynamicDomain* der *Variablen target* gegebenenfalls eingeschränkt. Falls für diese gar keine Belegung mehr möglich ist, deutet dies auf einen Konflikt hin, der durch die Belegung von *creator* verursacht wurde.

```
//InjectivityQuery - eval()
public boolean eval(){
```

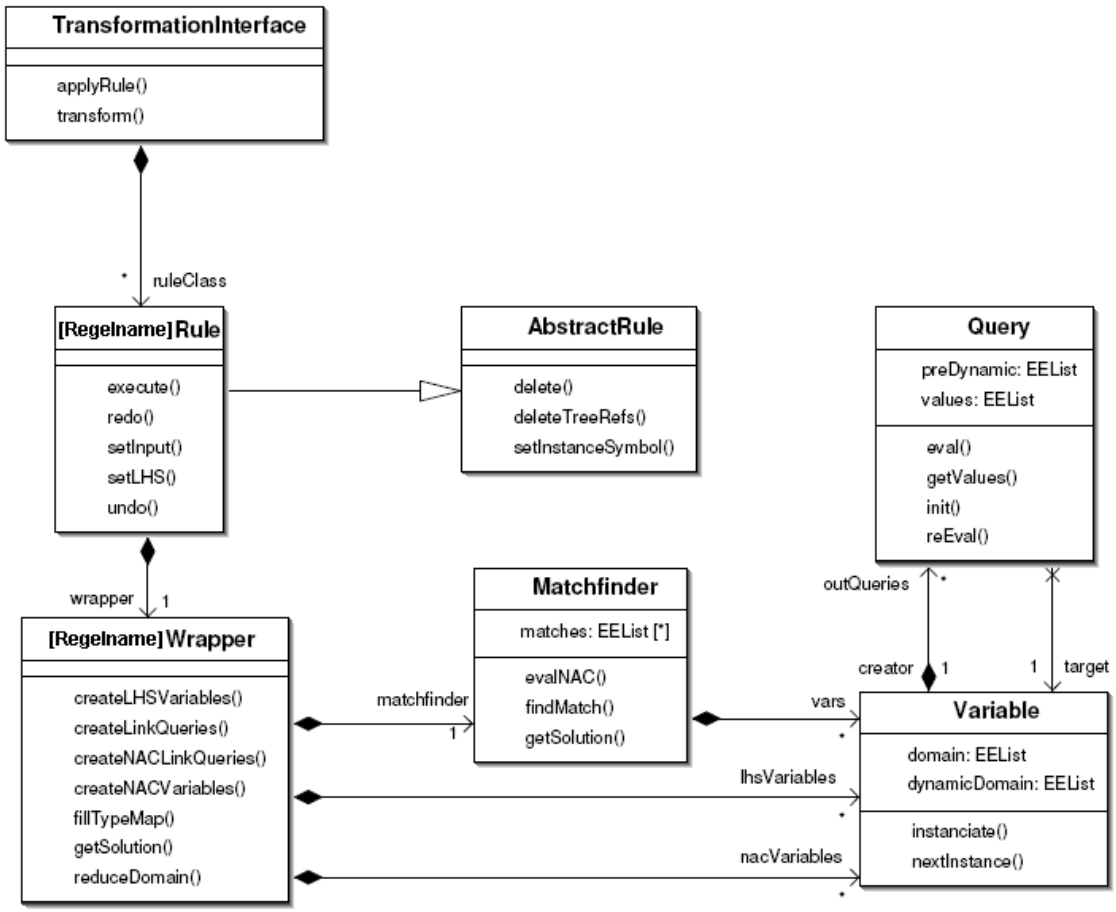

Abbildung 6.5: Klassen im Laufzeit-Code

```
init();
values.addAll(this.preDynamic);
//Wert von creator nicht fuer target zulassen (Injektivitaet)
values.remove(creator.getInstanceValue());
if(values.size()>0){
  target.setDynamicDomain(values);
  return true;
}
return false; //values leer - Konflikt
```

```
}
```

In der Methode *init*() zunächst die *dynamicDomain* von *target* gespeichert, damit der Vorzustand wiederhergestellt werden kann, falls sich die Belegung von *creator* später als falsch herausstellt. In *eval*() werden dann aus der *dynamicDomain* der $target - Variablen$ die Werte entfernt, die wegen der aktuellen Belegung von *creator* das Constraint verletzen. Falls dadurch für *target* keine Belegung mehr möglich ist, da deren *dynamicDomain* leer ist, gibt *eval*() als Rückgabewert $false$ zurück - dies signalisiert einen Konflikt und führt dazu, dass die aktuelle Belegung von *creator* verworfen wird und dieser der nächste Wert zugewiesen wird.
Durch die $Source-$ und $TargetQuery$ wird sichergestellt, dass innerhalb des Matches auch die geforderten Referenzen zwischen den $EObjekts$ bestehen. Damit ein Match immer injektiv ist, werden zwischen Variablen gleichen Typs noch $InjectivityQueries$ erzeugt. Variablen für die NACs können noch $TypeQueries$ enthalten, falls ein Symbol in der NAC einen anderen Typ hat, als in der LHS.

6.4.6 Matchfinder

Im $Matchfinder$ läuft ein Constraint-Solving-Algorithmus ähnlich dem im AGG [19], um konsistente Belegungen für die Knoten der linken Regelseite zu finden. Da das Problem der Graphisomorphie NP-vollständig ist, ist der Zeitaufwand hierfür im schlimmsten Falle exponentiell. Daher haben wir versucht diesen soweit es geht zu minimieren und dazu den Backtracking-Algoritmus für CSPs verwendet [20]. Dieser basiert stark auf Tiefensuche, wobei die zu belegenden $Variablen$ die Tiefe und die *domains* die Breite des Suchbaums festlegen. Im Gegensatz zum Standard-Algorithmus verwenden wir in den *Queries* das sogenannte $forward\ checking$, um Konflikte in den Belegungen so früh wie möglich festzustellen, ohne dazu den gesamten Suchbaum ablaufen zu müssen.
Alle $Variablen$ werden nacheinander belegt, wobei in jedem Schritt die Verträglichkeit der Belegungen über die *Queries* sichergestellt wird - in den

dynamicDomains befinden sich nur noch Werte, die mit bereits belegten *Variablen* verträglich sind. Durch die *Queries* der aktuellen *Variablen* wird überprüft, ob deren Belegung einen Konflikt mit den folgenden unbelegten *Variablen* verursacht.
Wenn für die aktuelle *Variable* alle Belegungen durchprobiert und verworfen wurden, ist sicher, dass durch die Belegung einer der vorangegangenen *Variablen* ein Konflikt verursacht wird. Daher wird die Belegung der aktuellen Variablen aufgehoben und für die vorangegangene *Variable* der nächste Wert gewählt. Dies wird so lange nach oben im Suchbaum fortgesetzt, bis die *Variable*, die den Konflikt verursacht hat, einen neuen Wert bekommen hat.
Sobald alle *Variablen* belegt wurden, ist ein Match gefunden worden und die Belegungen werden gespeichert. Anschliessend wird versucht, die letzte *Variable* durch den nächsten Wert zu belegen, um weitere Matches zu finden. Der Algorithmus bricht ab, sobald alle Variablen-Belegungen durchprobiert worden sind. Alle gefundenen Lösungen werden dann noch daraufhin überprüft, ob sie eine NAC verletzen.
Die Methode *getSolution*(*int index*) liefert entweder ein bestimmtes Match - wenn *index* eine positive Zahl ist, dann wird das Match mit dem selben Index aus dem Lösungsvector zurückgegeben - oder ein beliebiges Match, falls der *index* eine negative Zahl ist.

Algorithmus in Pseudocode

Zunächst der Algorithmus zum Finden aller Matches - dieser wird für die LHS (*findMatches*()) und ähnlich für die NACs (*findNACMatch*()) verwendet. Kleinere Teile des Algorithmus stammen auch aus *instanciate*() bzw *nextInstance*() der Klasse *Variable*:

```
varIndex = 0;
while(true){
  var = vars.get(varIndex);
```

```
  if(!var.isInstanciated()){
    //Variable mit erstem Wert aus der dynamicDomain belegen
    var.instanciate();
  }
  else{
    //Variable mit naechstem Wert aus der dynamicDomain belegen
   var.nextInstance();
  }
  //alle Queries der Variablen mit aktueller Belegung ueberpruefen
  queryOK = var.allQueries.eval();
  if(queryOK){
    varIndex++;
   if(varIndex == vars.size()){
     //Match gefunden - speichern
     saveMatch();
     //weiter suchen
     varIndex--;
   }
  }
  else{
    if(!var.hasNextInstance()){//keine Belegung moeglich
      //Belegung und Queries aufheben
      var.deInstanciate();
      var.allQueries.reEval();
      //weiter bei vorheriger Variablen
      varIndex--;
      if(varIndex==-1){
        //Suchbaum ist durchlaufen - Schleifen-Abbruch
        break;
      }
    }
  }
}
```

In dieser Beschreibung des Algorithmus' werden Veränderungen der *dynamicDomains* nicht explizit erwähnt: Deren Inhalt wird in einem Attribut der *Queries* gespeichert und durch die Methode *eval*() auf die Werte eingeschränkt, die mit der aktuellen Belegung verträglich sind. Aber sobald eine *Variable* kein gültige Belegung mehr hat, werden in deren *Queries* durch die Methode *reEval*() die *dynamicDomains* aller abhängigen *Variablen* wieder in deren, zu Anfang gespeicherten, Vorzustand versetzt.
Für eine NAC muss lediglich ein Match gefunden werden, daher würde der Algorithmus in dem Fall schon beim ersten *saveMatch*() abbrechen. Sowohl für die LHS als auch bei den NACs wird ein *Vector* von *Variablen* abgearbeitet - der Unterschied besteht darin, dass bei einer NAC einige Elemente dieses *Vectors* auf *null* gesetzt sein können. Das muss erkannt werden und die betreffenden Elemente übersprungen werden.
Wenn alle Matches für die LHS gefunden wurden, wird überprüft ob diese ein NAC verletzten, also ob mit einem bestimmten Match für die LHS gleichzeitig auch ein Match für eine NAC gefunden werden kann. Dazu werden für eine NAC alle *Variablen*, die ebenfalls in der LHS vorkommen, wie in den gefundenen Matches belegt. Falls eine *Variable* dabei einen anderen Typ als in der LHS hat - zum Beispiel einen Subtypen - wird durch eine *TypeQuery* sichergestellt, dass das *EObject*, mit dem die *Variablen* belegt wird, auch diesen Typen hat. Anschliessend wird versucht für die *Variablen*, die nur in der NAC vorkommen ein Match zu finden - wenn dies gelingt, wird das Match für die LHS verworfen.

Alternativen zu diesem Algorithmus wären die Verwendung von Backjump anstatt Backtracking - dabei wird im Falle eines Konfliktes direkt zu der *Variablen* zurückgesprungen, die diesen verursacht hat und diese neu belegt. Unser Algorithmus tut dies nicht immer, da er im Falle eines Konfliktes nicht analysiert welche *Variable* diesen verursacht hat. Falls jedoch ein Konflikt schon bei der Belegung der verursachenden *Variablen* deutlich wird, wird dieser durch *forward checking* in den *Queries* erkannt.
Ausserdem wäre es möglich, jeweils nur einen Match für die LHS zu suchen, und diesen durch die NACs zu untersuchen - so lange bis der erste gültige Match gefunden wurde. Dass dieser Match zufällig ist, kann in einem gewissem

Maße dadurch erreicht werden, dass man die Werte in den *domains* in eine zufällige Reihenfolge bringt. Um die Suche nach einem Match fortzusetzen, muss einfach nur der Variablenvector für die LHS gespeichert werden und dann jeweils der Algorithmus zu Beginn der Schleife neu gestartet werden, wobei der Index auf die letzte $Variable$ verweist.

Kapitel 7

Beispiele

7.1 Petri-Net

Petrinetze sind eine Modellierungssprache zur Beschreibung prozessorientierter Systeme. Sie bestehen aus Stellen und Transitionen, die über gerichtete Kanten verbunden sind. Dabei können jeweils nur Stellen mit Transitionen oder Transitionen mit Stellen verbunden werden, nicht aber beispielsweise Stellen mit Stellen. Im Rahmen der Lehre wurde an der Technischen Universität Berlin ein Editor für Petrinetze in Eclipse mit GEF und EMF implementiert. Um die Möglichkeiten von EMF Transformationen zu zeigen, wollen wir die Funktionalität dieses Editors erweitern beziehungsweise ersetzen. Als Modell verwenden wir dabei das in Abbildung 2.11 dargestellte, erweitern es aber noch um eine *active*-Kante an der Transition. Diese Kante dient uns später als Markierung für eine aktivierte Transition.

Petri-Net Interpreter

Ein Merkmal von Petrinetzen ist ihre Ausführbarkeit. Bei Petrinetzen bedeutet dies das Schalten einer Transition. Schalten kann eine Transition genau dann, wenn auf allen Stellen in ihrem Vorbereich ein Token vorhanden ist. Beim Schalten werden dann alle Token von den Stellen im Vorbereich entfernt und alle Stellen im Nachbereich werden mit Token gefüllt. Diese Semantik kann man zum Beispiel durch einen EMF Interpreter simulieren lassen. Für einen

Transformationsschritt benötigt man vier Regeln:

- ActivateTransition - Setzt eine Transition auf aktiv, überprüft dabei die Bedingungen für den Vor- und Nachbereich.
- EmptyPre - Löscht alle Token im Vorbereich
- FillPost - Setzt das Token für jeweils eine Stelle des Nachbereichs
- DeactivateTransition - Setzt die Transition wieder auf inaktiv

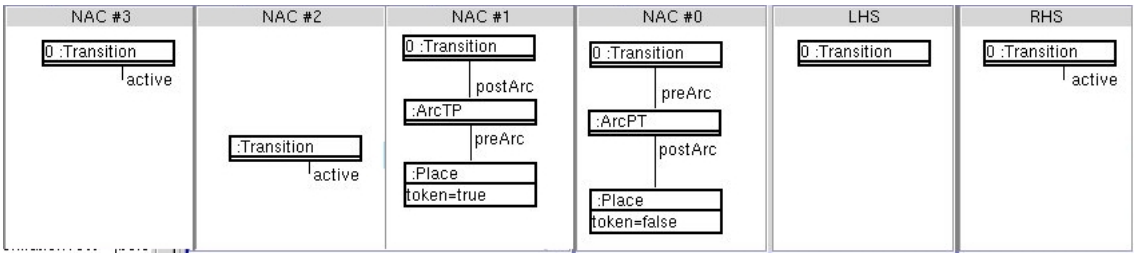

Abbildung 7.1: ActivateTransition Regel

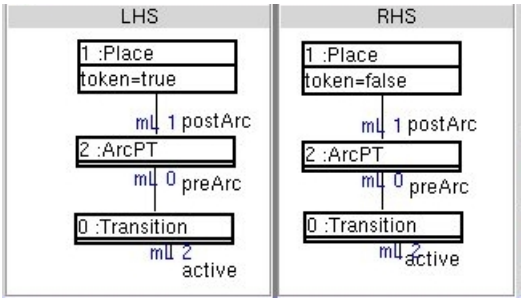

Abbildung 7.2: EmptyPre Regel

Durch diese vier Regeln ist es möglich eine Transition genau einmal schalten zu lassen. Ein Codeausschnitt, der diese vier Regeln anwendet, könnte zum Beispiel wie folgt aussehen:

Das Petrinetz auf das die Regln angewendet werden sollen, ist in abstrakter Darstellung in Abbildung 7.5 und in konkreter Darstellung in Abbildung 7.6

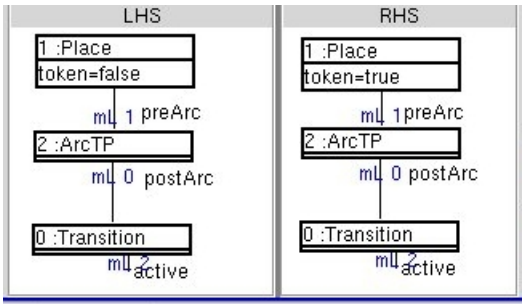

Abbildung 7.3: FillPost Regel

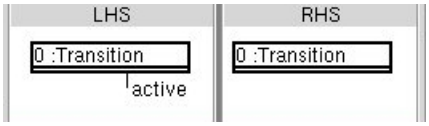

Abbildung 7.4: DeactivateTransition Regel

zu sehen.
Die vier Simulationsregeln seien in einer Datei mit dem Namen petrisimulate.tfm gesichert.

```
Interpreter interpreter = new Interpreter(petriNet1);
interpreter.loadTransformation("petrisimulate.tfm");

Vector<EObject> mapping = new Vector<EObject>();
mapping.add(t1);

applyRule("ActivateTransition", mapping, null);
while(applyRule("EmptyPre", null, null);
while(applyRule("FillPost", null, null);
applyRule("DeactivateTransition", null, null);
```

Dieser Codeblock lässt die Transition t1 Schalten, sofern die Bedingungen von *ActivateTransition* von t1 erfüllt werden. Ansonsten liefert applyRule() jeweils *false* zurück und die Anwendung des Codeblocks bleibt ohne Auswirkungen.
In diesem Fall sind die Bedingungen für die Transition t1 erfüllt, und die

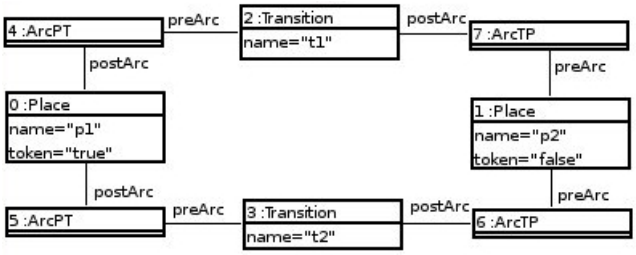

Abbildung 7.5: Abstrakte Darstellung des Petrinetzes vor Regelanwendung

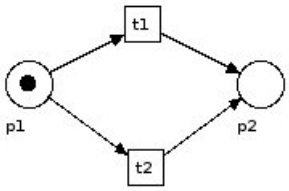

Abbildung 7.6: Konkrete Darstellung des Petrinetzes vor Regelanwendung

Transition schaltet, wodurch sich das Petrinetz aus Abbildung 7.6 zu dem Petrinetz aus Abbilung 7.7 ändert.

Falls es egal ist welche Transition schalten soll, kann man den mapping-

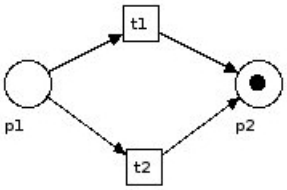

Abbildung 7.7: Konkrete Darstellung des Petrinetzes nach Regelanwendung

Parameter der Regel auf *null* setzen. Generell gilt, das lückenhaft eingegebene Informationen über den Context einer Regel automatisch vervollständigt werden, sofern dies möglich ist.

Im Falle des Beispiels hätte sowohl t1 als auch t2 die Bedingungen für das Schalten erfüllt.

Bei *EmptyPre* und *FillPost* ist es nicht mehr nötig ein Mapping zu übergeben, da nur die Transition t1 eine Active-Kante besitzt und somit als einzige

Transition für einen Match in Frage kommt.

Petri-Net Compiler

Die Hauptaufgabe des Petrinetz-Editors ist die Bereitstellung von Editoroperationen, mit denen sich ein Benutzer ein Petrinetz erstellen kann. Diese reichen vom einfachen Erstellen von Stellen und Transitionen über Move-Operationen, bis hin zum Erstellen von Kanten zwischen Stellen und Transitionen. Aufgrund der zugegebenermassen doch recht einfachen Struktur von Petrinetzen lassen sich all diese Operationen auch relativ schnell in Java realisieren. Es kann dennoch hilfreich sein Editoroperationen grafisch mit Hilfe des EMF Regeleditors zu definieren und sich dann über den Compiler daraus Code erzeugen zu lassen.
Für eine bereits erstellte tfm-Datei lässt sich der Compilierungsprozess über das Contextmenü starten, genauer über den Eintrag *Generate Rule Classes*.
Daraufhin wird ein Wizard, siehe Abb. 7.8 geöffnet, in dem man den Namen

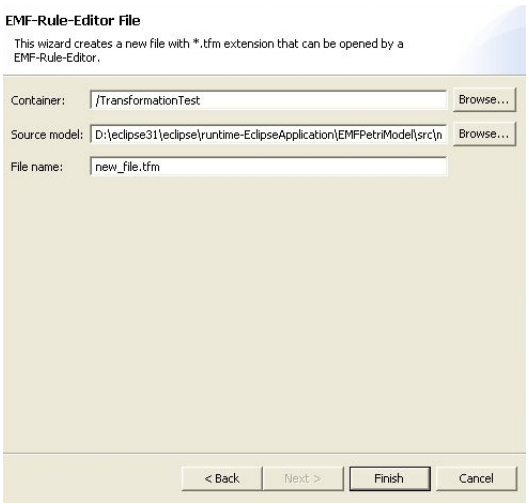

Abbildung 7.8: Wizard für Generate Rule Classes

des zu generierenden Projekts eingeben kann.
Falls man einen solchen Compilierungsprozess beispielsweise für eine Grammatik mit *CreateArcPTRule* anwendet, erhält man unter anderem

die beiden Klassen *CreateArcPTRule* und *CreateArcPTWrapper*. *CreateArcPTRule* ist die Klasse, die vom Benutzer verwendet werden soll, um Transformationen durchzuführen, während CreateArcPTWrapper nur intern verwendet wird, um die Daten der Regel dem Matchfinder zu übergeben, näheres siehe Kapitel 6.
Der generelle Aufbau einer Regelklasse gliedert sich in folgende Teile:

- Deklaration der Variablen für LHS und RHS, sowie der Factory für die Erstellung von neuen Objekten:

```
public class CreateArcPTRule extends AbstractRule {
  private PetrimodelFactory petrimodelFactory;
  private Place place0;
  private Transition transition0;
  private PetriNet petriNet0;
  private ArcPT newArcPT0;
```

- Methoden zum Setzen der Ausgangsbedingungen für die Regelanwendung:

```
public void setPlace0(Place place) {
  this.place0 = place;
}

public void setTransition0(Transition transition) {
  this.transition0 = transition;
}

public void setLHS(Vector matches){
  if(matches==null){
    setPlace0(null);
    setTransition0(null);
    setPetriNet0(null);
  }else{
    setPlace0((Place)matches.get(0));
```

```
      setTransition0((Transition)matches.get(1));
      setPetriNet0((Petrinet)matches.get(2));
    }
  }
```

- Hilfsmethoden zum Löschen alter Objekte und Erzeugen von neuen:

```
  /**
   * Creates new elements according to the rule.
   */
  private void createNewObjects() {
    newArcPT0 = petrimodelFactory.createArcPT();
    newObjects.add(newArcPT0);
  }

  /**
   * Executes changes of attribute values according to the rule.
   */
  private void setExpressions() {

  }

  private void setNewReferences() {
    place0.getPostArc().add(newArcPT0);
    transition0.getPreArc().add(newArcPT0);
    petriNet0.getArcPT().add(newArcPT);
    newArcPT0.setPetriNet(petriNet0);
  }
```

- Die execute()-Methode, um die Regel anzuwenden:

```
  public boolean execute() {
    CreateArcPTWrapper wrapper = new CreateArcPTWrapper(this.root);
    wrapper.instanciateVariables(getLHS());
```

```
    wrapper.reduceDomains();

    Vector<EObject> match = wrapper.getSolution(-1);
    if (match == null){
      return false;
    }else{
      setLHS(match);
    }

    if (canExecute() && !isExecuted) {
      initVariables();

      createNewObjects();
      deleteOldObjects();

      deleteOldReferences();
      setNewReferences();

      setExpressions();

      isExecuted = true;
      return true;
    }
    return false;
  }
```

So könnte man zum Beispiel in GEF Commands einfach eine passende Regelklasse erzeugen und in execute() anwenden, hier am Beispiel von ConnectionArcPTCommand. Das Kommando soll eine ArcPT Kante zwischen einer Stelle und einer Transition einfügen:

```
private CreateArcPTRule createArcPTRule = null;
```

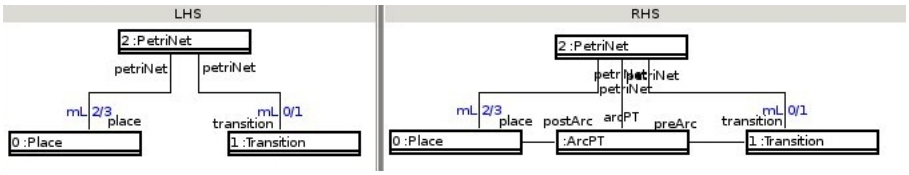

Abbildung 7.9: CreateArcPT Regel

```
/**
 * Constructor.
 */
public ConnectionArcPTCommand() {
  createArcPTRule = new CreateArcPTRule();
}

/**
 * Entry point for the command execution.
 */
public void execute() {
  createArcPTRule.setPlace0(source);
  createArcPTRule.setTransition0(target);
  createArcPTRule.setInstanceSymbol(source.getPetriNet());
  createArcPTRule.execute();
}

undo() {
  createArcPTRule.undo();
}

redo() {
  createArcPTRule.redo();
}
```

7.2 Refactoring Ecore

Unter Refactoring versteht man im Allgemeinen Veränderungen an einem Software-Projekt, mit dem Ziel dessen Struktur zu verbessern [15, 16]. Wir haben einige Refactoringregeln an das Ecore-Modell angepasst und wollen diese nun auf dessen Instanzen - auf EMF-Modelle - anwenden. Da viele dieser Regeln entweder Eingabeparameter benötigen oder ohne diese beliebig oft angewendet werden können, werden wir im Rahmen dieses Beispiels keine automatische Transformationssequenz vorstellen.
Als erstes haben wir die Regel *MoveClass* - sie verschiebt eine Klasse mit dem vorgegebenen Namen 'n' in ein Package dessen Namen ebenfalls durch einen Input-Parameter 'p' festgelegt ist.

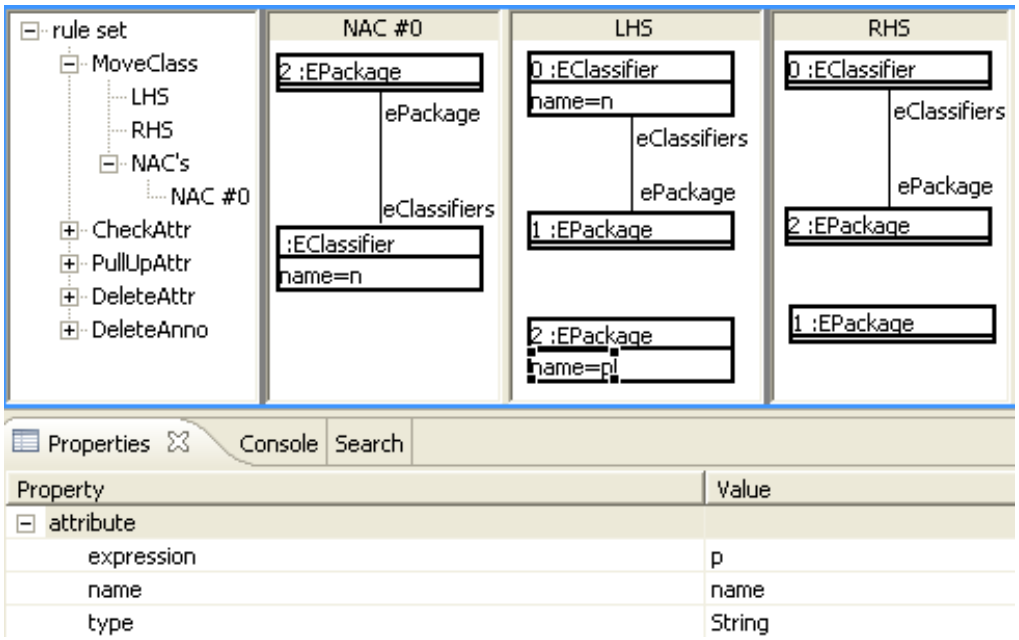

Abbildung 7.10: MoveClass Regel

Als nächstes betrachten wir eine Regelsequenz: Wenn ein Attribut in allen Kind- aber nicht in der Elternklasse vorkommt, soll es dort erzeugt, und in allen Kindklassen gelöscht werden. Dazu werden alle diese Regeln durch

Input-Parameter genauer spezifiziert: 'c' gibt den Namen der Elternklasse an, 'a' den des untersuchten Attributs und 't' dessen Typ.
Zuerst muss dazu in der Regel *CheckAttribute* überprüft werden, ob die oben beschriebene Ausgangssituation wirklich vorliegt. In der LHS werden zwei *EClasses* gematcht, wobei eine die Elternklasse der anderen ist. Die NAC0 schliesst aus, dass in der Kindklasse das gesuchte Attribut mit dem passenden Typ vorkommt. Durch die Regel wird eine *EAnnotation* mit dem Text 'no attribute' an der Kindklasse erzeugt. Sobald die Regel einmal angewendet wurde, schliessen die anderen beiden NACs aus, dass sich dies wiederholt, da in dem Falle die *EAnnotation* vorhanden ist und für diese NACs ein Match gefunden werden könnte.

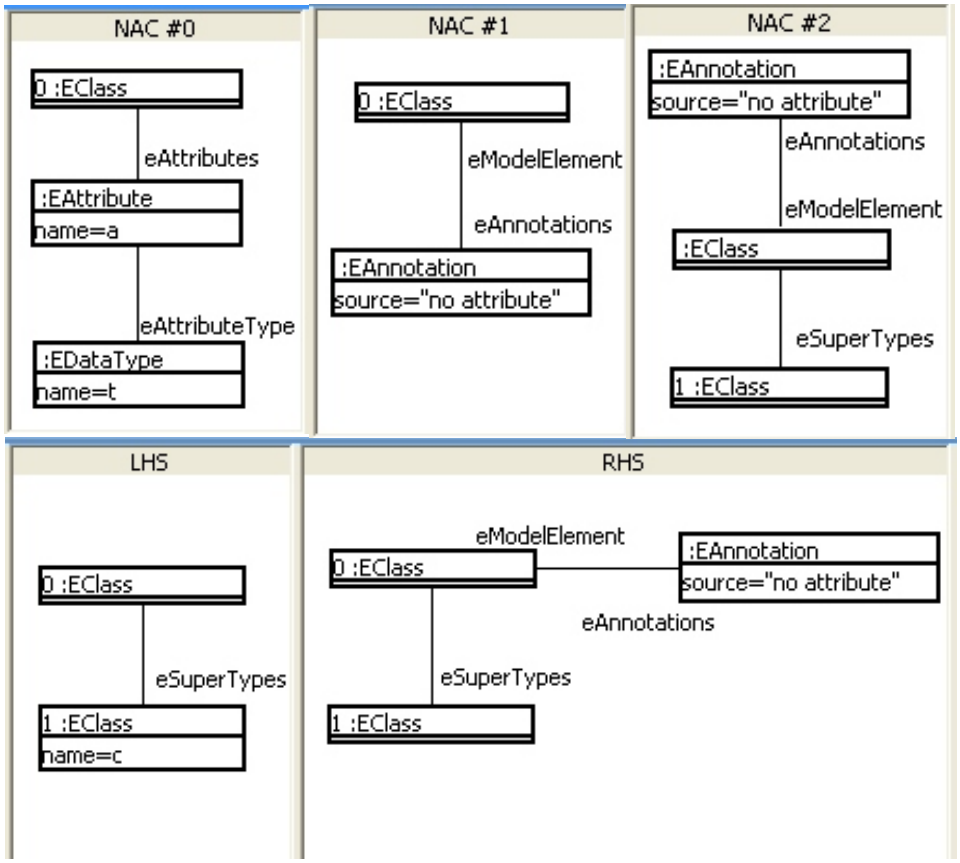

Abbildung 7.11: CheckAttribute Regel

PullUpAttribute erzeugt in der Elternklasse ein Attribut mit dem selben Namen und dem selben Typ wie in einer beliebigen Kindklasse. Falls die Regel *CheckAttribute* angewendet werden konnte und die *EAnnotation* erzeugt hat, gibt es ein Match für die erste NAC und diese Regel kann nicht angewendet werden. NAC1 schliesst aus, dass es das Attribut in der Elternklasse schon vorher gab und dass die Regel mehr als einmal angewendet werden kann.

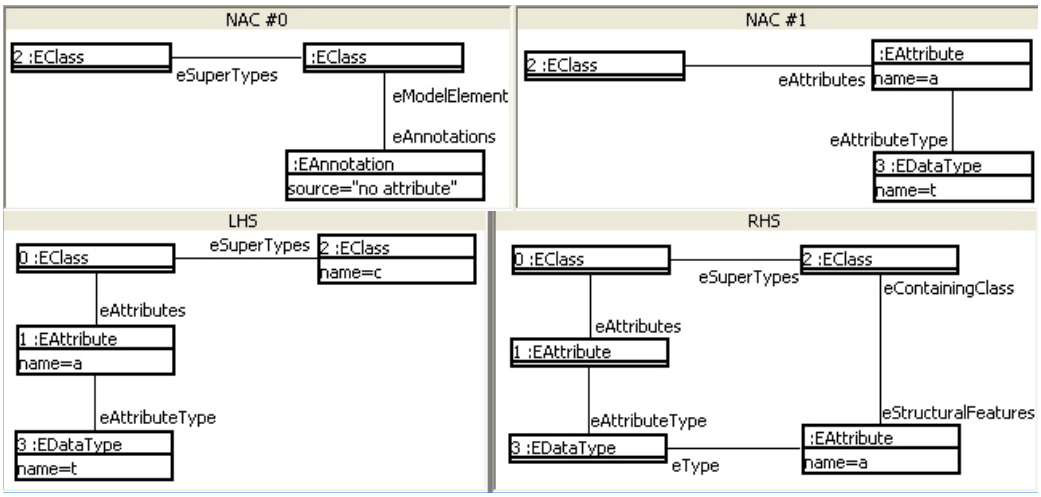

Abbildung 7.12: PullUpAttribute Regel

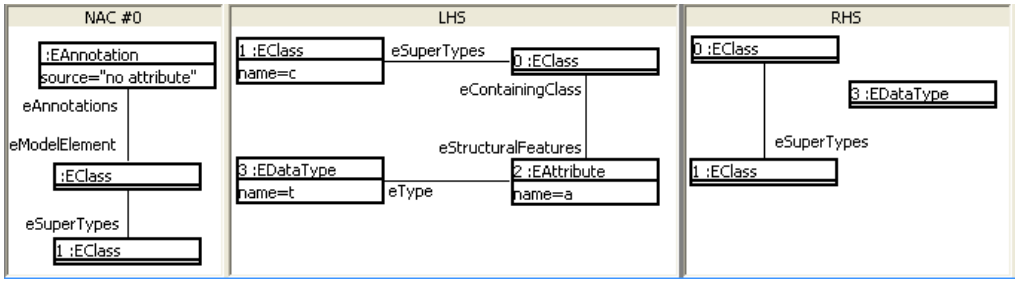

Abbildung 7.13: DeleteAttribute Regel

Eine Anwendung von *DeleteAttribute* löscht das gesuchte Attribut in einer beliebigen Kindklasse. Auch hier verhindert die NAC, dass diese Regel nach einer erfolgreichen Anwendung von 'CheckAttribute' ausgeführt werden kann.

Sonst kann sie so oft ausgeführt werden, bis in allen Kindklassen das Attribut gelöscht ist.

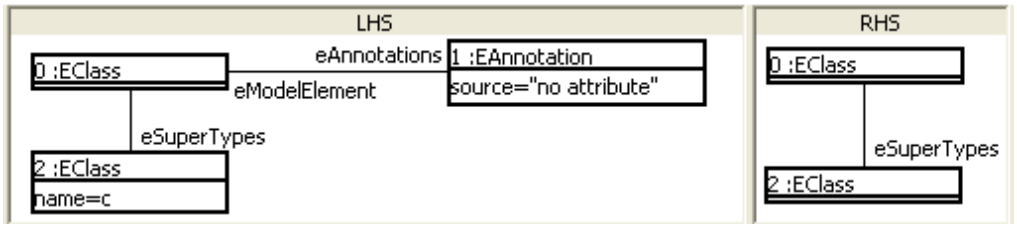

Abbildung 7.14: DeleteAnnotation Regel

Falls durch die Regel *CheckAttribute* eine *EAnnotation* erzeugt wurde, wird diese schliesslich in *DeleteAnnotation* gelöscht, sodass der Ausgangszustand wiederhergestellt ist.

Refactoring Ecore Interpreter

Da das Ecore-Modell wiederum selbst ein EMF-Modell darstellt, kann der Interpreter auf diesem ebenso wie auf allen anderen EMF-Modellen operieren. Ein Problem gibt es allerdings mit AGG, da hier die Attribute nur einige ausgewählte Typen haben können. Im Ecore-Modell kommen dagegen Attribute mit andere Typen vor, was dazu führt, dass diese bei der Transformation von EMF nach AGG nicht übersetzt werden können. Da der AGG-Graph unvollständig ist, kann kein Match gefunden und somit auch keine Regel angewendet werden.
Um dies zu beheben, wollen wir AGG so verändern, dass es auf mehr Typen unterstützt. Alternativ könnten wir bereits beim Typgraphen solche Attribute nicht erzeugen. Dies wäre zwar eine sichere Lösung dieses Problems, würde aber keine Regeln zulassen, welche die betreffenden Attribute verändern.

Refactoring Ecore Compiler

Beim Compiler ergeben sich durch die Verwendung der Ecore-Modells einige zusätzliche Schwierigkeiten - die Namen für die Package- und die Factory-Klasse entsprechen nicht dem üblichen Namensschema wie bei anderen 'normalen' EMF-Modellen. Daher haben wir speziell für dieses Beispiel ein neues Template für die Rule- und die Wrapperklassen entwickelt, das sich von dem für andere EMF-Modelle unterscheidet.
Der Code, der speziell für die Regel 'PullUpAttribute' erzeugt wird, sieht dann folgendermassen aus:

```
....

EcoreFactory fact = new EcoreFactoryImpl();

private void createNewObjects() {
  newEAttribute1 = fact.createEAttribute();
  newObjects.add(newEAttribute1);
}

private void setNewReferences() {
  EReference ref = null;

  //set reference EClass->EAttribute
  for(Iterator it=
    eclass1.eClass().getEAllReferences().iterator();it.hasNext();){

    ref = (EReference) it.next();
    if(ref.getName().equals("eStructuralFeatures")){
      if(ref.isChangeable() && !ref.isContainer()){
        if(ref.isMany()){
        EList values = (EList)eclass1.eGet(ref);
        values.add(newEAttribute1);
```

```
      }
      else{
        eclass1.eSet(ref,newEAttribute1);
      }
      }
    }
  }
  ...
  //other references are set similar
}
```

Die *EFactory* für das Ecore-Modell ist vom Typ *EcoreFactory*. Mit deren Hilfe wird in der Methode *createNewObjects*() ein *EAttribute* erzeugt. Schliesslich werden dann in *setNewReferences*() die Referenzen zwischen der Elternklasse und dem neuen *EAttribute* sowie zwischen dem *EAttribute* und dem passenden *EDataType* gesetzt. Alle andern Methoden, die in *execute*() aufgerufen werden, sind leer, da diese Regel die betreffenden Veränderungen nicht durchführt.

Ein mögliches Anwendungsbeispiel für die Refactoring-Regeln wird in Abbildung 7.15 gezeigt:
Alle Unterklassen von *Person* haben das *EAttribute* 'name' vom Typ *EString*. Daher gibt es für $CheckAttribute('Person','name','EString')$ keinen Match und es wird keine *EAnnotation* erzeugt. Durch $PullUpAttribute('Person','name','EString')$ wird auch in der Elternklasse das gleiche *EAttribute* wie in den Kindklassen erzeugt. $DeleteAttribute('Person','name','EString')$ löscht dann bei jeder Anwendung das *EAttribute* in einer der Kindklassen - da es in dem Beispiel zwei gibt, kann auch die Regel zweimal angewendet werden. Zum Schluss gibt es dann schliesslich kein Match für *DeleteAnnotation*(), da *CheckAttribute* keine *EAnnotation* erzeugen konnte.

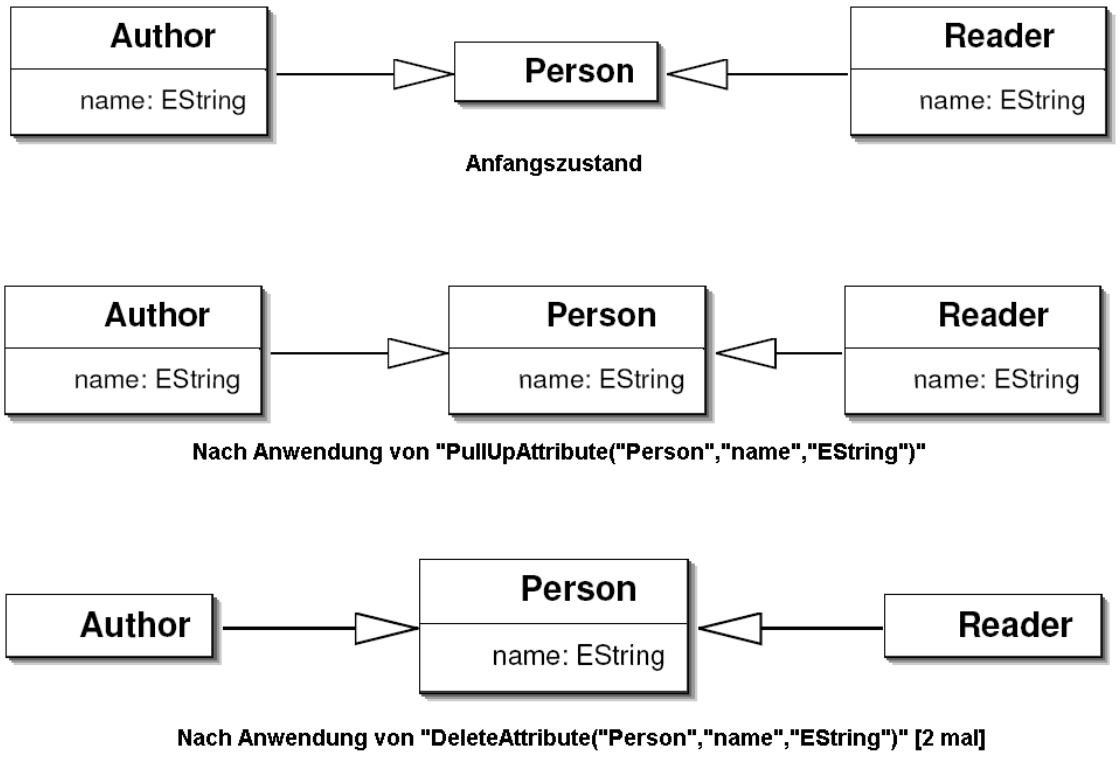

Abbildung 7.15: DeleteAnnotation Regel

Kapitel 8

Zusammenfassung

8.1 Auswertung und Rückblick

Zum Zeitpunkt des Verfassens dieses Textes liegt ein EMF Transformationsmodell, der AGG Interpreter, der EMF Compiler und ein Regeleditor vor. Im Folgenden finden sich unsere Designentscheidungen, der derzeitige Stand der Implementierung und einige geplante Verbesserungen. Während der Arbeit, hat das Transformationsmodell verschiedene Änderungen erfahren. Beispielsweise war zu Beginn geplant die Objekte in den Regeln implizit abzubilden. Aufgrund des in Kapitel 4.1.3 beschriebenen Grundes mussten wir das Modell dann doch auf explizites Mapping umstellen.
Weiterhin haben die einzelnen Klassen des Transformationsmodells einige Namensänderungen erfahren. So wurden beispielsweise Morphism durch Mapping oder Object durch Symbol ersetzt. Zu Beginn der Diplomarbeit lehnten sich die meisten Begriffe stark an den bei Graphtransformationen gebräuchlichen Wortschatz an, wurden dann aber durch allgemeinere Begriffe ersetzt. Der Grund keine EMF Begriffe wie EReference und EObject zu verwenden, liegt in den Namenskonflikten innerhalb des automatisch erzeugten Codes zu den originalen EMF Klassen.
Ein Kritikpunkt an unserem Transformationsmodell ist die sehr schwache Beziehung zwischen einer Transformationsmodellinstanz und dem dazugehörigen EMF Modell. Der einzige Bezugspunkt zwischen den

Modellklassen und den Symbolen in der Transformationsmodellinstanz ist das Attribut *Type* im Symbol. Derzeit untersuchen wir, ob es möglich ist anstelle von Symbols und Links, direkt EClass und EReference im Transformationsmodell zu verwenden. Der Vorteil wäre, dass sowohl Transformationsmodellinstanz als auch das dazugehörige EMF Modell die gleichen Klassen verwenden würden. Bisher gab es bei dem Versuch diese zu verwenden allerdings Probleme zwischen aus geladenen EMF Modellen erzeugten Klassen und solchen die direkt aus dem generierten EMF Source Code erzeugt wurden.
Unsere weitere Planung sieht vor, einen weiteren Versuch zu unternehmen, das Transformationsmodell auf EClass und EReference umzustellen und falls dies gelingt den Interpreter, Compiler und Regeleditor darauf anzupassen.

Der AGG Interpreter ist die als erstes umgesetzte Transformationsengine für EMF. Eines der Hauptprobleme bei der Implementierung war die Übersetzung der Attribute von Klassen in Attribute von Knoten. Zu Beginn war es nicht möglich Attribute in AGG mit *null* zu belegen. Somit konnte der Interpreter nur für Instanzen verwendet werden bei denen jedes Attribut jeder Klasse mit einem Wert ungleich *null* belegt wurde.
Inzwischen ist die Belegung mit *null* möglich, allerdings noch nicht in allen Einzelheiten getestet. Weiterhin noch nicht möglich ist die Verwendung von eigenen Klassen als Attribut.

Nachdem der Interpreter implementiert wurde, haben wir begonnen einen Compiler zu implementieren.. Der vom Compiler erzeugte Java Code verwendet die zum EMF Modell generierten EMF Klassen, allerdings untersuchen wir derzeit ob es nicht möglich ist auf den generierten Code zu verzichten und Modelländerungen nur über die reflexiven EMF Methoden *eGet*() und *eSet*() zu beschreiben.

8.2 Ausblick

Nach Abschluss der Entwicklung der Transformationengine soll damit begonnen werden diese in das Tiger Projekt zu integrieren. Das Tiger Projekt hat zum Ziel, automatisch Editoren für visuelle Sprachen zu generieren. Dazu gibt ein Benutzer eine Menge von grafischen Objekten und Regeln für Editieroperationen an, aus denen dann ein GEF Editor generiert wird.
Zur Zeit liegt das unterliegende Modell sowie die Regeln noch in AGG vor, sollen aber in Zukunft durch ein EMF Modell und den vom Compiler erzeugten Regelklassen ersetzt werde.

Literaturverzeichnis

[1] *AGG-System* http://tfs.cs.tu-berlin.de/agg/, 2006.

[2] *Eclipse* http://www.eclipse.org, 2006.

[3] *ATL: The Atlas Transformation Language Home Page* http://www.sciences.univ-nantes.fr/lina/atl, 2006.

[4] Frédéric Jouault and Ivan Kurtev Transforming Models with ATL http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev__transforming_models_with_atl.pdf

[5] *Eclipse Graphical Editing Framework (GEF)* http://www.eclipse.org/gef, 2006.

[6] *Eclipse Modeling Framework (EMF)* http://www.eclipse.org/emf, 2006.

[7] F. Budinsky, R. Ellersick, T. J. Grose, E. Merks, D. Steinberg. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[8] *Java Emitter Templates (JET) as part of the Eclipse Modeling Framework (EMF)* http://www.eclipse.org/emf, 2006.

[9] *VIATRA2 (VIsual Automated model TRAnsformations) framework* http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/VIATRA2/index.html, 2006.

[10] *VIATRA2 Model Transformation Framework User's Guide* *http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/viatratut.pdf?rev=HEAD*

[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[12] M. Lawley and J. Steel. Practical Declarative Model Transformation With Tefkat. In *In Proc. Model Transformation in Practice Workshop, Models Conference*, 2005.

[13] T. Levendovszky, L. Lengyel, G. Mezei, and H. Charaf. Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. In *2nd International Workshop on Graph Based Tools (GraBaTs), workshop at ICGT 2004, Rome, Italy*, 2004.

[14] T. Levendovszky, L. Lengyel, and H. Charaf. Elimination Crosscutting Constraints from Visual Model Transformation Steps `http://dawis2.icb.uni-due.de/events/AOM_MODELS2005/Lengyel.pdf`

[15] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts using Critical Pair Analysis. In *In R. Heckel and T. Mens, editors, Proc. Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra'04), Satellite Event of ICGT'04), Rome, Italy*, 2004.

[16] T. Mens, G. Taentzer, and O. Runge. Analysing Refactoring Dependencies Using Graph Transformation. *Software and System Modeling*, 2006. to appear.

[17] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. In *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, number 152 in Electronic Notes in Theoretical Computer Science, Tallinn, Estonia, 2006. Elsevier Science.

[18] *Query/View/Transformation (QVT). QVT-Merge Group, version 2.0 (2005-03-02). http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf*, 2005.

[19] Michael Rudolf. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In *6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), LNCS 1764*, pages 238–251. Springer Verlag, 2000.

[20] Roman Barták. Constraint Satisfaction - Systematic Search Algorithms `http://ktiml.mff.cuni.cz/~bartak/constraints/backtrack.html`