INSTITUT FÜR SOFTWARETECHNIK UND THEORETISCHE INFORMATIK
TECHNISCHE UNIVERSITÄT BERLIN

Bachelorarbeit

Entwicklung eines visuellen Editors zur Steuerung von EMF-Modelltransformationen

Johann Schmidt

08.09.2010

Betreuung: Prof. Dr. H. Ehrig und Dr. Claudia Ermel

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Thesis selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Quellen sind vollständig zitiert.

Berlin den,08.09.2010 Johann Schmidt _____

Inhaltsverzeichnis

Index 5

Abbildungsverzeichnis 6

1. Einleitung 8

2. Grundlagen 9

2.1. Modellierung 9

2.1.1. Modell 9

2.1.2. Metamodell 10

2.1.3. Code Generierung in Eclipse Modelling Framework 10

2.2. Graph und Graphtransformationen 11

2.2.1. Gerichteter Graph 11

2.2.2. Typisierter Graph 12

2.2.3. Graphmorphismus 13

2.2.4. Transformationsregeln 14

2.2.5. Graphtransformationen 14

2.2.6. Anwendungsbedingungen 16

2.2.7. Attributierte, getypte Graphtransformation mit Vererbung 17

2.3. Henshin 18

2.3.2. Basis-Henshinmodell 19

2.3.3. Basis-Henshineditor 20

2.3.4. EMF-Kompatibilität 21

2.3.5. Editor-Ansicht des Basis-Henshin-Editors 24

3. Erweiterung des Henshin-Editors um automatisch gesteuerte Transformationen 26

3.1. Modellerweiterung 26

3.2. Transformation Units 28

3.2.1. Counted Unit 28

3.2.2. Sequential Unit 28

3.2.3. Conditional Unit 28

3.2.4. Independent Unit 29

3.2.5. Priority Unit 29

3.3. Realisierung eines Editors für Transformation Units 29

3.4. Verwendungsbeispiel „Personal Mobility Manager“ 32

4. Interne Architektur der Erweiterungen des Henshin-Editors 43

4.1. Umsetzung des Model-View-Controller Prinzip 43

4.2. Schnittstelle zur Transformation Engine von Henshin 45

5. Verwandte Arbeiten 46

6. Zusammenfassung und Ausblick 47

6.1. Durchgeführte Erweiterungen und erbrachte Vorteile. 47

6.2. Ausblick auf zukünftige Arbeiten 48

Literaturverzeichnis 49

Benutzerhandbuch 52

Index

A

Anwendungsbedingung · 7, 16, 19, 24, 35, 36, 38, 39
AttributeCondition · 26

E

Edge · 19
EMF - Eclipse Modeling Framework · 11, 21, 22, 24

G

GEF - Graphical Editing Framework · 20
Graph · 6, 7, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 24, 27, 43, 44, 45, 46, 47, 49, 50
Graphmorphismus · 6, 13, 14, 15, 16, 17, 18
Graphtransformation · 8, 16, 17

H

Henshin · 6, 8, 18, 19, 20, 21, 22, 24, 25, 26, 43, 45, 47, 49

K

Klassendiagramm · 9, 11
Klasseninstanzen · 9, 10

M

Mapping · 19, 24, 27, 29, 30, 31
Metamodel · 10
Modell · 9, 10, 11, 20, 21, 23, 24, 26, 32
MUVITOR - Multi View Editor · 20

N

NestedCondition · 19
Node · 19

P

Parameter · 19, 24, 26, 28, 29, 30, 31, 40

S

source · 19

T

target · 19
Transformation Unit · 26, 28, 29, 30, 31
Transformationsregel · 6, 14, 26, 40, 48
Typgraph · 6, 12, 13, 17, 18

X

XMI - XML Metadata Interchange · 11

Abbildungsverzeichnis

Abbildung 1. Pizza Service Modell....9
Abbildung 2. Pizza Service Metamodell....10
Abbildung 3. Graph visuelle Darstellung....12
Abbildung 4. Typisierter Graph....13
Abbildung 5. Graphmorphismus....13
Abbildung 6. Transformationsregel mit Schnitt K und Graphmorphismen l und r....14
Abbildung 7. Graphmorphismus von LHS nach Graph G....15
Abbildung 8. Graph D....15
Abbildung 9. Graph H....16
Abbildung 10. Attributierter Typgraph mit Vererbung....18
Abbildung 11. Attributierter Graphmorphismus....18
Abbildung 12. Henshinmodell aus „Visuelle Sprache“ Projekt....20
Abbildung 13. Mehrere Container sind verboten....21
Abbildung 14. Zyklisches Containment ist verboten....22
Abbildung 15. Erstellung ein neuer Knoten /neuen Kanten....22
Abbildung 16. Löschung einen Knoten/ eine Kante....23
Abbildung 17. Knotenzugehörigkeit Wechsel....23
Abbildung 18. Ansicht des Henshin Editors....25
Abbildung 19. Transformation Units....27
Abbildung 20.Transformation Units: Baum-Ansicht....30
Abbildung 21. Transformation Unit Ansicht....31
Abbildung 22. Transformation Unit mit untergeordnetem Unit....31
Abbildung 23. Counted Unit....32
Abbildung 24. Conditional Unit....32
Abbildung 25. "Personal Mobility Manager" Metamodell....33
Abbildung 26. "Personal Mobility Manager" Startgraph....33
Abbildung 27. Rule "ChooseDestination"....34
Abbildung 28. Rule "RequestRouteMap"....34
Abbildung 29. Rule "ResponseRouteMap"....34
Abbildung 30. Rule "RequestJamStatus"....35
Abbildung 31. Rule "ResponseJamStatus"....35
Abbildung 32. Rule "NotAllRoutesJammed"....36

Abbildung 33. Erste Anwendungsbedingung von "NotAllRoutesJammed" 36
Abbildung 34. Zweite Anwendungsbedingung von "NotAllRoutesJammed" 36
Abbildung 35. Rule "ForbidCar" 37
Abbildung 36. Rule "SelectBike" 37
Abbildung 37. Rule "ChooseRoute" 37
Abbildung 38. Erste Anwendungsbedingung von "ChooseRoute".....38
Abbildung 39. Zweite Anwendungbedingung von "ChooseRoute" 38
Abbildung 40. Rule "DeleteJamStatus".....38
Abbildung 41. Rule "DeleteUnusedRoute" 39
Abbildung 42. Erste Anwendungsbedingung von "DeleteUnusedRoute" 39
Abbildung 43. Zweite Anwendungsbedingung von "DeleteUnusedRoute".....39
Abbildung 44. Sequential Unit "mainUnit" 40
Abbildung 45. Counted Unit "pollTrafficWS" 40
Abbildung 46. Sequential Unit "trafficWS" 41
Abbildung 47. Conditional Unit "decideMeans" 41
Abbildung 48. Sequential Unit "switchToBike".....42
Abbildung 49. Counted Unit "removeAllUnusedData" 42
Abbildung 50. Independent Unit "removeUnusedData" 42
Abbildung 51. Model-View-Controller Prinzip.....43
Abbildung 52. Graph View mit Historie von angewendeten Transformationsregeln.....46

1. Einleitung

Die Modellierung bei der Softwareentwicklung spielt heutzutage eine immer größere Rolle um die Komplexität der Aufgabe zu beherrschen. Außerdem kann anhand eines Modells bereits vor der Implementierung geprüft werden ob die Aufgabe richtig verstanden wurde und die vorgesehene Lösung der Aufgabestellung entspricht. Dabei kann es dann nötig werden die Modelle anzupassen.

Diese Arbeit handelt nicht nur von der Modellierung selbst, sondern von der Prüfung des Modells und dessen Anpassung. Auf der Basis der Modell-Transformation soll es ermöglicht werden eine Prüfung eines Modells durchzuführen um dessen Eigenschaften und Beziehungen zu ermitteln. Auch automatische Modelländerungen oder Modell-Übersetzungen können durchgeführt werden. Das ist besonders wichtig bei komplexeren Projekten wo nicht alle Eigenschaften des Systems zu überblicken sind. Bei solchen Projekten können oft durch manuelle Änderungen die Fehler verursacht werden.

Die Arbeit basiert auf dem Tool Henshin zur Transformation von EMF-Modellen. Bisher gestattete der Henshin-Editor lediglich die Definition von einzelnen visuellen Transformationsregeln. Die Anwendung der Regeln musste über zusätzlichen Java-Code gesteuert werden.

Ziel der Bachelorarbeit ist daher, die Weiterentwicklung des Editors von Henshin zu einem visuellen und komplexen Modellierungstool, das automatisch gesteuerte und regelbasierte Modelltransformationen von EMF- Modellen unterstützt.

Die Bachelorarbeit ist folgendermaßen aufgebaut: Nach einer Einführung der Grundlagen für die Modellierung mit EMF und Graphtransformation in Kapitel 2.1 und 2.2, folgt in Kapitel 2.3 die Beschreibung des Basis-Henshin-Editors, auf dem diese Arbeit aufbaut. Kapitel 3 beschreibt schrittweise die Modellerweiterungen um verschiedene Kontrollstrukturen (genannt Transformation Units), deren Semantik und Visualisierung im Tool. Anhand der Fallstudie Personal-Mobility-Manager, wird die Verwendung des visuellen Editors für Transformation Units illustriert. Die Architektur des implementierten Editors, der auf den Eclipse-Frameworks GEF und MUVITOR basiert, wird in Kapitel 4 beschrieben. Es werden auch die verwandten Arbeiten in Kapitel 5 vorgestellt. Kapitel 6 fasst die Ergebnisse der Arbeit zusammen und schließt die Arbeit ab mit einem Ausblick auf weiterführende Arbeit.

2. Grundlagen

2.1. Modellierung

2.1.1. Modell

Ein Modell ist eine abstrakte Darstellungsform von einem System. Sie ermöglicht ein System besser zu verstehen, indem sie nicht relevante Teile des Systems abstrahiert. Es existieren viele verschiedene Typen von Modellen, die die unterschiedlichsten Aspekte darstellen können. Ein Modell besteht aus Modellelementen (Objekte) und ihren Relationen (Links). Modellelement-Typen werden in dieser Arbeit als Klassen dargestellt. Die Klassen können verschiedene Attribute haben und auf andere Klassen verlinkt werden. Ein Klassendiagramm bezeichnet man auch als Metamodell. Modell selbst stellen wir als Objektdiagramme dar. Als Beispiel nehmen wir ein Modell eines Pizza Service (Abbildung 1). Da ist ein Objektdiagramm zu sehen, das zwei Customer-Objekte enthält (zwei Kunden). Außerdem haben die Kunden Bestellungen (Order) aufgegeben. Hier ist gut zu sehen, dass ein Modell mehrere Objekte aus der gleichen Klasse enthalten kann. Objekte einer Klasse werden auch Klasseninstanzen dieser Klasse genannt.

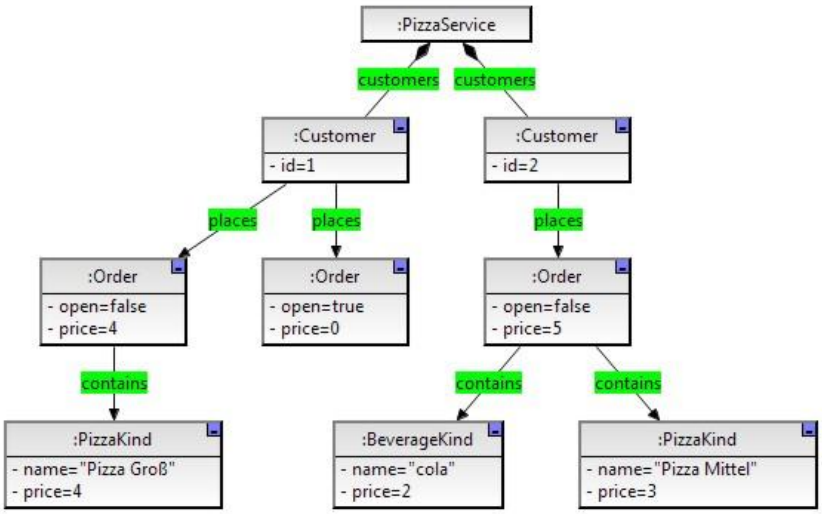

Abbildung 1. Pizza Service Modell

2.1.2. Metamodell

Ein Metamodell beschreibt Modellelement-Typen und ihre Relationen, d.h. es beschreibt wie ein Modell aussehen kann, Instanzen welcher Klassen im Modell sein können und welche Beziehungen zwischen welchen Klassen existieren dürfen. Das Metamodell für Pizza Service ist in Abbildung 2 dargestellt. Es gibt eine Klasse „PizzaService“, dessen Instanzen Objekte der Klassen „Customer“, „Order“ und „OrderItem“ enthalten können. Außerdem kann eine Instanz der Klasse „Customer“ mehrere Links „places“ zu Klasseninstanzen der Klasse „Order“ und eine Instanz der Klasse „Order“ kann mehrere Links „orders“ zu Klasseninstanzen der Klasse „OrderItem“ haben. Von der Klasse „OrderItem“ erben noch drei Klassen („BeverageKind“, „PizzaKind“, „ToppingKind“) und deshalb können sie als „OrderItem“ von Instanzen der Klasse „Order“ verlinkt werden. Die Attribute, die die Klasse „OrderItem“ hat, werden auch in den geerbten Klassen vorhanden sein.

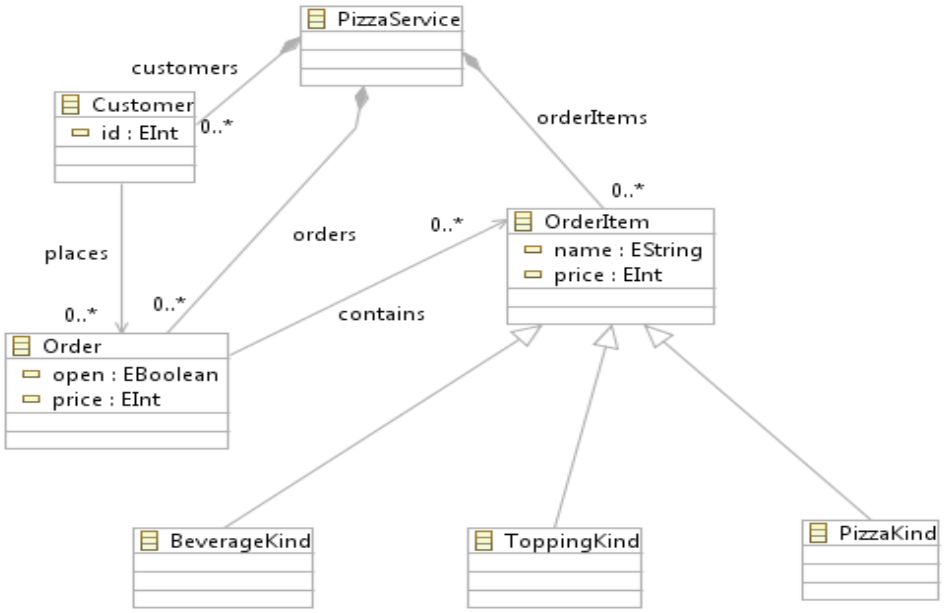

Abbildung 2. Pizza Service Metamodell

2.1.3. Code Generierung in Eclipse Modelling Framework

Wenn man ein Metamodell hat, kann man schon viel über das modellierte System sagen, zum Beispiel welche Klassen in System sind, welche Attributen sie haben, welche Interfaces sie implementieren und von welchen Klassen sie erben. Die Implementierung eines Metamodells ist eine Routinearbeit, die automatisiert werden kann. Genau das übernimmt die Code Generierung des Eclipse Modeling Frameworks

(EMF) [1]. Der Programmierer bekommt auf diese Weise einen sauberen Java-Code mit allen aus Modell bekannten Klassen. In dem Modell vordefinierte Attribute und entsprechende Standardmethoden werden in den Klassen ebenfalls vorhanden sein.

Ein EMF-Modell[1] repräsentiert ein Klassendiagramm, wie es schon aus anderen Modellierungstools, wie z.B. UML, bekannt ist. Außerdem benutzt EMF für seine Modelldefinitionen das XMI-Format (XML Metadata Interchange). Deshalb ist es möglich ein Modell direkt in XMI zu definieren. So ist es gut sichtbar, dass EMF verschiedene Repräsentationen desselben Datenmodells besitzt (Java-Code, Klassendiagramm, XML). Was man nicht gleich sehen kann ist, dass EMF bei Änderungen der Modell Repräsentation die restliche zwei Repräsentationen automatisch anpasst oder aus einer dieser drei Formen die anderen zwei generieren lässt. Es existieren mehrere Möglichkeiten ein Modell in EMF zu erstellen:

- Mit Hilfe eines XML- oder Text-Editors ein XMI-Dokument erstellen.
- Das XMI-Dokument aus einem anderen Modellierungswerkzeug exportieren.
- Das Modell aus annotierten Java-Interfaces erstellen.
- Mit Hilfe des EMF Ecore Baum-Editors
- Mit Hilfe des visuellen Klassendiagramm-Editors von GMF, der auf dem Ecore- Metamodell basiert.

2.2. Graph und Graphtransformationen

2.2.1. Gerichteter Graph

Ein gerichteter Graph besteht aus einer Knotenmenge, einer Kantenmenge und einem Paar von Abbildungen, die für jede Kante einen Quell- und einen Ziel-Knoten zuweisen. Sei V eine Knotenmenge, E eine Kantenmenge, src und tar ein Paar von Abbildungen, dann ist ein Graph G folgendermaßen definiert:

$$G=(V^G,E^G,src^G,tar^G)$$

$$src^G,tar^G: E^G \rightarrow V^G$$

[1] In dieser Arbeit bezeichne ich EMF-Modelle als Metamodelle und EMF-Instanzen als Modelle.

Eine visuelle Darstellungsform ist in Abbildung 3 zu sehen. Für bessere Identifizierung haben alle Elemente einen Namen. So sind die Knoten als Kreise mit den Namen v1,..,vn und die Kanten als Pfeile mit den Namen e1,..,em dargestellt. Abbildungen src und tar bestimmen Quell- und Ziel-Knoten für jeden Pfeil, z.B.:

src: e1 → v4, tar: e1 → v1

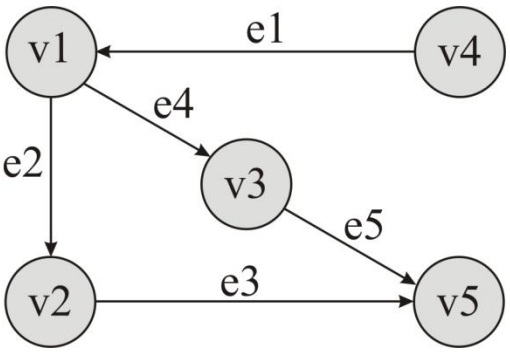

Abbildung 3. Graph visuelle Darstellung

2.2.2. Typisierter Graph

Um einen typisierten Graph zu definieren, wird einen Typgraph gebraucht. Ein Typgraph ist ein gerichteter Graph in dem jedes Element einen eindeutigen Typen repräsentiert. So wird für jeden Kantentyp bestimmt, welche Knotentypen für den Quell- und Ziel-Knoten erlaubt sind, oder andersrum, für jeden Knotentyp wird bestimmt welche Kantentypen von ausgehenden bzw. eingehenden Kanten er haben darf.

Weiterhin besteht ein typisierter Graph aus einem Graph G, einem Typgraph T und einem Paar von Abbildungen t^V, t^E, die für jeden Knoten aus dem Graph G einen Knoten aus Typgraph T bzw. für jede Kante aus dem Graph G eine Kante aus Typgraph T zuweisen. Außerdem soll für jede Kante e gelten, dass der Typ ihres Quell- Knotens src(e) für den Typ der Kante als Quell-Knoten-Typ bzw. Typ des Ziel-Knotens tar(e) als Ziel-Knoten-Typ erlaubt ist: $t^V(\mathrm{src}(e)) = \mathrm{src}(t^E(e))$ und $t^V(\mathrm{tar}(e)) = \mathrm{tar}(t^E(e))$

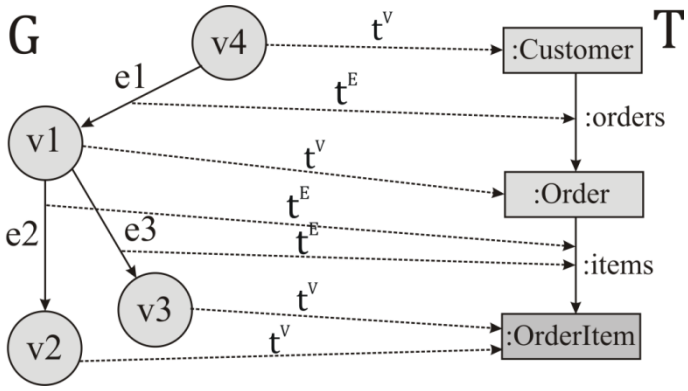

Abbildung 4. Typisierter Graph

2.2.3. Graphmorphismus

Ein Graphmorphismus $h(h_V,h_E):G_1 \rightarrow G_2$ ist ein Paar von Abbildungen, die jeden Knoten von Graph G_1 auf einen Knoten von Graph G_2 bzw. jede Kante von Graph G_1 auf eine Kante von Graph G_2 abbilden und dabei die Typen und die Quell- und Zielknoten bewahren. Das heißt:

$h_V(src^{G1}(e))=src^{G2}(h_E(e))$

$h_V(tar^{G1}(e))=tar^{G2}(h_E(e))$

$t_V(v)=t_V(h_V(v))$ bzw. $t_E(e)=t_E(h_E(e))$

Da die Typen von Knoten und Kanten bewahrt werden sollen, müssen die beiden Graphen einen und denselben Typgraph haben. Ein Beispiel von Graphmorphismus ist in Abbildung 5 zu sehen. Dabei werden entsprechende Typen von Typgraph direkt in Knoten gezeigt.

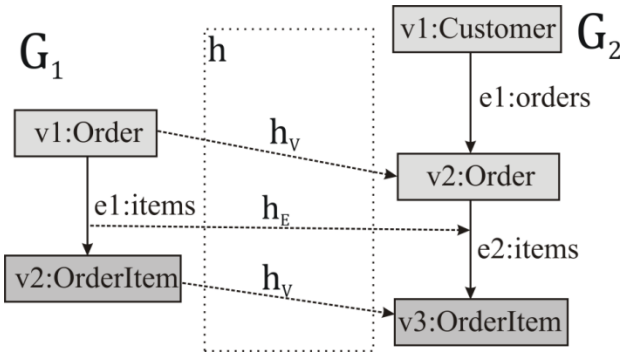

Abbildung 5. Graphmorphismus

2.2.4. Transformationsregeln

Um Änderungen, von Graphen, zu definieren, ist noch eine Definition des Schnitts von Graphen notwendig. Ein Schnitt von zwei Graphen G_1 und G_2 ist ein Graph $K=(V^K,E^K,src^K,tar^K)$ zusammen mit zwei Graphmorphismen $K \rightarrow G_1$ und $K \rightarrow G_2$.

Eine Graph-Transformationsregel besteht aus drei Graphen LHS, RHS und deren Schnitt K (Abbildung 6). Sei $l:K \rightarrow LHS$ der injektive Graphmorphismus von K nach $LHS=(V^{LHS},E^{LHS},src^{LHS},tar^{LHS})$ und $r:K \rightarrow RHS$ der injektive Graphmorphismus von K nach $RHS=(V^{RHS},E^{RHS},src^{RHS},tar^{RHS})$, dann ist $LHS \xleftarrow{l} K \xrightarrow{r} RHS$ eine Graphtransformationsregel. Der Schnitt K beschreibt die Elemente, die durch die Regel bewahrt werden.

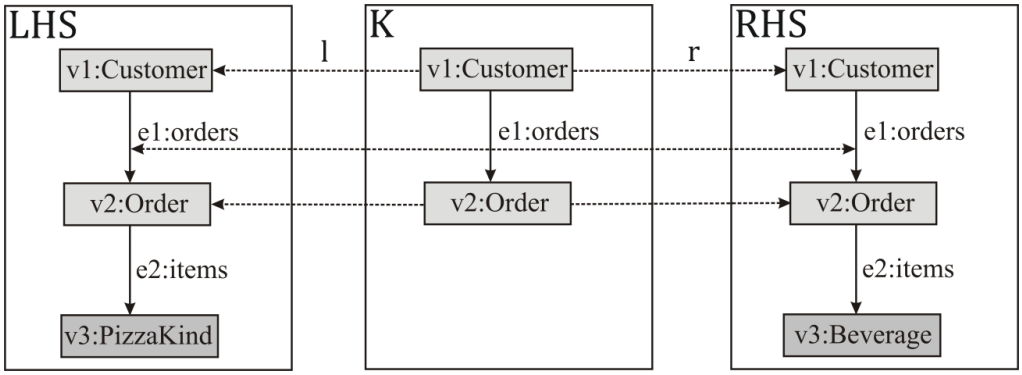

Abbildung 6. Transformationsregel mit Schnitt K und Graphmorphismen l und r

2.2.5. Graphtransformationen

Jetzt bleibt noch die Frage wann eine Transformationsregel ausgeführt werden kann und wie die Änderungen an einem Graphen durchgeführt werden. Sei $G=(V^G,E^G,src^G,tar^G)$ ein Graph auf den eine Regel angewendet werden soll. Eine Regel kann auf einen Graphen angewendet werden, wenn mindestens ein injektiver Graphmorphismus m (genannt Match) von LHS nach Graph G existiert (Abbildung 7). Die Regelanwendung wird dann in zwei Schritten durchgeführt:

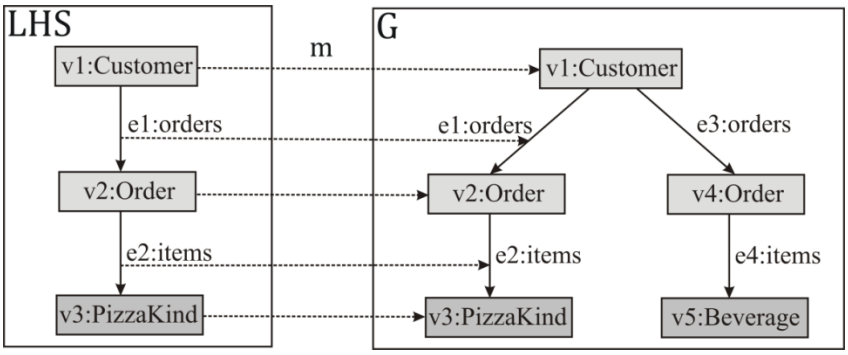

Abbildung 7. Graphmorphismus von LHS nach Graph G

1. Alle Knoten und Kanten in G die über m abgebildet werden, aber zu welchen es kein Urbild in K gibt, werden gelöscht. Außerdem können optional auch alle sogenannten hängenden Kanten aus G gelöscht werden. Das sind Kanten, die nicht in Match m liegen, aber die einen Quell- oder Zielknoten haben, der gelöscht wird. Dadurch entsteht aus Graph G der Graph D:

   $D = (V^D, E^D)$ mit

   $V^D = V^G \backslash m(V^{LHS}) \cup m(l(V^K))$

   $E^D = (E^G \backslash m(E^{LHS})) \backslash DE \cup m(l(E^K))$, wobei

   $DE = \{ e \in E^G \backslash m(E^{LHS}) \mid$

   $(\exists v \in V^{LHS}: m(v) = src(e) \wedge \neg \exists x \in K: l(x) = v) \vee$

   $(\exists w \in V^{LHS}: m(w) = tar(e) \wedge \neg \exists x \in K: l(x) = w)\}$

   die Menge der hängenden Kanten (Dangling Edges) in G bezeichnet.

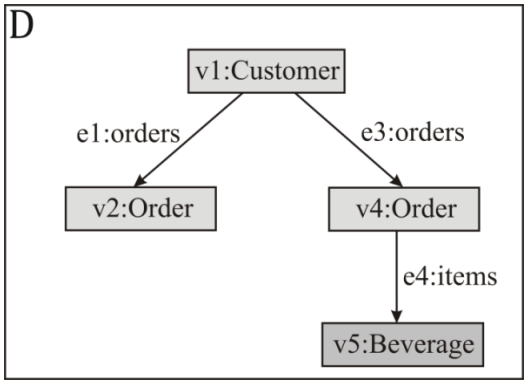

Abbildung 8. Graph D

2. Im zweiten Schritt werden die neuen Knoten und Kanten zu D hinzugefügt die in RHS vorhanden sind, aber zu welchen es kein Urbild in K gibt. Dadurch entsteht aus Graph D das Resultat der Graphtransformation - der Graph H:

   $H = (V^H, E^H)$ mit

   $V^H = V^D \uplus (V^{RHS} \setminus r(V^K))$

   $E^H = E^D \uplus (E^{RHS} \setminus r(E^K))$, wobei „⊎“ die disjunkte Vereinigung bezeichnet die sicherstellt, dass die Elemente aus $RHS \setminus r(K)$ als neue Elemente hinzugefügt werden.

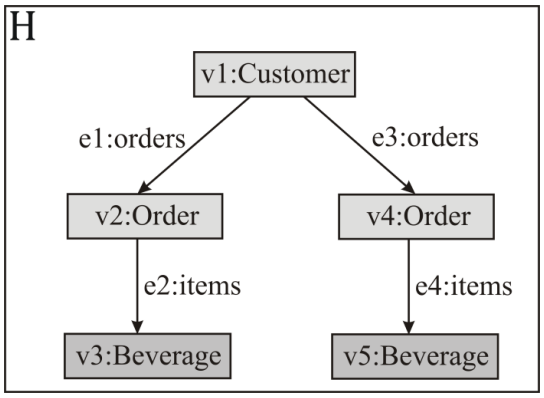

Abbildung 9. Graph H

2.2.6. Anwendungsbedingungen

Zusätzlich zur Existenz eines Matches LHS→G kann eine Anwendungsbedingung definiert werden, die bestimmt, ob die Regel angewendet werden darf. Eine Anwendungsbedingung ist ein Graph AB zusammen mit einem injektiven Graphmorphismus ab:LHS→AB. Eine Anwendungsbedingung $LHS \xrightarrow{ab} AB$ ist erfüllt (und liefert true), wenn es einen injektiven Graphmorphismus x:AB→G in den Graphen G gibt, der mit dem Match m:LHS→G verträglich ist, d.h. es gilt

$x \circ ab = m$

$$\begin{tikzcd} LHS \arrow[rr, "ab"] \arrow[dr, "m"'] & \arrow[d, phantom, "="] & AB \arrow[dl, "x"] \\ & G & \end{tikzcd}$$

Um mehr Freiheit bei den Anwendungsbedingungen zu haben, werden logische Ausdrücke AND, OR und NOT für die Verknüpfung der Bedingungen benutzt. Eine negative Anwendungsbedingung (NAC) $NOT(LHS \xrightarrow{ab} AB)$ ist erfüllt, wenn es keinen

injektiven Graphmorphismus $x:AB \rightarrow G$ gibt, so dass $x \circ ab = m$. Die Regel darf dann angewendet werden, wenn der logische Ausdruck von den formulierten Anwendungsbedingungen als gesamtes Ergebnis „true“ zurück liefert.

2.2.7. Attributierte, getypte Graphtransformation mit Vererbung

Wenn Modelle mit Graphen verglichen werden, ist es leicht zu sehen, dass ein Modell eine erweiterte Variante von Graph und ein Metamodell von Typgraph ist. Es fehlen noch die Attribute, Multiplizitäten und Vererbungskanten, die aber auch für Graphen hinzugefügt werden können. Ein Graph mit Attributen AG enthält dann zusätzlich Attributknoten V_A^{AG} und Attributierungskanten E_A^{AG} von den Graphknoten V^{AG} zu den Attributknoten V_A^{AG} [2]. Attributknoten entsprechen Datenelementen einer Algebra oder Variablen. Ein attributierter Graphmorphismus $h_A:AG_1 \rightarrow AG_2$ zwischen zwei attribuierten Graphen bildet dann auch die Attributierungskanten und Attributknoten von AG_1 auf Attributierungskanten und Attributknoten von AG_2 ab, und zwar, verträglich mit Quell- und Zielknoten von Kanten. Datenelementen (wie die Zahl 7 oder der String „Cola“) dürfen nur identisch abgebildet werden. Variablen dürfen auf andere Variablen oder auf Datenelementen vom selben Datentyp abgebildet werden.

Ein attributierter Typgraph ATG enthält zusätzlich zu einem normalen Typgraphen Attributknotentypen V_A^{ATG} und Attributierungskantentypen E_A^{ATG}, die die normalen Graphknotentypen mit ihren Attributknotentypen verbinden.

Von einem getypten attributierten Graphen AG gibt es dann einen Typmorphismus $t=(t_V,t_E,t_A,t_{AE})$, der über $t_A: V_A^{AG} \rightarrow V_A^{ATG}$ und $t_{AE}: E_A^{AG} \rightarrow E_A^{ATG}$ die Typen der Attributknoten bestimmt.

Ein Typgraph ATG kann zusätzlich Vererbungskanten $\uparrow$ enthalten, die bedeuten, dass die Attribute und Kanten des Supertyps (Ziel der Vererbungskante) auch im Subtyp (Quelle der Vererbungskante) vorliegen.

Dies bedeutet für einen attributierten Graphmorphismus $h_A:AG_1 \rightarrow AG_2$, dass ein Knoten v in AG_1 entweder auf ein Knoten in AG_2 vom gleichen Typ abgebildet ($t_V(v)$) wird oder aber auf einen Knoten eines Subtyps von v. Ein Beispiel für einen attributierten Graphen mit Vererbung und für einen attributierten Graphmorphismus ist in Abbildungen 10,11 dargestellt.

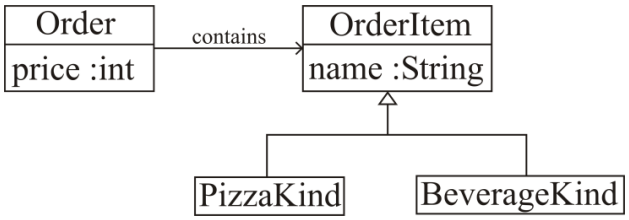

Abbildung 10. Attributierter Typgraph mit Vererbung

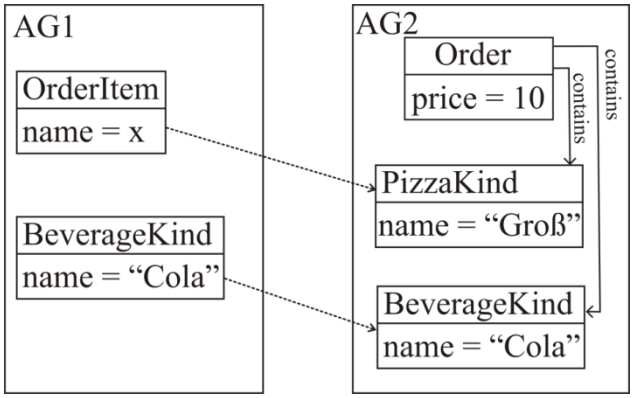

Abbildung 11. Attributierter Graphmorphismus

Ein attributierter Typgraph mit Vererbung und Multplizitäten, hat zusätzlich eine Abbildung, die den Kantentypen eine minimale und maximale Anzahl von Knoten zuordnet die es im Graphen ausgehend von einem Quellknoten geben darf. Das entspricht dann einem Metamodell.

Zusätzlich führen wir noch einen extra Kantentyp „Containment-Kante" ein. Mit Graphmorphismen werden Containment-Kanten auch wieder nur auf Containment-Kanten zwischen passenden Knotentypen abgebildet.

2.3. Henshin

Nach dem alle nötigen Erweiterungen des Graphs gemacht wurden, ist es nun möglich mit Henshin Modeltransformationen basierend auf Graphtransformationen zu realisieren. In der Praxis werden alle Transformationen in Henshin mit Prinzip „In-Place" durchgeführt, was bedeutet, dass der resultierende Graph H nach einer Transformation den ursprünglichen Graphen G ersetzt.

Außerdem darf Henshin mehrere Metamodelle in einem Transformationssystem haben und Graphen getypt über einer Vereinigung der Metamodelle transformieren. Mit

diesem Prinzip kann in Henshin sogenannte „Model to Model“ Transformationen durgeführt werden.

2.3.2. Basis-Henshinmodell

In diesem Kapitel wird der Kern des Modells betrachtet (Abbildung 12). Wie er vor Beginn der Arbeit vorlag. Wichtigstes Element für die Transformation ist, wie in der Theorie, ein Graph. Ein Graph besteht aus Knoten (Node) und Kanten (Edge) die einen Knoten als Quell- (source) und einen als Ziel- (target) Knoten haben. Außerdem kann ein Knoten beliebig viele Attribute haben. Ein Graph kann als ein selbständiges Element in dem Transformationssystem vorkommen oder als Bestandteil einer Regel (Rule) bzw. einer Anwendungsbedingung (NestedCondition). Um einen Schnitt zwischen zwei Graphen eindeutig zu definieren, wird in Henshin das Mapping benutzt. Das Mapping bestimmt welcher Knoten in einem Graph welchem Knoten in anderem Graph entspricht. Mappings können in der Regel selbst oder in einer Anwendungsbedingung sein. Mappings in der Regel bestimmen den Schnitt zwischen LHS und RHS bzw. in Anwendungsbedingung den Schnitt zwischen Bedingungsgraph und LHS. Außerdem kann eine Anwendungsbedingung mit Hilfe von Attribute „negated“ negiert werden, und mit Hilfe von Formulaselement „And“ in einen logischen Ausdruck zusammengefasst. Eine Anwendungsbedingung oder Ausdruck aus mehreren Anwendungsbedingungen wird schließlich in dem Graph LHS gespeichert. Eine Regel kann auch mehrere Variablen (Parameter) haben, die dann für die Regelanwendung vom Benutzer angegeben oder durch die Regel selbst belegt werden und als Attributwerte benutzt werden können. So kann man die Regel flexibel definieren.

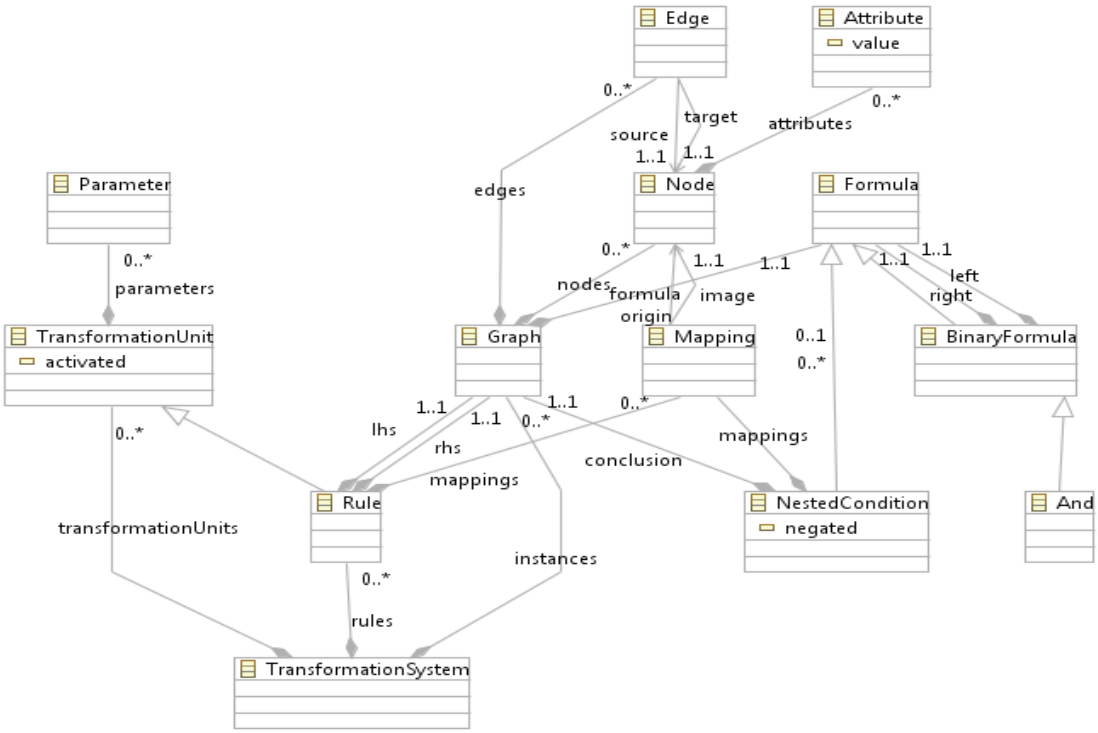

Abbildung 12. Henshinmodell aus „Visuelle Sprache“ Projekt

2.3.3. Basis-Henshineditor

Ein Visueller-Henshin-Editor wurde in Laufe des Projektes „Visuelle Sprache“ 2010 entwickelt. Henshin-Editor ist ein Eclipse Plugin, das auf GEF „Graphical Editing Framework“ und MUVITOR basiert und als Transformation Engine Henshin benutzt [3].

Graphical Editing Framework [4] ist ein Eclipse Plugin, welches eine Standardfunktionalität für einen Graphischen Editor anbietet und sich auf Prinzip Model-View-Controller basiert. Als Basis ist ein Modell notwendig, welches bei Henshin-Editor selbstverständlich Henshinmodell ist.

MUVITOR [5] steht für Multi View Editor und bietet eine erweiterte Funktionalität von GEF an. Zuerst, wie der Name schon sagt, ermöglicht der MUVITOR mehrere Ansichten in einem Editor zu haben. Als nächstes werden in MUVITOR solche Funktionalitäten, wie z.B. Copy, Cut und Paste und Exportierung einer Ansicht, als ein Bild sowie Wizard und Perspektive Vorgaben angeboten.

Ziel des Basis-Henshin-Editors war es auf Basis eines vordefinierten EMF Modells eine visuelle Erstellung und Änderung von alle Instanzen des Henshinmodells zu ermöglichen. Außerdem kann man aus dem Henshin-Editor heraus die Transformationen durchführen und EMF-Kompatibilität prüfen. Das Modell von Henshin ist für den Graphen sehr frei gemacht. Ein Knoten kann beliebig viele Attribute und eine Kante kann beliebige Knoten als Quell- bzw. Ziel-Knoten haben. Das heißt aber nicht, dass mit Henshin-Editor ein ungültiges Modell erstellt werden kann. Bei der Graph Erstellung bzw. Editierung werden für die Kanten automatisch Typen von Quell- und Ziel-Konten sowie die Multiplizitäten geprüft. Wenn eine Kante zwischen zwei Knoten erstellt werden soll, wird zuerst überprüft welchen Kantentypen zwischen den beiden Knoten existieren können und welche schon existieren. Bei mehreren Möglichkeiten wird eine Auswahl angeboten und wenn zwischen zwei Knoten nur ein Kantentyp erstellt werden kann, wird der Typ automatisch gesetzt. So wird das Modell mit Metamodell übereinstimmen. Aber auch das sagt noch lange nicht, dass das Modell EMF-kompatibel bleibt.

2.3.4. EMF-Kompatibilität

Um die Vorschriften für ein gültiges EMF-Modell zu definieren, soll zuerst ein Knoten definiert werden der von anderem Knoten des Modells sich unterscheiden wird. Dieser Knoten wird ein Wurzelknoten sein. Ein Wurzelknoten ist ein oberstes Element des Instanzen-Modells und kann nicht in anderen Knoten des Modells enthaltet sein. Dann soll ein nicht leeres gültiges EMF-Instanzen-Modell folgende drei Vorschriften erfüllen:

1. **Ein nicht leeres gültiges EMF-Instanzen-Modell hat genau einen Wurzelknoten.**
2. **Alle Knoten, außer Wurzelknoten, sollen genau einen Container haben.**

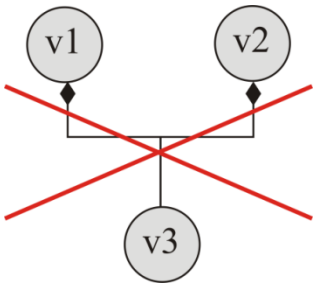

Abbildung 13. Mehrere Container sind verboten.

3. **Zyklisches Containment ist verboten.**

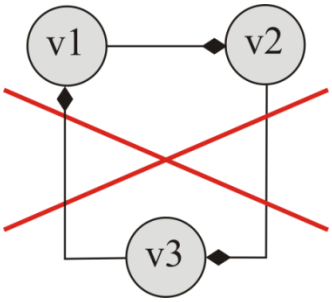

Abbildung 14. Zyklisches Containment ist verboten

Diese drei Bedingungen können im Henshin-Editor bei einem Graphen geprüft werden. So kann man sicherstellen, dass z.B. ein Startgraph EMF-Kompatibel ist. Um sicherzustellen, dass der Graph auch nach der Transformation EMF-Kompatibel bleibt, soll man nach der Transformation den Graph noch mal auf EMF-Kompatibilität prüfen lassen oder noch vor der Transformation sicherstellen, dass die Regeln die EMF-Kompatibilität nicht verletzen können. Dazu sollen die Regeln folgende fünf Vorschriften erfüllen [6]:

1. **Erstellung neuer Knoten nur mit zugehöriger „Containment“- Kante.** (Abbildung 15)

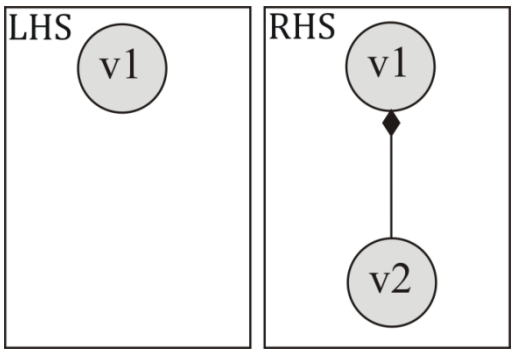

Abbildung 15. Erstellung ein neuer Knoten /neuen Kanten

2. **Löschung von Knoten nur mit zugehöriger „Containment“- Kante.** (Abbildung 16)

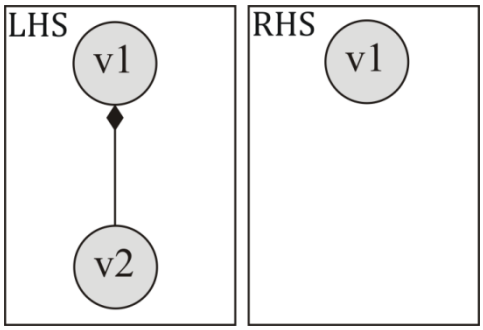

Abbildung 16. Löschung einen Knoten/ eine Kante

3. **Erstellung einer „Containment“- Kante nur mit enthaltendem Knoten.** (Abbildung 15)
4. **Löschung einer „Containment“- Kante nur mit enthaltendem Knoten.** (Abbildung 16)
5. **Knotenzugehörigkeit wechseln nur in demselben Zugehörigkeitspfad.** (Abbildung 17)

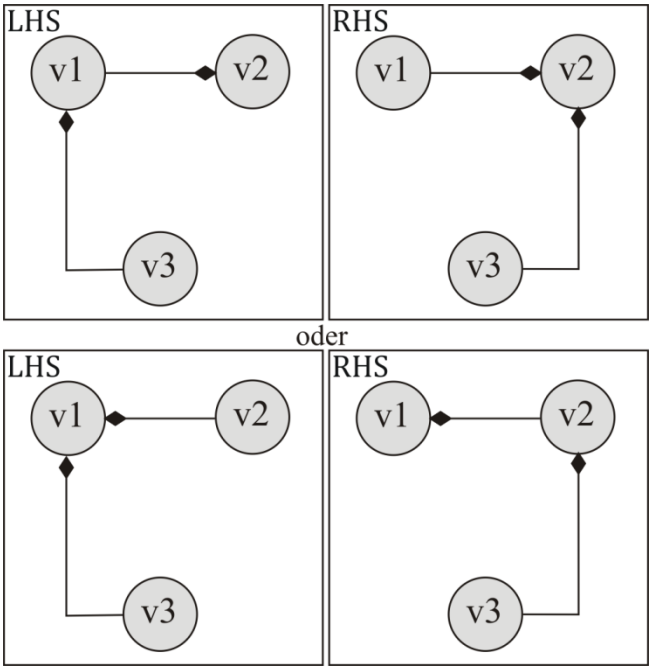

Abbildung 17. Knotenzugehörigkeit Wechsel

Die Erfüllung von den ersten vier Vorschriften stellt sicher, dass auch nach der Transformation alle Knoten genau einen Container haben werden. Fünfte Vorschrift stellt sicher, dass das Modell keine Containments-Zyklen haben wird. Was noch fehlt ist eine Vorschrift die sicherstellt, dass das Modell genau einen Wurzelknoten haben wird. Das kann man mit zwei verschiedenen Methoden sicherstellen. Erste Methode sagt,

dass der Wurzelknoten bereits im Startgraph vorhanden sein soll. Zweite Methode erlaubt einen leeren Startgraph und sagt, dass eine Axiom-Regel definiert werden soll, die einen Wurzelknoten erstellet. Außerdem soll die Axiom-Regel als erste und nur einmal ausgeführt werden.

2.3.5. Editor-Ansicht des Basis-Henshin-Editors

Die Ansicht von dem Henshin-Editor ist auf der Abbildung 18 zu sehen. Henshin-Editor basiert auf einem Baumbasierten Editor und bietet einen baumbasierten Überblick (Baum-Ansicht) vom gesamten Modell (Teil 1 von Abbildung 18). Oberstes Element ist hier, dem Model entsprechend, ein Transformationssystem. Ein Transformationssystem hat einen Ordner „Imported EPackage(s)“, wo alle importierten Modelle angezeigt werden und, wie aus Henshin-modell zu sehen ist, die Graphen und die Regeln. Im angezeigten Beispiel ist nur ein Modell „pizzaService“ importiert worden. Dann kann man einen Graph „Start Graph“ mit seinen Knoten und Kanten sehen. Unter dem Graph sind zwei Transformationsregeln „createPizza“ und „AddOrderItem“. Die Regel „createPizza“ soll eine neue Instanz von der Klasse „PizzaKind“ erstellen und einen vom Benutzer angegebenen Wert für das Attribut „price“ setzen. Dafür wurde ein Parameter „PizzaPreis“ in der Regel definiert der bei der Regel-Ausführung von Benutzer angegeben werden soll. Die Regeldefinition kann man auch in einer Extra-Ansicht anzeigen lassen, wenn man ein Doppelklick auf die entsprechende Regel in der Baum-Ansicht macht. Dabei wird eine Regel-Ansicht, die man in Teil 3 von Abbildung 18 sehen kann, geöffnet. Diese Regel hat keine extra definierten Anwendungsbedingungen und besteht nur aus zwei Graphen „LHS“ und „RHS“. Den Schnitt zwischen den beiden Graphen, der mit Hilfe von Mapping definiert ist, kann man über die Farben des Knotens erkennen. So sieht man, dass die Knoten „PizzaService“ in den beiden Graphen die gleiche Farbe haben, was bedeutet, dass dieser Knoten im Schnitt enthalten ist. Die Knoten die kein Mapping haben, haben graue Farbe. Außer den Graphen hat die Regel-Ansicht eine Palette und eine Toolbar. Die Palette hat unten eine kleine Ansicht vom momentan aktiven Graph und den Elementen die man für die Regeldefinition benutzen kann. Die Toolbar bietet eine Regelvalidierung, Regel-Ausführung, sowie die Kopierung von LHS Graph zu eine Anwendungsbedingung bzw. RHS Graph an. Die Regelvalidierung prüft ob die Regel die EMF-Kompatibilität verletzt. Die Regel-Ausführung führt aktuelle Regel aus, wobei

wenn mehrere Graphen in ein Transformationssystem existieren eine Graph-Auswahl angeboten wird. Ähnlich zu der Regel-Ansicht, hat der Henshin- Editor eine Graph-Ansicht, die in Teil 2 von Abbildung 18 zu sehen ist. Diese Ansicht hat auch eine Palette für die Graph-Erstellung bzw. Editierung und eine Toolbar, die die Graph-Validierung und die Regelausführung anbietet. Auch wie bei der Regel- Ausführung aus der Regel-Ansicht, wird bei Existenz mehreren Regeln eine Regel-Auswahl angeboten.

Der Henshin-Editor hat noch einige Ansichten die in der Eclipse Standardansicht integriert sind. Wichtig für den Benutzer sind die Ansichten „Navigator“ (Teil 4 von Abbildung 18) und „Properties“ (Teil 5 von Abbildung 18). Die Ansicht „Navigator“ ermöglicht die Navigation in den Projekten vom aktuellen Workspace. Die Ansicht „Properties“ stellt die Eingeschalten von dem aktuell ausgewählten Objekt dar.

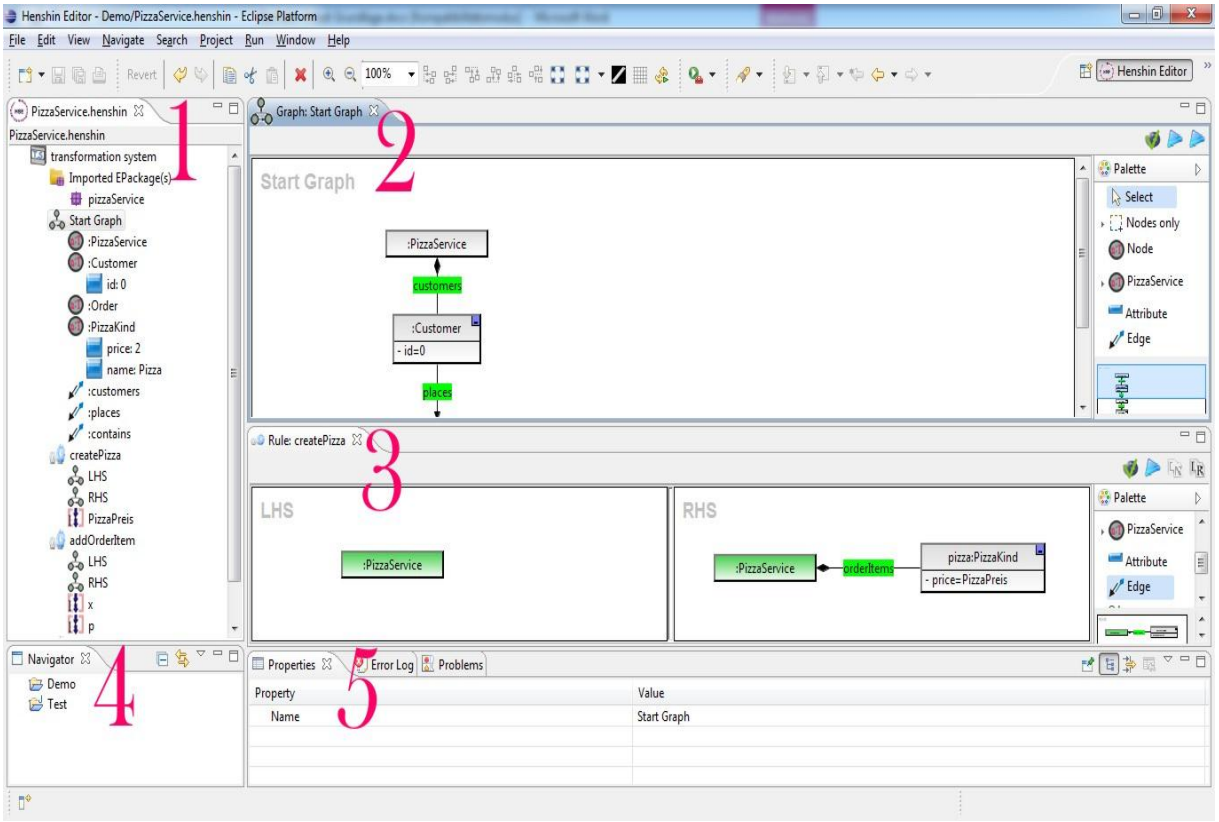

Abbildung 18. Ansicht des Henshin Editors

3. Erweiterung des Henshin-Editors um automatisch gesteuerte Transformationen

Die Aufgabe dieser Arbeit ist die Erweiterung des Henshin-Editors um automatisch gesteuerte Transformationen zu ermöglichen. Für die komplexeren Transformationsaufgaben ist die manuelle Ausführung von einzelnen Transformationsregeln nicht genügend. Erstens fordert es bestimmte Kenntnisse von der Person die eine Transformation ausführen will. Bei vielen Fällen muss man den aktuellen Modellzustand analysieren um die nächste Regel auszusuchen. Letztendlich sollen einige Transformationen nicht deterministisch laufen, was bei einer manuellen Transformation nicht möglich ist. So sind die Anforderungen an automatisch gesteuerte Transformationssysteme formuliert. Es soll möglich sein, ein Henshin-Transformationssystem so zu definieren und zu speichern, dass der Benutzer nichts mehr machen muss als die Datei zu öffnen und die Transformation zu starten. Alle vom Benutzer benötigten Daten werden vor der Transformation und nicht während der Transformation abgefragt. Transformation soll nicht nur automatisch sondern auch gesteuert durgeführt werden. Es soll auch möglich sein, ein Teil oder ganze Transformation nicht deterministisch durchzuführen. Wenn die Transformation nicht wie geplant durchgeführt werden kann, wird der Benutzer benachrichtigt und alle während der Transformation durchgeführten Änderungen werden zurückgesetzt.

3.1. Modellerweiterung

Um die gestellte Aufgabe zu lösen, wurden zum Henshinmodell Transformation Units eingefügt (Abbildung 19). Als eine elementare Transformation Unit wurde eine Transformationsregel genommen. Außerdem kann jetzt eine Regel mehrere „AttributeCondition“ haben, die die Regelausführung bei bestimmten Attributwerten verbieten können. Andere Transformation Units verursachen selbst keine Änderungen im Modell, sondern enthalten andere Transformation Units, die am Ende Transformationsregel sind, und steuern deren Ausführung. Für die Steuerung von der Ausführung, können auch, wie bei einer Transformationsregel, Parameter benutzt werden. Die Werte der Parameter von einer Transformation Unit, die vom Benutzer

ausgeführt wird, können vom Benutzer angegeben oder während der Ausführung belegt werden. Eine Transformation Unit kann ihre Parameterwerte und die in ihr enthaltene Transformation Unit übergeben, oder andersrum, Ihre Parameterwerte übernehmen. Für die Bestimmung der Werteübergaberichtung, hat eine Transformation Unit ein Parameter-Mapping. Mit Hilfe von Parametern, die früher nur für Attributen benutzt wurden, können jetzt die Knoten von verschieden Regel miteinander übereinstimmen. Zum Beispiel wenn in einer Regel ein Parameter mit dem gleichen Namen wie von einem Knoten (beispielsweise x) aus dieser Regel vorhanden ist, dann wird dieser Parameter nach der Regelanwendung mit einem Knoten aus dem Graph belegt der von Knoten x abgebildet wird. Wird ein solcher Parameter vor der Ausführung der Regel belegt, wird dann genau bestimmt auf welchem Graph-Knoten dieser Regel-Knoten abgebildet werden soll. Nach der Ausführung einer Transformation Unit wird ein boolescher Wert zurückgeliefert, mit dessen Hilfe kann die Transformation Unit, die die ausgeführt hat, weitere Entscheidungen treffen. Eine Transformation Unit kann auch leicht (mit Hilfe von dem Attribut „activated“) ausgeschaltet bzw. eingeschaltet werden.

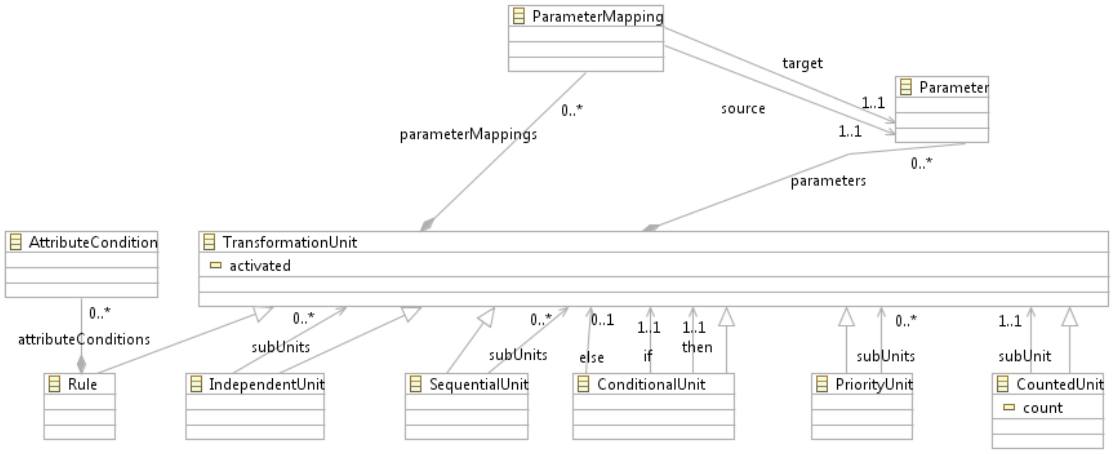

Abbildung 19. Transformation Units

3.2. Transformation Units

3.2.1. Counted Unit

Eine Counted Unit ermöglicht eine in ihr enthaltene Transformation Unit mehrmals auszuführen. Als die Ausführungsanzahl kann der Benutzer einen positiven Integer Wert oder -1 angeben. Wenn die Ausführungsanzahl -1 ist, wird die untergeordnete Unit so lange wie es geht ausgeführt. Nach jeder erfolgreichen Ausführung werden alle Parameter die von ausgeführter Transformation Unit belegt werden sollen neu belegt. Die Ausführung von Counted Unit wird als erfolgreich gezählt, wenn untergeordnete Transformation Unit so oft erfolgreich ausgeführt wurde wie die vom Benutzer angegebene Ausführungsanzahl. Wenn die Ausführungsanzahl -1 beträgt, dann wird die Ausführung von Counted Unit immer als erfolgreich gezählt. Falls die Ausführung nicht erfolgreich war, werden alle Änderungen zurückgesetzt und „false“ als Ergebnis zurückgeliefert. Wenn alle Ausführungen erfolgreich waren, wird entsprechend „true“ zurückgeliefert.

3.2.2. Sequential Unit

Eine Sequential Unit kann mehrere Transformation Units enthalten und sie in der entsprechenden Reihenfolge ausführen. Eine Ausführung von einer Sequential Unit ist nur dann erfolgreich, wenn alle in ihr enthaltene Transformation Units erfolgreich ausgeführt worden sind. Bei einer nicht erfolgreichen Ausführung werden alle Änderungen zurückgesetzt.

3.2.3. Conditional Unit

Eine Conditional Unit besteht aus drei Segmenten: „if“, „then“ und „else“. Jedes Segment kann nur eine Transformation Unit enthalten. Segment „else“ muss nicht belegt werden. Es wird zuerst die Transformation Unit des Segments „if“ ausgeführt und entsprechend dem Ergebnis wird dann die Transformation Unit des Segments „then“ oder des Segments „else“ (falls vorhanden) ausgeführt. Als Ergebnis wird das Ergebnis von der zuletzt ausgeführte Transformation Unit zurückgeliefert. Falls das Ergebnis „false“ ist, werden alle Änderungen zurückgesetzt.

3.2.4. Independent Unit

Eine Independent Unit ermöglicht eine Ausführung von einer zufälligen Transformation Unit die in ihr enthalten ist. Wenn keine untergeordnete Transformation Units ausgeführt werden können, wird als Ergebnis „false“ zurückgeliefert, ansonsten „true“.

3.2.5. Priority Unit

Eine Priority Unit ist eine Unit in der alle enthaltene Transformation Units unterschiedliche Prioritäten haben. Die Prioritäten werden absteigend von der ersten bis zur letzten enthalten Transformation Unit gesetzt. Eine Priority Unit wird eine ausführbare Unit mit der höchsten Priorität ausführen. Wenn keine untergeordnete Transformation Units ausgeführt werden können, wird als Ergebnis „false“ zurückgeliefert, ansonsten „true“.

3.3. Realisierung eines Editors für Transformation Units

Um eine Transformation zu realisieren, werden oft mehrere Transformation Units benutzt. Dabei kann man schnell den Überblick verlieren. Dafür werden Transformation Units die nicht in anderen Transformation Units enthalten sind in der Baum-Ansicht gleich nach den Transformationsregeln angezeigt (Abbildung 20). Diese Units werden normalerweise für den Transformationsstart benutzt. Am Ende wird auch ein Ordner „Transformation Units“ angezeigt. Dieser Ordner enthält alle erstellten Transformation Units und kann per Doppelklick geöffnet bzw. geschlossen werden. Weil viele Transformation Units andere Transformation Units enthalten, ist es erlaubt eine Transformation Unit mit dem Inhalt zu erstellen. Dafür soll der Benutzer bereits existierende Transformation Units die in der neuen Unit enthalten sein sollen auswählen und im Kontextmenü „create transformation unit with content“ den Typ von der neuen Transformation Unit auswählen. Wenn das in einer Ansicht von Transformation Unit gemacht wird, werden die bereits ausgewählten Units durch die neuerstellte Unit ersetzt. Dabei wird das existierende Parameter Mapping über in der neuen Unit automatisch erstellte Parameter weitergeleitet.

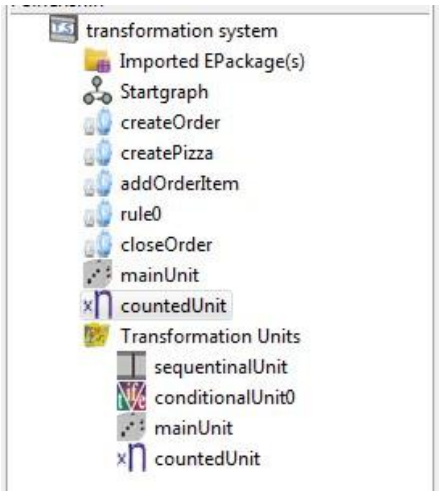

Abbildung 20.Transformation Units: Baum-Ansicht

Eine Ansicht von einer Transformation Unit sieht folgenderweise aus (Abbildung 21). Der oberste Teil von einer Transformation Unit enthält ein Icon, einen Namen und eine Checkbox „Activated“, die die Transformation Unit aktiviert bzw. deaktiviert. Den Typ der Unit kann man leicht über das oben genannte Icon oder über die Farbe der Unit erkennen. Restlicher Teil der Unit wird senkrecht in zwei Teile geteilt. Der linke Teil enthält alle Parameter der Unit und der rechte Teil alle in ihr enthaltenen Units. Dabei werden die Parameter Mappings mit den Pfeilen und entsprechenden Richtungen[2] gezeigt. Untergeordnete Units können per Doppelklick geöffnet werden. Dabei wird rechts in der Ansicht die ausgewählte Unit angezeigt (Abbildung 22). In dieser Ansicht kann man auch die Mappings der Parameter über ihre Farbe erkennen. Hier kann man auch neue Parameter-Mappings erstellen. Auch in dieser Ansicht kann man eine untergeordnete Unit öffnen. Dabei wird die Unit aus rechten Ansicht links gezeigt und die neu geöffnete Unit rechts. So kann man in dieser Ansicht beliebig tief und zurück nach oben gehen. Um zurück nach oben zu gehen, macht man einen Doppelklick auf den oberste Teil der links angezeigten Unit. Wenn diese Unit nicht die oberste Unit aus dieser Ansicht ist, wird diese Unit rechts und die Unit in der sie enthalten ist wieder links angezeigt.

[2] Pfeilrichtung: von Output-Unit zum Parameter bzw. vom Parameter zur Input-Unit.

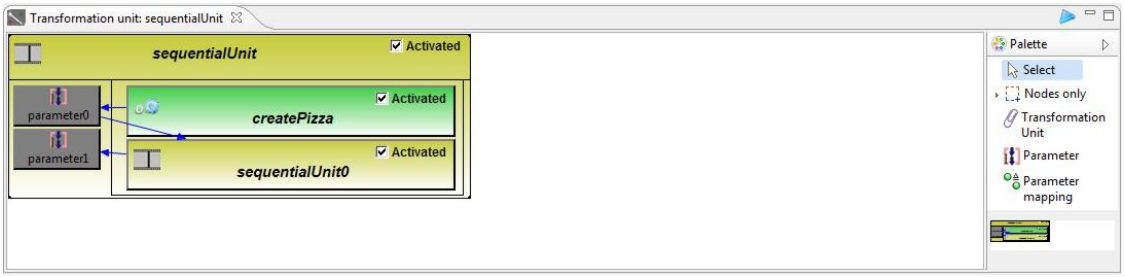

Abbildung 21. Transformation Unit Ansicht

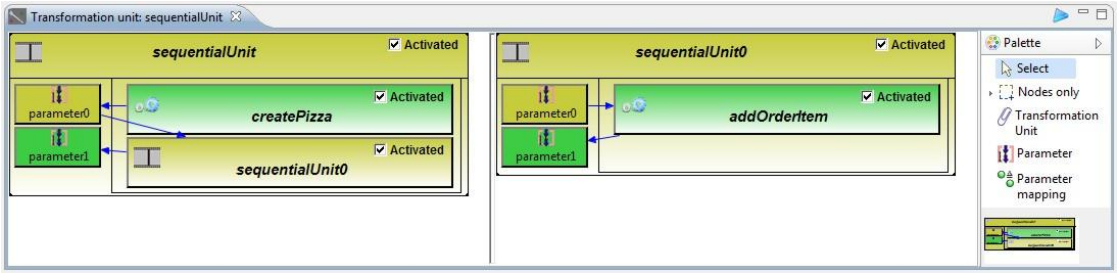

Abbildung 22. Transformation Unit mit untergeordnetem Unit

Wenn man eine neue Transformation Unit hinzufügen will, kann man in der Palette den Punkt „Transformation Unit“ auswählen und auf die gewünschte Position in der Transformation Unit klicken. Wenn man eine neue Unit ganz am Ende hinzufügen will, soll man auf den oberste Teil der Unit klicken. Dabei werden alle mögliche Transformation Units angeboten. Weil die zyklische Einkapselung für Transformation Units verboten ist, können einige Units aus der Auswahl ausgeschlossen werden. Die Reihenfolge der Units ist mit der sogenannten „Drag and drop“ Methode beliebig konfigurierbar. Der Punkt „Parameter“ in der Palette ermöglicht einen neuen Parameter zu erstellen und der Punkt „Parameter mapping“ erstellt ein neues Mapping zwischen den Parameter aus einer Transformation Unit und den Parametern aus der Unit die ihr untergeordnet ist. Die Toolbar für eine Transformation Unit besteht nur aus einem Punkt, der diese Unit Ausführen lässt.

Einige Transformation Units unterscheiden sich ein wenig von anderen. So zeigt z.B. eine Counted Unit auch die Ausführungsanzahl an (Abbildung 23).Eine Conditional Unit besteht aus drei Teilen „if“, „then“ und „else“ (Abbildung 24).

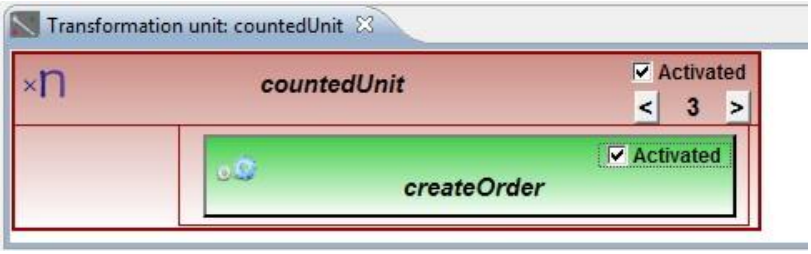

Abbildung 23. Counted Unit

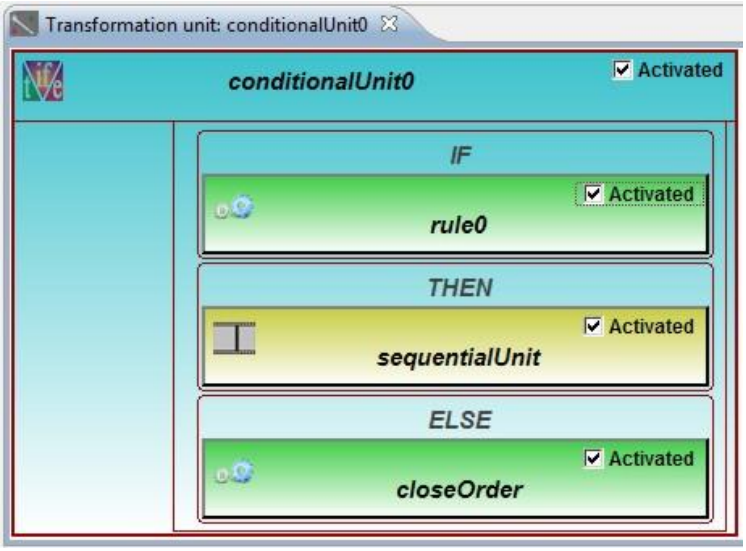

Abbildung 24. Conditional Unit

3.4. Verwendungsbeispiel „Personal Mobility Manager“

Das Bespiel „Personal Mobility Manager“ ist ein Modell eines Web-Services. Das Ziel ist eine Route zwischen zwei Orten zu finden und entsprechend der Verkehrssituation ein Transportmittel vorzuschlagen. Das Metamodell ist auf der Abbildung 25 zu sehen.

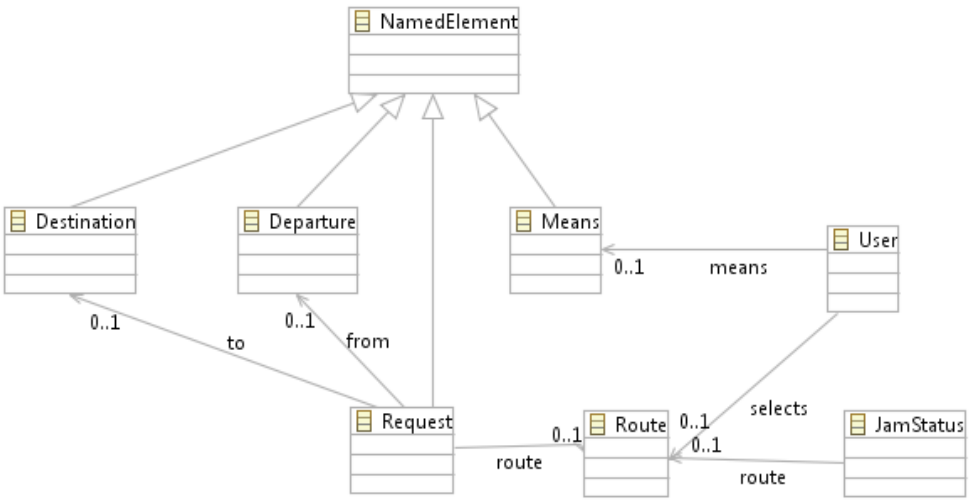

Abbildung 25. "Personal Mobility Manager" Metamodell

Dieses Beispiel soll eine Simulation durchführen. Dabei sind Startort und die mögliche Transportmittel in Startgraph festgelegt. Der Zielort wird vom Benutzer angegeben. Client-Service Kommunikation wird durch Objekten-Zuweisung simuliert. Startgraph ist auf der Abbildung 26 zu sehen. Als mögliche Transportmittel sind ein Auto und ein Fahrrad definiert.

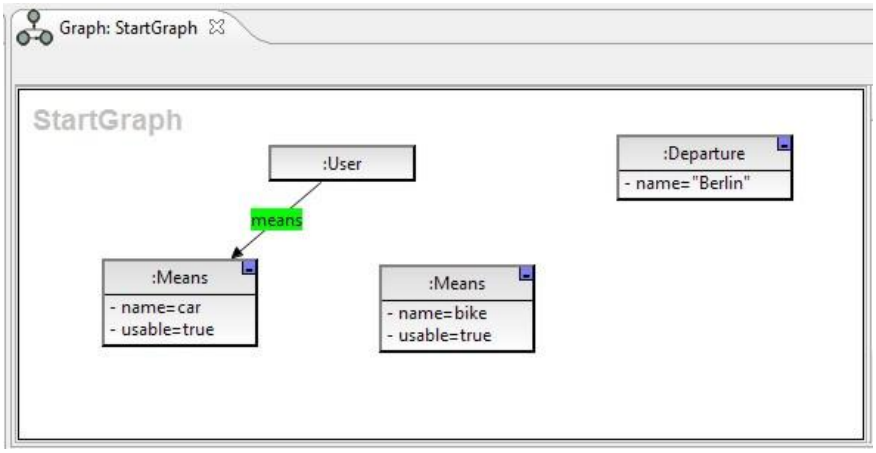

Abbildung 26. "Personal Mobility Manager" Startgraph

Als mögliche Aktionen sind folgende Transformationsregeln definiert.

- „ChooseDestination“ erzeugt einen neuen Zielort mit übergebenem Namen.

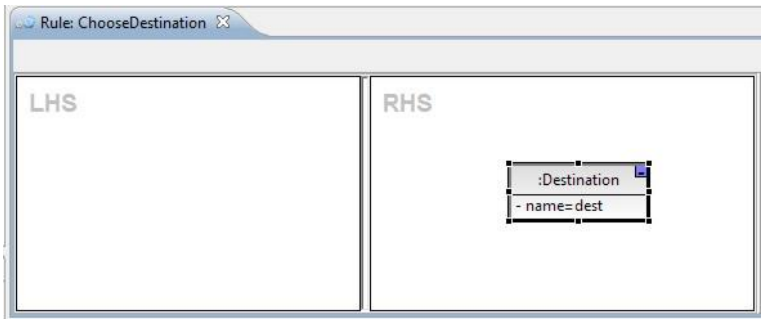

Abbildung 27. Rule"ChooseDestination"

- „RequestRouteMap“ stellt eine Anfrage über eine mögliche Route zwischen den Start- und Ziel-Orten.

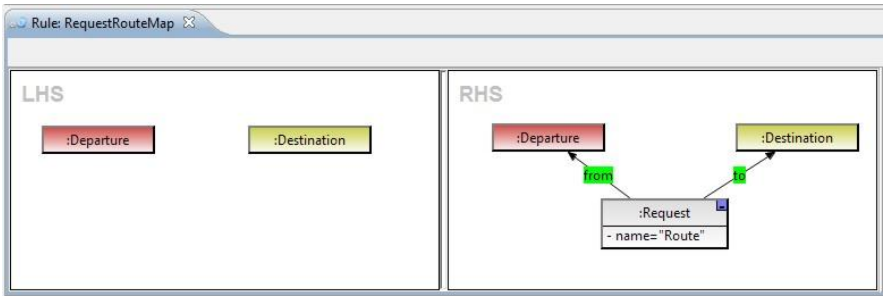

Abbildung 28. Rule "RequestRouteMap"

- „ResponseRouteMap“ liefert als Ergebnis an die gestellte Anfrage eine neue Route zurück.

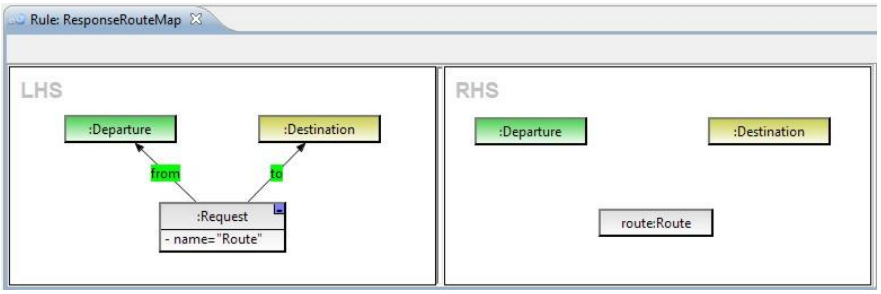

Abbildung 29. Rule "ResponseRouteMap"

- „RequestJamStatus“ stellt eine Anfrage über die Verkehrssituation an einer bestimmten Route.

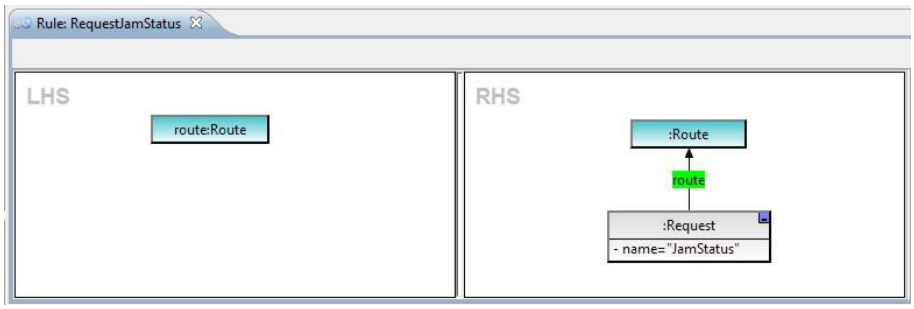

Abbildung 30. Rule"RequestJamStatus"

- „ResponseJamStatus“ liefert als Ergebnis an die gestellte Anfrage den Status der Verkehrssituation zurück (in der Simulation einen zufällig ermittelten booleschen Wert).

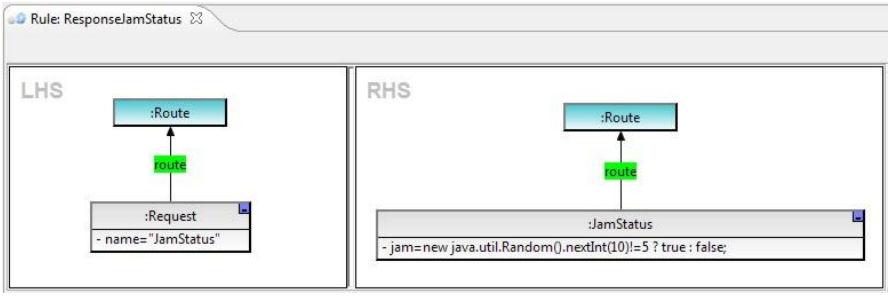

Abbildung 31. Rule "ResponseJamStatus"

- „NotAllRoutesJammed“ ist eine extra Regel, die nichts am Graphen ändert sondern nur den Routen-Status überprüft. Dafür sind zwei Anwendungsbedingungen formuliert. Erste Anwendungsbedingung (Abbildung 33) ist negiert und prüft ob das was in der Bedingung definiert ist nicht existiert. Die Bedingung ist erfüllt wenn eine Route existiert, die keinen Jam- Status hat. Zweite Anwendungsbedingung (Abbildung 34) prüft ob eine Route existiert, dessen Jam- Status „false“ ist. Da die Anwendungsbedingungen mit „OR“ verknüpft sind, reicht es wenn nur eine von beiden erfüllt wird. Insgesamt ist die Rückgabe von NotAllRoutesJammed „true“, wenn eine Route ohne JamStatus-Knoten existiert oder wenn einer der JamStatus-Knoten den Attributwert „jam“=„false“ hat.

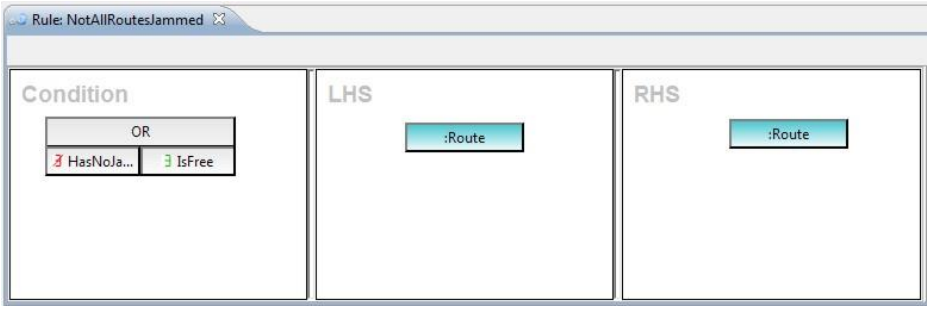

Abbildung 32. Rule "NotAllRoutesJammed"

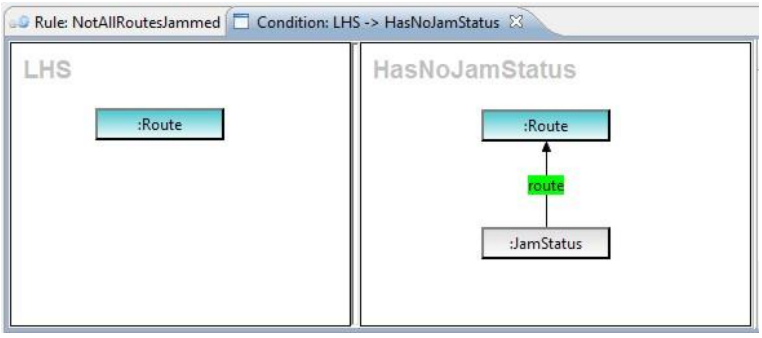

Abbildung 33. Erste Anwendungsbedingung von "NotAllRoutesJammed"

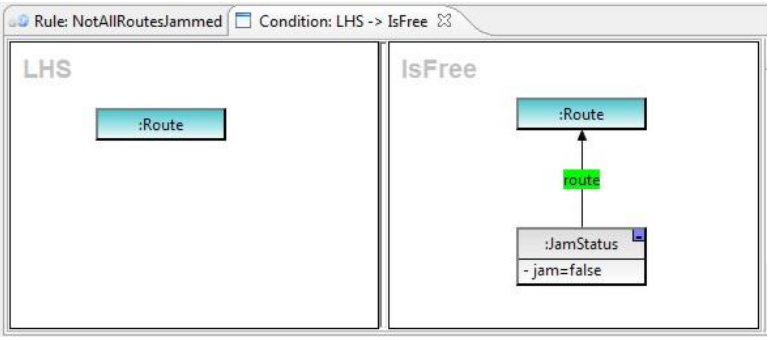

Abbildung 34. Zweite Anwendungsbedingung von "NotAllRoutesJammed"

- „ForbidCar“ – weist die Benutzung des Autos ab.

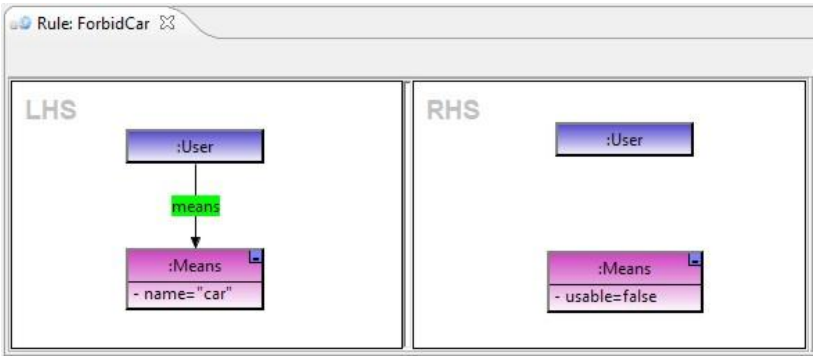

Abbildung 35. Rule "ForbidCar"

- „SelectBike“ wählt eine Route und das Fahrrad als Transportmittel aus.

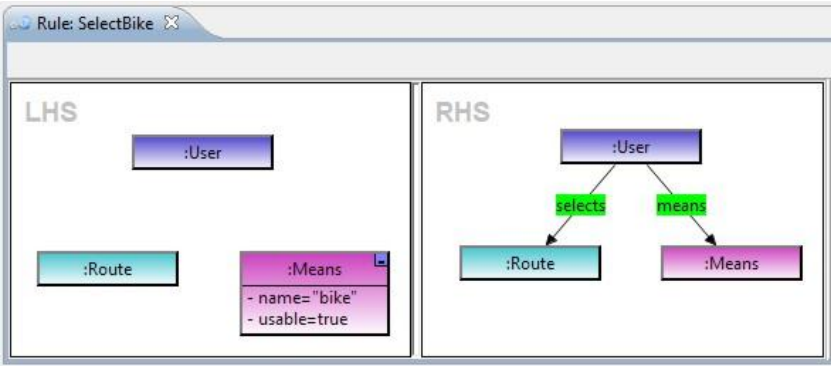

Abbildung 36. Rule "SelectBike"

- „ChooseRoute“ wählt eine Route die frei ist oder keinen Jam- Status hat aus. Um festzustellen welche Route ausgewählt werden soll, sind zwei Anwendungsbedingungen definiert. Die Anwendungsbedingungen sind ähnlich wie in die Regel „NotAllRoutesJammed“ definiert.

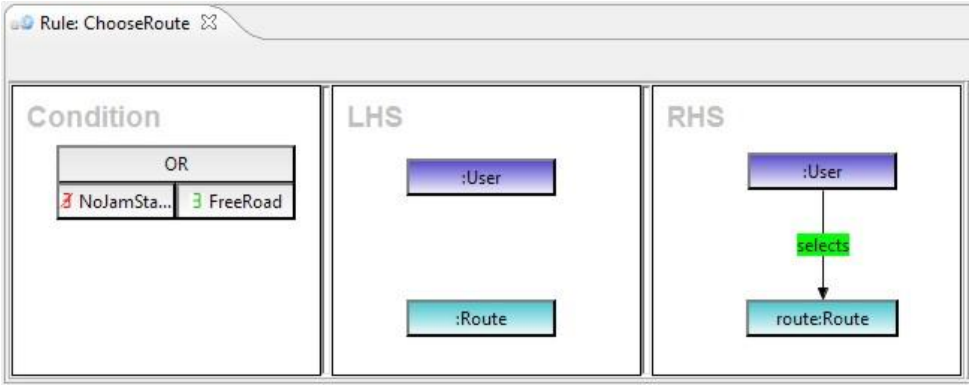

Abbildung 37. Rule "ChooseRoute"

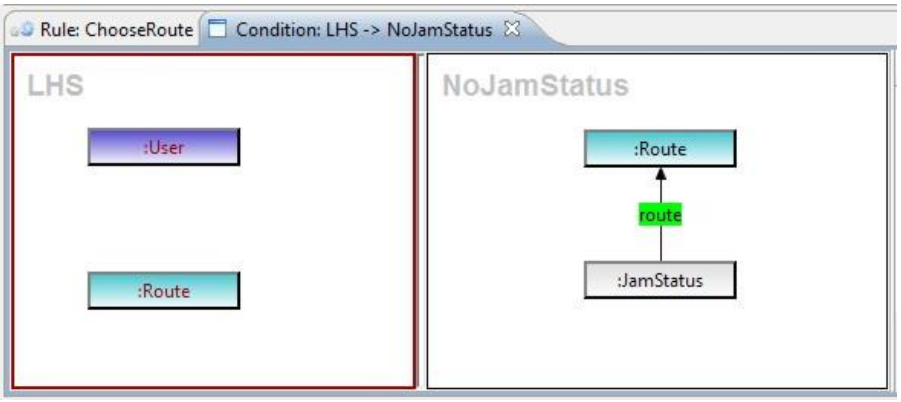

Abbildung 38. Erste Anwendungsbedingung von "ChooseRoute"

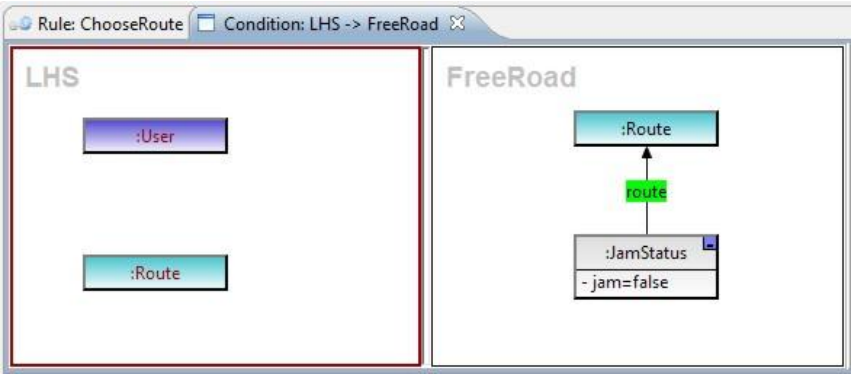

Abbildung 39. Zweite Anwendungbedingung von "ChooseRoute"

- „DeleteJamStatus“ löscht Jam- Status von einer Route.

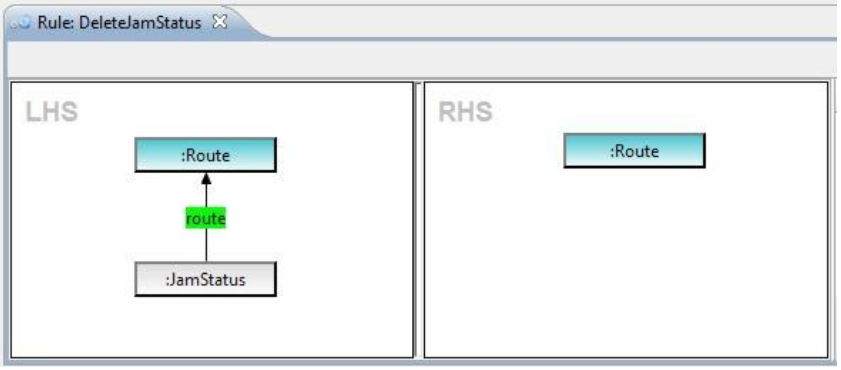

Abbildung 40. Rule "DeleteJamStatus"

- „DeleteUnusedRoute“ löscht eine nicht ausgewählt Route. Ob die Route gelöscht werden darf, wird mit Hilfe von den Anwendungsbedingungen festgestellt. Dabei sind beide Bedingungen negiert und mit „AND“ verknüpft. Erste Bedingung prüft ob die Route keinen Jam-Status hat und die zweite ob die Route nicht ausgewählt wurde.

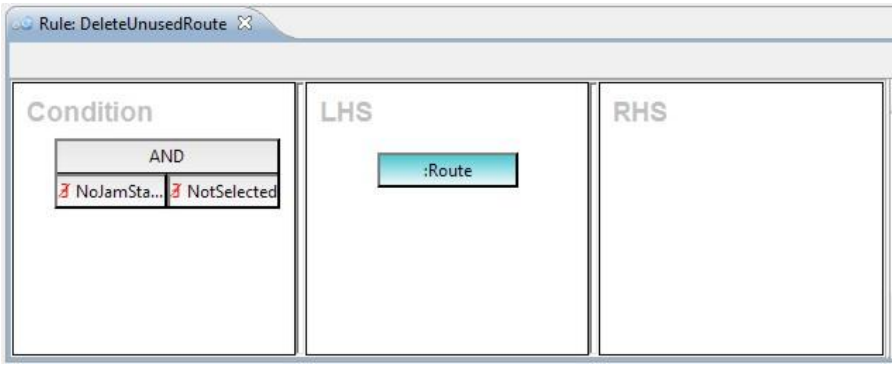

Abbildung 41. Rule "DeleteUnusedRoute"

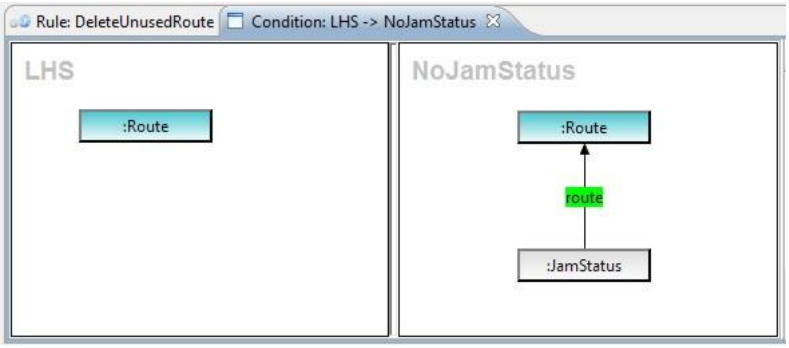

Abbildung 42. Erste Anwendungsbedingung von "DeleteUnusedRoute"

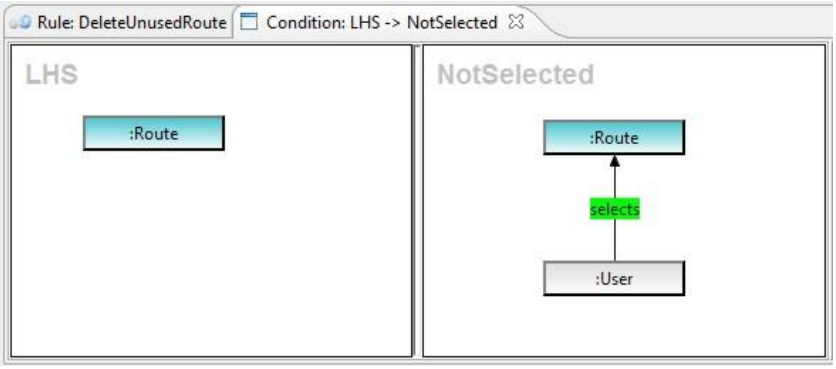

Abbildung 43. Zweite Anwendungsbedingung von "DeleteUnusedRoute"

Die gesamte Simulation wird mit einer Sequential Unit „mainUnit“ (Abbildung 44) ausgeführt. Außerdem sind in der „mainUnit“ zwei Parameter „dest“ und „route“ definiert. Der Parameter „dest“ wird vom Benutzer mit dem Namen vom Zielort festgelegt und in Transformationsregel „ChooseDestination“ weiter übergegeben. Der Parameter „route“ wird während der Ausführung von der Conditional Unit „decideMeans“ mit der ausgewählten Route festgelegt.

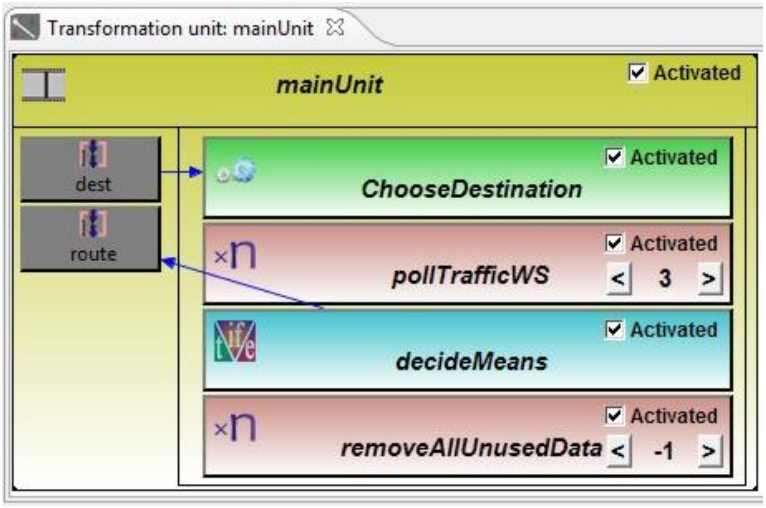

Abbildung 44. Sequential Unit "mainUnit"

Zuerst führt die „mainUnit“ eine Transformationsregel ChooseDestination, die einen neuen Zielort mit übergebenem Namen erzeugt. Dann wird eine Counted Unit „pollTrafficWS“, die dreimal die Sequential Unit „trafficWS“ ausführt, ausgeführt. (Abbildung 45)

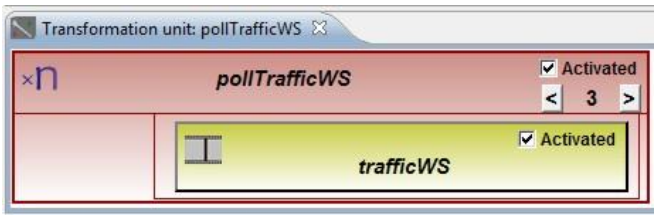

Abbildung 45. Counted Unit "pollTrafficWS"

Die Sequential Unit „trafficWS“ ihrerseits, sucht eine neue Route und bestimmt die Verkehrssituation auf dieser Route (Abbildung 46). Eine neue Route wird folgendermaßen gesucht. Es wird eine Anfrage über „RequestRouteMap“ an den Web-Service geschickt, der als Antwort über „ResponseRouteMap“ eine neue Route liefert. Diese Route wird dann über den Parameter „route“ der „trafficWS“ Unit in die Anfrage

über die Verkehrssituation „RequestJamStatus“ benutzt. Die Antwort „ResponseJamStatus“ für die gestellte Anfrage wird dann der Verkehrssituation entsprechen.

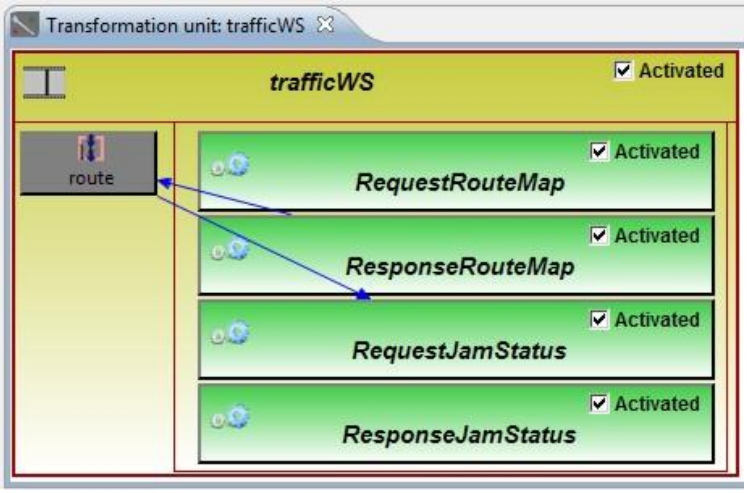

Abbildung 46. Sequential Unit "trafficWS"

Nachdem drei mögliche Routen mit den entsprechenden Verkehrssituationen bestimmt wurden, soll nun eine Route und ein Transportmittel ausgewählt werden. Das wird in der Conditional Unit „decideMeans“ (Abbildung 47) realisiert. Dabei wird überprüft ob nicht auf allen Routen Stau herrscht, und wenn das der Fall ist, wird eine freie Route ausgewählt. Sonst wird das Fahrrad als Transportmittel ausgewählt (Abbildung 48).

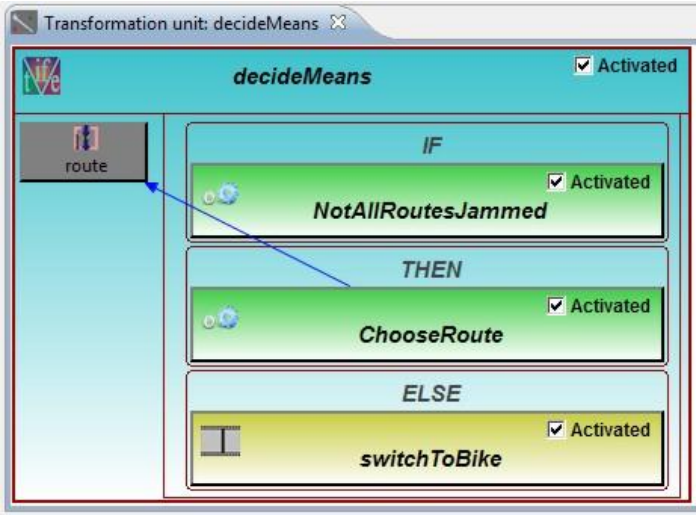

Abbildung 47. Conditional Unit "decideMeans"

Abbildung 48. Sequential Unit "switchToBike"

Nachdem die Route ausgewählt wurde, können nicht benutzte Daten gelöscht werden. Es wird mit einer Counted Unit „removeAllUnusedData“ mit der Ausführungsanzahlgleich -1 eine Independent Unit „removeUnusedData“ so lange wie möglich ausgeführt.

Abbildung 49. Counted Unit "removeAllUnusedData"

Independent Unit „removeUnusedData“ löscht ihrerseits einen Jam Status (DeleteJamStatus) oder eine Route die keinen Jam Status hat und nicht in Benutzung ist.

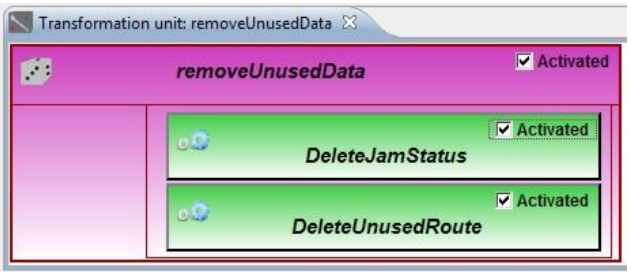

Abbildung 50. Independent Unit "removeUnusedData"

Das „Personal-Mobility-Manager“-Beispiel zeigt gut wie Transformation Units sinnvoll eingesetzt werden können, um den Regelfluss zu steuern. Weitere Beispiele sind die Steuerung von Refactorings von EMF-Modellen [3] und die Steuerung von automatischen Modelltransformationen [7].

4. Interne Architektur der Erweiterungen des Henshin-Editors

4.1. Umsetzung des Model-View-Controller Prinzip

Entsprechend der GEF Architektur wurde Henshin-Editor mit dem sogenannten „Model-View-Controller“ Prinzip realisiert. Entsprechende Verteilung und das Kommunikationsprinzip ist in der Abbildung 51 dargestellt. Als Modell wird Henshinmodell selbst genommen das mit EditParts kommunizieren kann. Für jedes Modell-Element, das im Editor dargestellt werden soll, wird mindestens ein EditPart erzeugt. EditPart kennt sein Modell-Element und bekommt von dem seine Ereignisse in einer Nachricht (EMF Notifier) damit der entsprechend reagieren kann. Zum Beispiel kann es seine Ansicht aktualisieren damit die Änderungen des Modell-Elements dargestellt werden. Die Änderungen des Modells werden über Ausführung der „Commands“, die ein EditPart angewiesen hat, verursacht. Der Benutzer kommuniziert mit View. Die Aktionen, die der Benutzer aufgerufen hat, werden in EditParts übergeben, damit der EditPart entsprechend reagieren könnte. Nehmen wir an, der Benutzer will einen Knoten in einem Graph löschen. Er selektiert diesen Knoten und drückt auf Delete Taste. Dabei bekommt der EditPart von dem ausgewählten Knoten einen Event von der View und erstellt entsprechende „Command“ um den Knoten zu löschen. Ausführung von dieser „Command“ löscht den Knoten aus dem Graph. Dabei werden die Änderungen im Modell(Graph) in einer Nachricht zu EditPart des Graphs geschickt, damit er seine View aktualisiert und die Figur des gelöschten Knoten löscht.

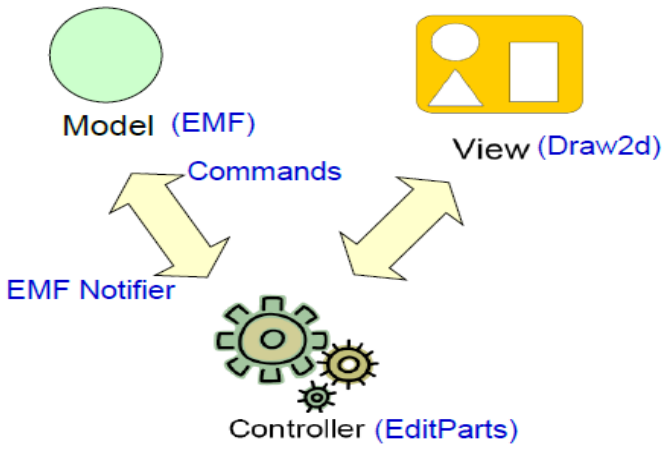

Abbildung 51. Model-View-Controller Prinzip

Der Erweiterung entsteht dann aus folgenden neuen Packages:

- **henshineditor.ui.transformation_unit** – entspricht einer View für Transformation Unit mit seiner Palette und Kontextmenü.
- **henshineditor.figure.transformation_unit**– die Figuren von der Transformation Unit View .
- **Henshineditor.actions.transformation_unit** – Aktionen die den Benutzern in einer Kontextmenü oder Toolbar angeboten werden.
- **henshineditor.model.preoperties.transformation_unit**– entspricht einem Controller, der für Properties View zuständig ist.
- **henshineditor.editparts.transformation_unit**– entspricht einem Controller der Transformation Unit, der für Transformation Unit View zuständig ist.
- **henshineditor.editparts.transformation_unit.parameter**– entspricht einem Controller des Parameters, der für Transformation Unit View zuständig ist.
- **henshineditor.editparts.tree.transformation_unit**– entspricht einem Controller der Transformation Unit, der für Baum-Ansicht zuständig ist.
- **henshineditor.editparts.tree.transformation_unit.parameter**– entspricht einem Controller des Parameters, der für Baum-Ansicht zuständig ist.
- **henshineditor.commands.transformation_unit**– „Commands“, die ein Controller für die Erstellung bzw. Änderung der Transformation Units benutzen kann.
- **henshineditor.commands.transformation_unit.parameter**–„Commands“, die ein Controller für die Erstellung bzw. Änderung der Parameter benutzen kann.

Außerdem wurden folgende Packages erweitert:

- **henshineditor.actions.tools**– es wurde die Tools für die Parameter Mapping Erstellung und Hinzufügung einer Transformation Unit erweitert.
- **henshineditor.ui.dialog**–neue Dialogen für die Transformation Unit wurden hinzugefügt.
- **henshineditor.ui.graph**– Graph-Ansicht wurde erweitert um die Historie der angewendeten Transformation Units anzeigen zu können.
- **henshineditor.editparts.graph**– es wurden die „Controllers“ für die Historie den angewendeten Transformation Units hinzugefügt.

4.2. Schnittstelle zur Transformation Engine von Henshin

Bei der Ausführung einer Transformation Unit wird eine Action „ExecuteTransformationUnit“ ausgeführt. Für die Transformationsausführung sind zwei Argumente (ein Graph und eine Transformation Unit) nötig. Eins von beiden wird automatisch zugeordnet, weil der Benutzer diese Action nur aus dem Graph- oder aus der Transformation Unit-Ansicht aufrufen kann. Wenn es für zweites Argument mehrere Möglichkeiten gibt, wird dem Benutzer eine Auswahl angeboten. Nachdem beide Argumente bestimmt worden sind, wird es geprüft ob die Transformation Unit Parameter enthält. Wenn das der Fall ist, wird vom Benutzer abgefragt, ob die Parameter belegt werden sollen. Eine Transformation verursacht die Änderungen des Modells und wird deshalb, wie bei allen Modelländerungen, als „Command“ „ExecuteTransformationUnitCommand“ ausgeführt. In dieser „Command“ ist die Schnittstelle zu Henshin Transformation Engine realisiert. Aus dem Graph wird ein *EmfGraph* erstellt, der in die *EmfEngine* übergeben wird. Von der Transformation Unit wird eine *unitApplication* erstellt, die als Parameter *EmfGraph* und eine Transformation Unit bekommt und die Transformationen starten lässt. Nach der Transformation wird ein boolescher Wert, der den Erfolg der Ausführung nachweist, zurückgeliefert. Die ausgeführte *unitApplication* hat am Ende der Ausführung eine Liste *appliedRules*, die die angewendeten Transformationsregeln in der Anwendungsreihenfolge enthält. Das wird nach der Ausführung als eine Historie neben dem Graph angezeigt, damit der Benutzer die Regelanwendung schrittweise zurücksetzen und dann wieder anwenden kann. (Abbildung 52.) Um die Regelanwendung zurückzusetzen und dann wieder anzuwenden, wählt der Benutzer eine Regel aus dem Historie mit einem Doppelklick darauf aus. Dabei wird diese Regel und die vor ihr stehenden angewendet und die nach ihr kommenden zurückgesetzt. Um alle Regelanwendungen zurückzusetzten, soll ein Doppelklick auf den Namen der Transformation Unit in der Historie gemacht werden.

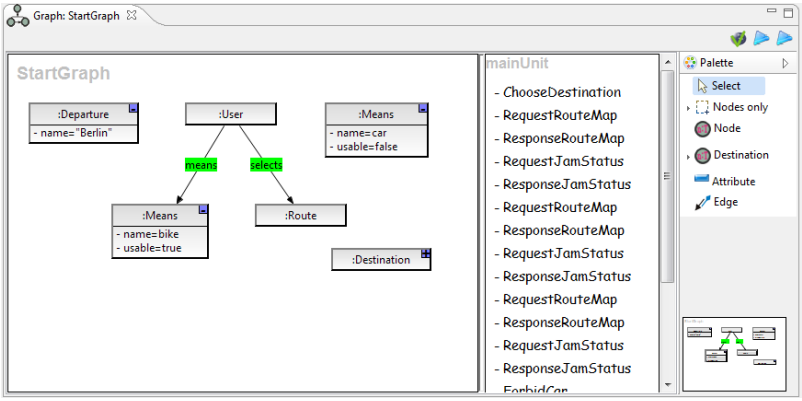

Abbildung 52. Graph View mit Historie von angewendeten Transformationsregeln

5. Verwandte Arbeiten

Es gibt mehrere Modeltransformationswerkzeugen, die EMF Modelle transformieren können. Beispielsweise ATL [8], EWL [9], Tefkat [8], VIATRA2 [10] und MOMENT [11].

Für ATL wurde vor kurzem eine formale Semantik basierend auf Maude vorgestellt [12]. Die im Maude definierte formale Semantik für „MOMENT“ und für ATL kann genutzt werden um die EMF Transformationsmodelle zu analysieren. Keine dieser Werkzeugumgebungen unterstützen visuelles Editieren von Kontrollstrukturen.

Graphen-Transformationswekrzeuge wie PROGRES [13], AGG [14], FuJaBa [15] und MoTMoT [16] enthalten visuelle Editoren die außerdem noch die Definition von Kontrollstrukturen (z.B. mit Hilfe von Diagrammen in FuJaBa die durch eine implizite Steuerung in [17] erweitert wurden) unterstützen. Das Tool Gr.Gen.NET [18] unterstützt ebenfalls beliebige Verschachtelungen der Anwendungsbedienungen. Jedoch basiert das Tool auf einer textuellen Spezifikationsprache.MoTMoT (Model driven, Template based, Model Transformer) ist ein Compilier der die visuellen Modeltransformationen in den Manipulationscode der Repositorys kompiliert. Der Kompilier nimmt die entsprechend dem UML Profil für „Story Driven Modeling“ genormte Modelle als Input und als Output wird das Java Metadata Interface (JMI) Code ausgegeben. Die Kontrollstrukturen werden durch die Aktivitätsdiagramme ausgedrückt. Da der MoTMot Codegenerator durch das Benutzen von AndroMDA entwickelt wurde, ist das Hinzufügen der Unterstützungen für weitere Repository

Platformen (wie EMF) prinzipiell möglich. Das kann durch das Hinzufügen von einer neuen Reihe der Codevorlagen ermöglicht werden.

Bisher unterstützt keine der soweit existierenden EMF Modeltransformationenansätze (basierend auf Graphentransformationen oder auf anderen Techniken) eine Konfluenz und Terminierungsanalyse der EMF Transformationsregeln, die wir mit dem Graphtransformationswerkzeug AGG durchführen können. Hierbei dient der HENSHIN Ansatz und die Werkzeugumgebung als eine Brücke um etablierte Werkzeugfeatures und formale Techniken für Graphentransformationen bereitzustellen für eine modelgetriebene Entwicklung mit EMF.

6. Zusammenfassung und Ausblick

6.1. Durchgeführte Erweiterungen und erbrachte Vorteile.

Mit den entwickelten Erweiterungen entspricht der Editor der Aufgabenstellung und ermöglicht eine automatisch gesteuerte Transformation durchzuführen. Es wird ermöglicht, mit einer Counted Unit eine Transformation Unit n-mal oder so lange wie möglich auszuführen, mit einer Independent Unit eine zufällige Transformation Unit anzuwenden, mit einer Conditional Unit die Verzweigung mit einer bedingten Anweisung zu definieren, mit einer Priority Unit die Ausführungsprioritäten zu setzen und mit einer Sequential Unit mehrere Transformation Units zu gruppieren. Außerdem steuert jede Transformation Unit die Parameterbelegung und die Ausführung von untergeordneten Units und macht damit die Transformation automatisch gesteuert. Weil die oberste Transformation Units in Baum-Ansicht extra gezeigt werden, kann man gleich sehen mit welcher Unit die Transformation ausgeführt werden soll. Oft ist das nur eine Transformation Unit pro Transformationssystem, was die Vorkenntnisse der Person die die Transformation ausführen soll auf die Angabe von Parameterwerten beschränkt. Das neue Henshin-Tool wurde 2010 auf dem Workshop „Graph-Based Tools“ vorgestellt [19].

6.2. Ausblick auf zukünftige Arbeiten

Als zukünftige Erweiterungen kommen Möglichkeiten für die Definition der Transformation Units in Betracht wo ein Teil der Transformationsregel beliebig oft ausgeführt werden kann. So wird eine Transformation Unit aus zwei Teilen bestehen. Ein Kernteil besteht aus einer Transformationsregel (Kernregel), das nur einmal ausgeführt wird und dabei der Arbeitsbereich des Graphs für den zweiten Teil bestimmen wird. Der zweite Teil (Multiteil) kann aus mehreren Transformationsregeln (Multiregeln) bestehen, auf die von der Kernregel eine totale Abbildung existieren soll. Jede Multiregel kann somit mehrmals ausgeführt werden, je nachdem, wie viele Matches es in den Graphen gibt. In der laufenden Diplomarbeit von A. Warning werden genau solche Units (sogenannte Amalgamation Units) implementiert.

Literaturverzeichnis

[1]. **Eclipse Consortium.** Eclipse Modeling Framework (EMF). [Online] [Zitat vom: 31. 08 2010.] http://www.eclipse.org/emf/.

[2]. **H.Ehrig, K.Ehrig, U.Prange. and G.Taentzer.** *Fundamentals of Algebraic Graph Transformation.* : Springer, 2006.

[3]. **Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, Gabriele Taentzer.** *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations Proc. of the ACM/IEEE 13th Inter. Conf. on Model Driven Engineering Languages and Systems (MoDELS´10).* 2010.

[4]. **Eclipse Consortium.** Eclipse Graphical Editing Framework (GEF). [Online] [Zitat vom: 31. 08 2010.] http://eclipse.org/gef.

[5]. **Tony Modica, Enrico Biermann, Claudia Ermel.** *An ECLIPSE Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models.* : GI-Workshop on Methodological Development of Modelling Tools, 2009.

[6]. **Enrico Biermann, Claudia Ermel, Gabriele Taentzer.** *Precise Semantik of EMF Modell Transformations by Graph Transformation Proc. ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS´08).* : Springer, 2008. 5301, 53-67.

[7]. **Enrico Biermann, Claudia Ermel, Stefan Jurack.** *Modeling the "Ecore to GenModel" Transformation with EMF Henshin.* : Proc. Transformation Tool Contest 2010 (TTC´10), 2010.

[8]. **F. Jouault, I. Kurtev.** Transforming Models with ATL. In MoDELS Satellite Events. Berlin : Springer, 2005, S. 128–138.

[9]. **D. Kolovos, R. Paige, F. Polack, L. Rose.** Update Transformations in the Small with Epsilon Wizard Language. *Journal of Object Technology 6(9):53–69.* [Online] 2007. http://www.jot.fm/issues/issues 2007 9/paper3.

[10]. *The model transformation language of the VIATRA2 framework.* **D. Varro, A. Balogh.** 2007, Science of Computer Programming 68(3).

[11]. **Boronat., A.** *MOMENT: A Formal Framework for Model Management. PhD thesis.* : Universitat Polit`ecnica de Val`encia, 2007.

[12]. **J. Troya, A. Vallecillo.** Towards a Rewriting Logic Semantics for ATL. In Proc. Int. Conf. on Model Transformation (ICMT'10). : Springer, 2010, S. 230–244.

[13]. **A. Schürr, A. Winter, A. Zündorf.** The PROGRES-Approach: Language and Environment. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools.* : World Scientific, 1999, S. 487 – 550.

[14]. **TFS-Group.** *TU Berlin. AGG.* [Online] 2009. http://tfs.cs.tu-berlin.de/agg..

[15]. **T. Fischer, J. Niere, L. Torunski, A. Zündorf.** Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. *In Proc. Int. Workshop on Theory and Application of Graph Transformation (TAGT).* Berlin : Springer, 2000, S. 296–309.

[16]. **FOTS-Group.** *University of Antwerp. MoTMoT: Model driven, Template based,Model Transformer.* [Online] 2010. http://www.fots.ua.ac.be/motmot/index.php.

[17]. *Towards a Hybrid Transformation Language: Implicit and Explicit Rule Scheduling in Story Diagrams.* **B. Meyers, P. Van Gorp.** 2008. In Proc. of the 6th Int. Fujaba Days.

[18]. **R. Geiß, M. Kroll.** GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool.. *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE'07).*: Springer, 2008.

[19]. **Enrico Biermann, Claudia Ermel, Johann Schmidt, Angeline Warning.** *Visual Modeling of Controlled EMF Model Transformation using HENSHIN.* : Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010), 2010.

[20]. **Philipp Schill, Ralf Schmauder.** sigs-datacom. *Codegenerierung mit dem "Eclipse Modeling Framework" und Jet.* [Online] 01 2005. [Zitat vom: 02. 08 2010.] http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2005/01/schmauder_schill_OS_01_05.pdf.

[21]. **TFS TU Berlin.** Visuelle-Sprachen-Projekt Folien WS09. [Online] 2009. [Zitat vom: 11. 08 2010.] http://user.cs.tu-berlin.de/~vila/www_ws09/folien/.

[22]. **Enrico Biermann, Karsten Ehrig, Claudia Ermel, Christian Köhler, Gabriele Taentzer.** The EMF Model Transformation Framework. *Tiger EMF Transformation Project.* [Online] 2007. [Zitat vom: 11. 08 2010.] http://tfs.cs.tu-berlin.de/publikationen/Papers08/EBT+08.pdf.

[23]. **M. Lawley, J. Steel.** Practical Declarative Model Transformation with Tefkat. In MoDELS Satellite Events. Berlin : Springer, 2005, S. 139–150.