Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Fakultät für Informatik
Lehrstuhl für Programmiersysteme

Graphersetzungssysteme als Werkzeuge für UML Modelltransformationen

Studienarbeit am
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Programmiersysteme
Prof. Dr. Walter F. Tichy
Fakultät für Informatik
Universität Karlsruhe (TH)

von

Bugra Derre

Betreuer:
Prof. Dr. Walter F. Tichy
Dipl.-Inform. Tom Gelhausen

Tag der Anmeldung: 24. April 2008
Tag der Abgabe: 24. Juli 2008

Hiermit erkläre ich, dass ich diese Studienarbeit selbstständig verfasst habe. Alle nicht von mir stammenden Inhalte sind durch Angabe der Quelle kenntlich gemacht.

Karlsruhe, den 18. Juli 2008

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Zusammenfassung

Graphen bieten sich als anschaulicher, mathematisch präziser und ausdrucksstarker Formalismus zur Modellierung von in Beziehung stehenden Objekte an. Mit Graphersetzungssystememem stehen uns mächtige Werkzeuge zur Manipulation und Transformation von Graphen zur Verfügung. Erweiterungen der Graphersetzungssysteme machen die Transformation und Manipulation von UML-Modellen möglich. Der vorgestellte Ansatz ist einfach und effektiv: er bietet eine komplette Abdeckung der UML2, die *alle* Diagrammtypen unterstützt. Der portierbare Ansatz ist für typisierte Graphersetzungssysteme mit textueller Eingabe anwendbar. Konkrete Codebeispiele demonstrieren und veranschaulichen den hier vorgestellten Ansatz.

2. Einleitung

Graphersetzungssysteme[1] bieten sich für mächtige und dennoch einfach handhabbare Operationen an komplexen Datenstrukturen an. Jüngste Untersuchungen zeigten eine beachtliche Geschwindigkeit in der Ausführung von Graphersetzungen [SNZ08].

Graphersetzungssysteme benötigen für den Einsatz in einer bestimmten Domäne eine Anpassung auf funktionaler Ebene, d.h. eine geeignete Modelldefinition. Diese Ausarbeitung zeigt eine einfache Vorgehensweise um automatisch generierte Modelldefinitionen für die Modellstrukturen der UML2 für Graphersetzungssysteme zu erhalten. Zusätzlich werden passende Import- und Exportfilter für UML-Diagramme präsentiert, die im XMI-Format vorliegen [OMG07b]. In dem weiterführenden Beispiel wird die Vorgehensweise an dem Graphersetzungssystem GrGen.NET veranschaulicht.

In zahlreichen Veröffentlichungen wird erwähnt, dass die Ausführungsgeschwindigkeit von Modelltransformationen ausschlaggebend ist, um einen integrierten Entwicklungsvorgang zu unterstützen [Chr04, CHM+02]. GrGen.NET ist eines der zur Zeit schnellsten Graphersetzungssysteme [GBG+06, SNZ08]. Außerdem ist seine Ausdrucksfähigkeit hinsichtlich der Mustersuche und den Möglichkeiten der Graphersetzung mindestens ebenwürdig zu anderen Graphersetzungssystemen und übertrifft die von QVT [OMG07c].

QVT ist der zur Zeit wohl prominenteste Ansatz der Object Management Group (OMG) für Modell-zu-Modell-Transformation, trotzdem ist keine vollständige Implementierung dieses Ansatzes bekannt. Neue Funktionen wie dynamische Muster werden in der nächsten Version GrGen.NET 2.0 diesen Vorsprung weiter ausbauen. Betrachtet man all diese Aspekte, bietet GrGen.NET eine exzellente Grundlage für Modelltransformationen.

Diese Ausarbeitung soll a) Nutzer des Systems über die Modelldefinition aufklären, wie z.B. die Knoten- und Kantenklassen aufgebaut werden, und b) Autoren anderer Graphersetzungssysteme in die Lage versetzen, selbst UML-konforme Modelldefinitionen zu generieren. Der Standard GXL [HWS00] in Graphersetzungssystem bietet sich an, um UML-Graphen untereinander auszutauschen ohne dafür Import- und Exportfilter für jedes Graphersetzungssystem zu implementieren. Dies macht den Weg für einen direkten Vergleich der Graphersetzungssysteme hinsichtlich brauchbarer Modelltransformationen frei.

[1] Im Folgenden betrachten wir nur universell einsetzbare Graphersetzungssysteme. Sie sind allgemeingültig und nicht an eine spezifische Domäne gebunden.

2.1. Systemüberblick

Der hier vorgestellte Ansatz ist klar und transparent: die UML-Spezifikation liegt als Quelldokument im bekannten XMI-Format vor. XSL Transformationsskripte bilden das Quelldokument automatisch in eine Modelldefinition für GrGen.NET ab.

Nach dieser Vorbereitung für GrGen.NET modellieren wir mit einem UML-Werkzeug[2] ein Diagramm. Abbildung 2.1 demonstriert den Systemverlauf und zeigt die einzelnen Schritte. Mit dem UML-Werkzeug kann das modellierte Diagramm 1) in einem XMI-Dokument repräsentiert und gespeichert werden. Das XMI-Diagramm wird dann 2) mit dem Importfilter in GrGen.NET als Graphinstanz geladen um 3) Transformationen daran auszuführen. Nach erfolgreicher Transformation des Graphen wird der Graph 4) wieder in ein XMI-Dokument zurück exportiert, um es wieder 5) in einem UML-Werkzeug öffnen und weiter bearbeiten zu können.

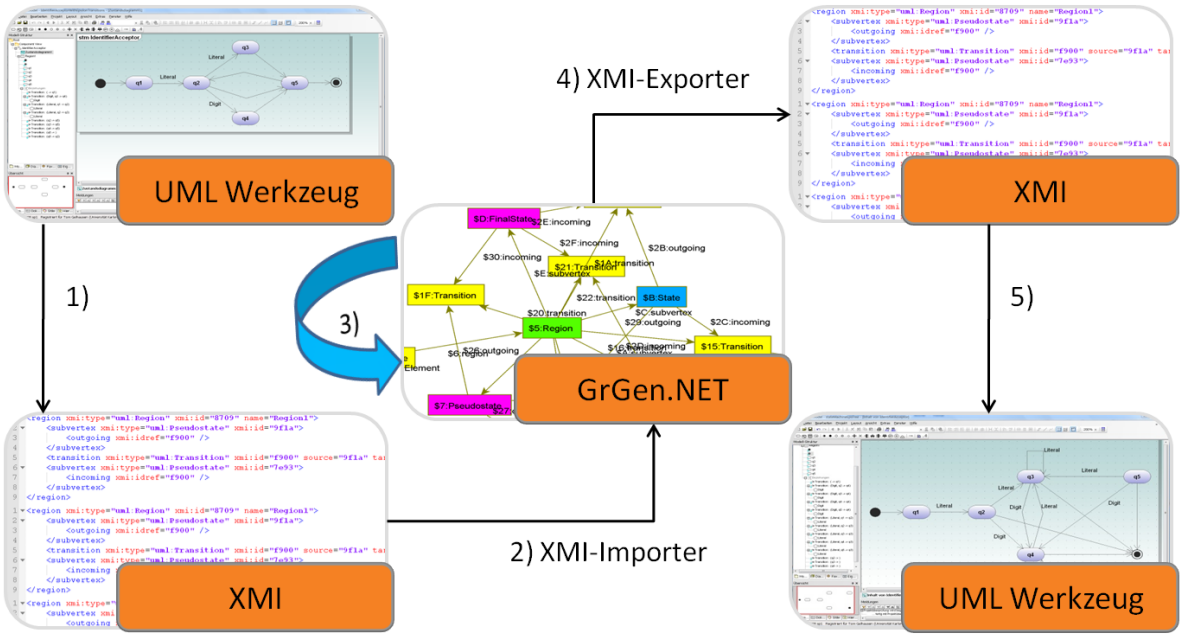

Abbildung 2.1.: Überblick über das System und den Prozessverlauf

2.2. Alternative Verfahren

Graphersetzungssysteme bieten gegenüber anderen alternativen Verfahren entscheidende Vorteile. Mit der deklarativen Programmiersprache können die zu suchenden Struktu-

[2] Das UML-Werkzeug muss den Export des UML-Diagramms in ein XMI-Dokument unterstützen. Eine Übersicht von UML-Werkzeugen und deren XMI-Unterstützung gibt es unter: http://www.jeckle.de/umltools.html und http://www.oose.de/umltools.htm

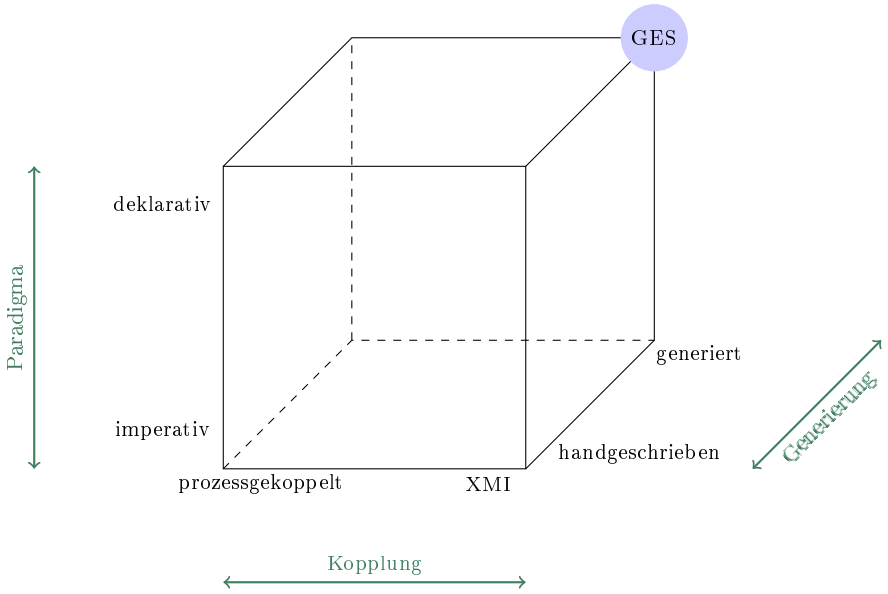

Abbildung 2.2.: Alternativen zu Graphersetzungssystemen (GES) im Überblick

ren und die Transformationsregeln in übersichtlichen Graphmustern angegeben werden. Damit ist eine intuitivere Handhabung von Graphen möglich als mit imperativen Programmiersprachen.

Starke Kopplung an einen systemgebundenen Prozess wird bei dem vorgestellten Ansatz ebenfalls vermieden. Im Gegensatz zu einem im UML-Werkzeug integrierten Plug-In ist dieser Ansatz kaum an das benutzende System gekoppelt. Die einzige Kopplung besteht zwischen dem XMI-Dokument und dem Graphersetzer und kann daher uneingeschränkt für XMI-basierte Dokumente verwendet werden.

Betrachtet man den Vergleich zwischen handgeschriebenem und automatisch generiertem Code, wirkt sich die Wahl dieses Ansatzes auch positiv aus. Die Im- und Exportfilter generieren die Graphinstanziierungsanweisungen vollautomatisch und werden nicht manuell bearbeitet. Modelltransformationen, basierend auf diesem Ansatz, führen zu einer enormen Codereduzierung, mit einer gleichzeitig intuitiv bedienbaren Programmiersprache.

2.3. Metamodellebenen

Die OMG geht bei der Metamodellierung von vier Metaebenen aus [OMG06a, OMG07a]. Jede Metaebene wird durch eine darüberliegende Metaebene spezifiziert, wobei die oberste Metaebene (M3) eine Außnahme ist. M3 ist in sich selbst definiert und beschreibt sich selber, daher wird über M3 keine weitere Metaebene angeordnet. Die Ebene M3 wird gerne als Fixpunkt in der Metamodellhierarchie bezeichnet.

Eine Metaebene verhält sich – genau wie in der Mathematik – wie ein Universum zu seinem Modell: das Universum U enthält eine Menge von Dingen, die im Modell M spezifiziert wurden. Das Modell M selbst wiederum ist ein Universum U' des Modells M'. Aus der Sicht des Universums U ist M das Modell und M' das Metamodell. Die vorgestellten Metaebenen werden, um die Konventionen der OMG einzuhalten, mit M0 bis M3 bezeichnet (s. Abb. 2.3).

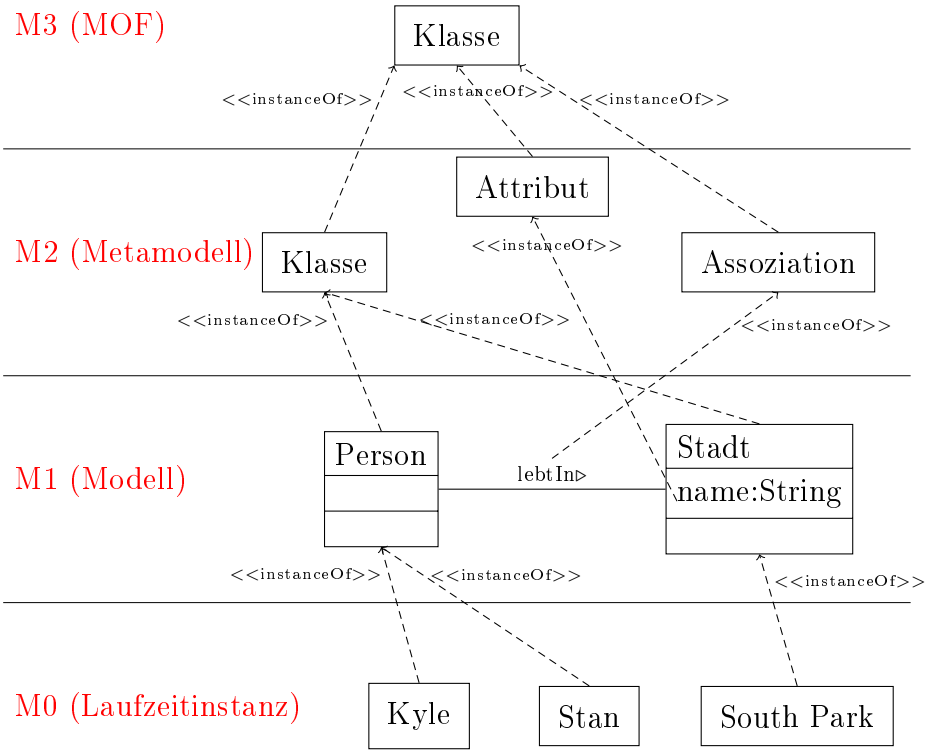

Abbildung 2.3.: Beispiel der vier Metaebenen

2.3.1. Beispiel

Abbildung 2.3 illustriert die verschiedenen Ebenen und ihre Beziehungen zueinander. Im Folgenden wird die Abbildung ausführlich beschrieben und erklärt.

Ebene M0 sei ein Universum: die in M0 enthaltenen Elemente seien `Kyle` und `Stan`, die in `South Park` leben.

Die Ebene M1 legt die Klassifizierungen für M0 fest. Sie gibt an, welche Elemente in M0 instanziiert werden können. Beispielhaft existiert eine Klasse `Person` und eine Klasse `Stadt` die durch eine Assoziation `lebtIn` verbunden sind. Außerdem enthält die Klasse `Stadt` ein Attribut `name`. `Kyle` und `Stan` sind Instanzen der Klasse `Person` und `South Park` ist eine Instanz der Klasse `Stadt`. M1 ist das Modell für das Universum M0.

Ebene M2 legt die Klassifizierungen für M1 fest: M1 darf Instanzen des Typs `Klasse`, `Attribut` und `Assoziation` anlegen. `Person` und `Stadt` sind Instanzen von `Klasse`, `lebtIn` eine Instanz vom Typ `Assoziation` sowie `name` eine Instanz von `Attribut`. M2 bildet das Metamodell.

M2 selbst darf lediglich Instanzen des Typs `Klasse` (M3) anlegen. `Klasse`, `Attribut` und `Assoziation` sind Instanzen von `Klasse`. M3 definiert das Metametamodell. Da M3 sich selber beschreibt wird keine weitere Metaebene definiert und stellt so einen Fixpunkt in der Metamodellhierarchie dar.

2.4. Metamodelle

Mit der MOF können verschiedene Metamodelle spezifiziert werden. Der bekannteste Vertreter ist wohl die UML, aber auch das Common Warehouse Metamodel (CWM) oder die Systems Modeling Language (SysML) [OMG03, OMG07d], sind mit MOF spezifizierte Metamodelle. Abbildung 2.4 verdeutlicht die Spezifikation verschiedener Metamodelle durch MOF.

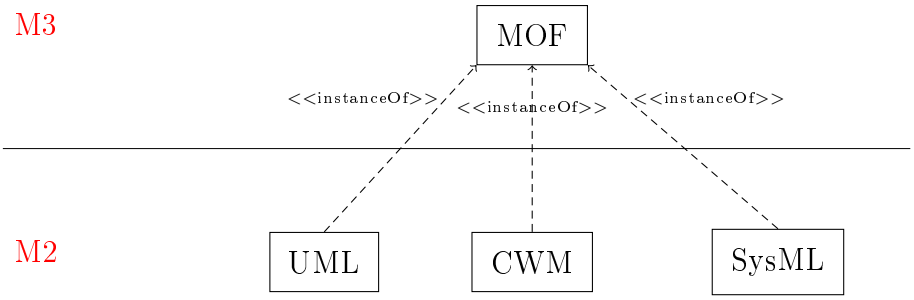

Abbildung 2.4.: UML und MOF auf verschiedenen Metaebenen

CWM bezeichnet das Common Warehouse Metamodel, dass in den Prozessen der Data Warehouse und Data Mining Verwendung findet. SysML (Systems Modeling Language) modelliert visuelle, graphische Informationen.

2.5. Abbilden der Metaebenen

Geplante Software-Projekte werden mit Hilfe von UML-Diagrammen modelliert. Die Modellierung erfolgt somit auf der M1-Ebene, siehe Kapitel 2.3. Für das Graphersetzungssystem sind die UML-Diagramme die Daten auf denen es effektiv operieren soll. Aus der Sicht des Graphersetzungssystems sind es Daten der Laufzeitinstanz auf M0 ④. Dementsprechend benötigen wir Transformationsskripte um den Inhalt der UML-Diagramme ② in Kanten und Knoten erschaffende Anweisungen für GrGen.NET abzubilden ④. Logischerweise – und auch aus formellen Gründen – benötigt man für streng typisierte Graphersetzungssysteme, wie GrGen.NET, eine Modelldefinition für Knoten und Kanten.

Diese Modelldefinition liegt aus der Sicht des Graphersetzungssystems auf M1. Wir erhalten die Modelldefinition durch Abbilden von M2 auf dem "Metamodellstapel" ① des UML-Metamodells auf die Ebene M1 des Metamodellstapel's von GrGen.NET ③, s. Abb. 2.5.

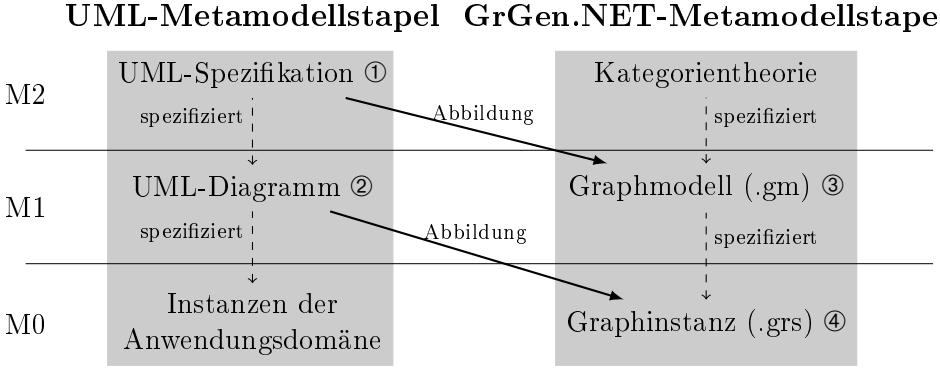

Abbildung 2.5.: Illustration der Abbildungen

3. Beispiel

Um einen Eindruck über den entstehenden Graphen zu gewinnen, wollen wir eine beispielhafte Modelltransformation betrachten. Wir modellieren mit einem UML-Werkzeug einen Zustandsautomaten um anschließend die ϵ–Übergänge zu entfernen. Dazu importieren wir die XMI-Darstellung des Zustandsautomaten in GrGen.NET. Nach der Ausführung einiger Graphersetzungsregeln erhalten wir einen ϵ–freien Zustandsautomaten. Im letzten Schritt wird der fertige in GrGen.NET als Graph vorliegende Zustandsautomat wieder in das XMI-Format exportiert (vgl. Abbildung 2.1).

Im Beispielcode, siehe Auszug 3.1, ist eine Graphersetzungsregel formuliert. Sie ist ein Auszug aus dem Regelsatz die einen gegebenen Zustandsautomaten in einen ϵ–freien Zustandsautomaten überführt. Der komplette Regelsatz wird im Anhang A.1 vorgestellt. Im Folgenden wollen wir uns jedoch auf den hier vorgestellten Teil der Regel konzentieren und uns mit dem wesentlichen Teil des Regelwerks vertraut machen.

Die Regel sucht nach einem vorhandenen Graphmuster mit einem ϵ–Übergang und entfernt diesen anschließend. Der Regel wird der Zustandsautomat sm als Parameter übergeben (Zeile 2). In diesem Automaten wird ein ϵ–Übergang gesucht. Nach der Definition der zu suchenden Struktur (Zeile 3 bis 6), auch Suchmuster genannt, folgen zwei negative Regelbedingungen[1].

Die erste negative Regelbedingung in den Zeilen 9 bis 12 besagt, dass ein gefundener Zustandsübergang *nicht* von einem Startzustand ausgehen soll. Die zweite negativen Regelbedingung in Zeile 16 fordert, dass eine Kante keinen Auslöser (trigger) haben darf, also ein ϵ–Übergang ist. Der Punktoperator in dieser negativen Regelbedingung stellt einen Knoten beliebigen Typs dar. Die modify-Anweisung in Zeile 19 ist der graphmanipulierende bzw. -transformierende Teil der Regel. Die entsprechenden Namen der Knoten- und Kantenklassen sind nicht zufällig gewählt, sondern sind genau die in der UML-Spezifikation definierten Typen [OMG07e]. Die – für diese Regel – benutzten Typen sind graphisch in Abbildung 4.1 und textuell in Abbildung 4.2 abgebildet und koloriert dargestellt. Sie entfernt die gefundene ϵ–Kante.

Die Regel removeEpsilonTransition ist nicht die einzige Regel zur Befreiung der ϵ–Übergänge. Sie baut auf weiteren, vorrangegangenen Transformationsregeln auf. Da die Implementierung des Regelsatzes für den weiteren Verlauf nicht von größerer Bedeutung ist, wird für interessierte Leser auf Anhang A.1 verwiesen. Dort sind alle Regeln aufgelistet und ausführlich dokumentiert.

[1] Eine negative Regelbedingung verhindert die Ausführung der Regel, falls die negative Regelbedingung zutrifft.

```
1  /* Removes an epsilon transition in the given state machine sm. */
2  rule removeEpsilonTransition(sm:StateMachine) {
3     sm -:region-> reg:Region;
4     reg -:transition-> xy:Transition;
5     // Linked state must not be a final state.
6     xy -:target-> :State\(FinalState);
7
8     // The transition xy must not leave an initial pseudostate.
9     negative {
10       reg -:subvertex-> x:Pseudostate;
11       x <-:source- xy;
12    }
13
14    // Transition xy is not allowed to have a trigger.
15    // Else it would not be an epsilon transition.
16    negative { xy -:trigger-> .; }
17
18    // Delete the epsilon transition.
19    modify { delete(xy); }
20 }
```

Auszug 3.1: Ersetzungsregel für den ϵ–freien Zustandsautomaten

Abbildung 3.1 zeigt die Modellierung eines Zustandsautomaten mit dem UML-Werkzeug Altova UModel 2008 [Alt08]. Der modellierte Automat akzeptiert das Eingabealphabet $\sum$={*Digit, Literal*}, wobei Digit für eine Ziffer von 0 bis 9 und Literal für einen beliebigen Buchstaben des Alphabets steht. In dem Teilfenster „Modellstruktur“ sind ϵ–Transitionen ohne Kindelemente dargestellt. In dem visuellen Diagramm enthalten die Kanten dementsprechende Kantenbeschriftungen.
Der Automat wurde in ein XMI-Dokument umgewandelt und anschließend in GrGen.NET importiert, um dort die Transformationsschritte zur ϵ–Befreiung durchzuführen. Nach der Modelltransformation wird es in das XMI-Dokument exportiert um es dann wieder mit Altova UModel 2008 zu laden. Die visuelle Anordnung und Größe des Diagramms (Größe der Zustände, Länge der Kanten, usw.) geht bei der Speicherung verloren, da XMI kein einheitliches Format für diese Art von visueller Information bietet. Abb. 3.2 zeigt das Ergebnis der Modelltransformation.

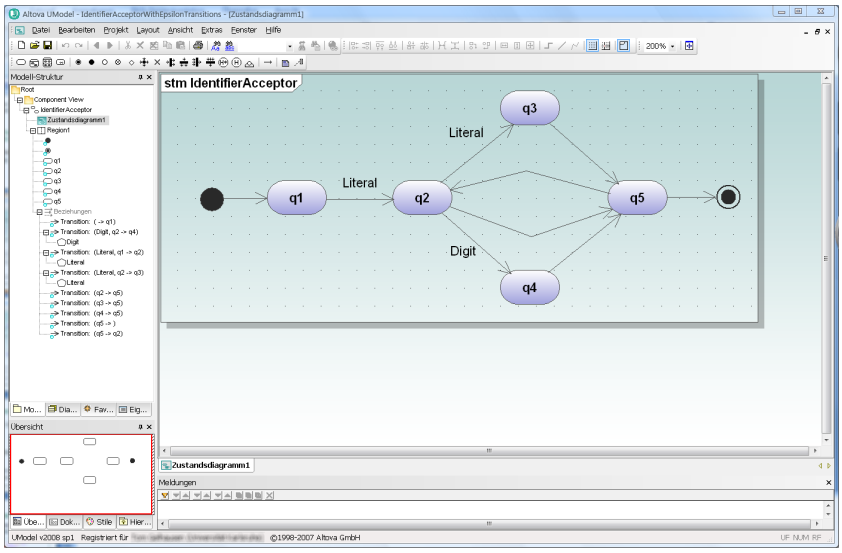

Abbildung 3.1.: Der Zustandsautomat mit ϵ–Übergängen ...

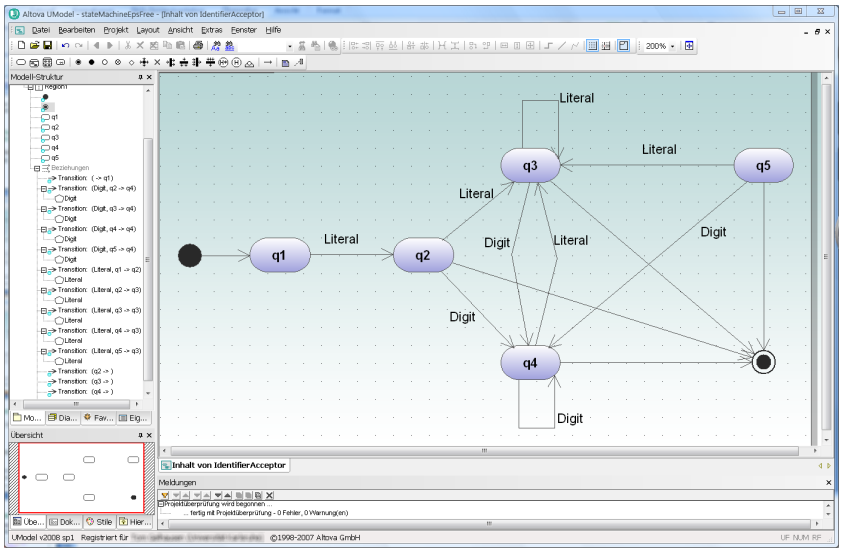

Abbildung 3.2.: ... wird zu einem ϵ-freien Zustandsautomat.

4. Implementierung

Die Aufgabe in der vorgestellten Arbeit ist die Abbildung der UML-Spezifikation in eine GrGen.NET Modelldefinition und des UML-Modells in eine Graphinstanz. Bei der Abbildung ist zu beachten, dass sich der Formalismus im Quellmodell und Zielmodell unterscheiden. UML2 ist durch die MOF beschrieben, GrGen.NET durch eine andere sprachliche Syntax. Es handelt sich um zwei verschiedene Formalismen, die jeweils in den anderen Formalismus überführt und abgebildet werden müssen. Man spricht auch von einer exogenen Abbildung. Die Anpassung des Quellmodells an die des Zielmodells wird in den Kapiteln 4.2 und 4.3 beschrieben. Zusätzlich wird der Exportfilter für GrGen.NET zum Exportieren des UML-Graphen in ein XMI-Dokument näher beschrieben.

4.1. Compliance Level

Die UML ist unterteilt in vier Normierungsebenen (compliance level) welche jeweils einen bestimmten Teil der UML unterstützen. Die Aufteilung der komplexen UML-Struktur dient zur Übersichtlichkeit und Vereinfachung beim Austausch von UML-Modellen. Jede Ebene (L0, ..., L3) stellt eine Teilmenge der UML-Struktur zur Verfügung und verzichtet dabei auf wechselseitige Importanweisungen verschiedener Pakete und Elemente.

- *Level 0 (L0).* Beinhaltet Elemente zur Modellierung von klassenbasierten Strukturen, beispielsweise die Modellierung von Klassendiagrammen. L0 stellt einen kleinsten gemeinsamen Nenner dar und bietet sich als Modellgrundlage für den Austausch zwischen verschiedenen Modellierungswerkzeugen an.
- *Level 1 (L1).* Erweitert L0 um neue sprachliche Elemente für Anwendungsfälle (use cases), Interaktionen (interactions), Strukturen (structures), Aktionen (actions) und Aktivitäten (activities).
- *Level 2 (L2).* Erweitert L1 um sprachliche Mittel für Verteilungsdiagramme (deployment diagrams), Zustandsautomaten (state machines) und Profile (profiles).
- *Level 3 (L3).* L3 repräsentiert die gesamte UML. Es erweitert L2 um die sprachlichen Mittel für Informationsabläufe (information flows), zur Modellierung von Vorlagen (templates) und Modellkapselung (model packaging).

Wir entschieden uns für "L3.merged.cmof" als Quelldatei [OMG06c] für die Modelldefinition von GrGen.NET, da L3 die gesamte UML wiedergibt (siehe [OMG07e]).

4.2. Abbildungsformalismus

Modellabbildungen von einem Quellformalismus in einen Zielformalismus sind eine der Kernaufgaben in der modellgetriebenen Architektur [CS03]. Sei *m(s)/f* das Modell m im System s mit dem Formalismus f.

Eine Abbildung ist eine Transformation eines Quellmodells $m_1(s)/f_1$ in ein Zielmodell $m_2(s)/f_2$. Da es sich bei einer Abbildung (für gewöhnlich) um dasselbe System s handelt, schreiben wir verkürzt $m_1/f_1 \rightarrow m_2/f_2$.

Ist der Zielformalismus der Gleiche wie der Quellformalismus, dann handelt es sich um eine endogene Abbildung. Die Abbildung $m_1/UML2 \rightarrow m_2/UML2$ ist eine endogene Abbildung, da der Formalismus der UML-Syntax in beiden Modellen derselbe ist. Bei der exogenen Abbildung $m_1/UML2 \rightarrow m_2/GrGen.NET$ müssen Ausdrücke des Formalismus f_1 in zulässige Ausdrücke von f_2 übersetzt werden. Oft fehlen die entsprechenden Ausdrücke in dem Zielformalismus oder haben eine andere Semantik als in dem Quellformalismus. Zwei Probleme gilt es zu klären:

1. Kann man mit einem Formalismus das Quellmodell exakt abbilden?
2. Welchen Einfluss nimmt der Formalismus auf die Qualität des Modells?

Eine Sprache ist ein Werkzeug um eine Theorie (eine Idee, einen Standpunkt, eine Meinung) über eine bestimmte Domäne zu formulieren. Gleichzeitig aber wirkt die Sprache wie ein Filter während der Wahrnehmung einer, möglicherweise unbekannten, Domänensprache. Nicht alle Formulierungen werden so wahrgenommen wie sie von der Quelle wiedergegeben wurden. Folgendes Zitat liefert eine mögliche Antwort.

> *„A language is both a means to describe a world and a means to understand it. Insofar as the primitives of a language are used to interpret and to understand a world, this world appears to be structured, and by a boomerang effect, these primitives are present — eventually with slight semantic variations — within all modeling formalisms.“* [CS03]

Für zwei verschiedene Formalismen f_1 und f_2 gilt also, dass sie durch – in das Modell – eingebaute Semantiken und durch Interpretation der Semantiken aneinander angepasst werden können. Daran entscheidet sich auch die Qualität des abzubildenden Modells.

4.3. Modelldefinition

OMG beschreibt die UML2 [OMG07e] durch die CMOF („Complete Meta Object Facility“). Das CMOF-Metamodell befindet sich auf der Meta-Metaebene M3. Allerdings sind nicht alle Elemente des Metamodells notwendig, um die UML2 zu formulieren. Für die Abbildung ist eine Untermenge ausreichend. Die verwendeten und in GrGen.NET abzubildenden Elemente sind in Tabelle 4.1 aufgelistet.

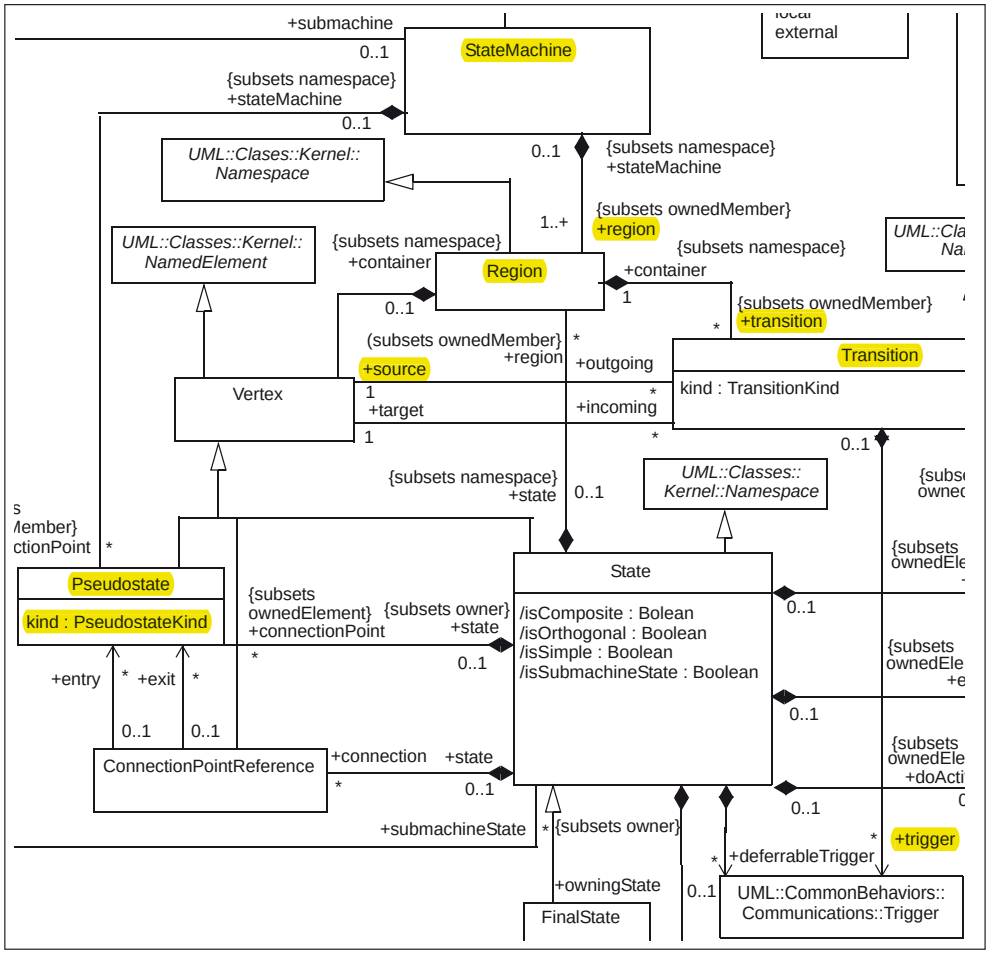

Abbildung 4.1.: Auszug aus: "Figure 15.2 – State Machines" der UML-Aufbauspezifikation

4.3.1. Abbildungen

CMOFClass-Elemente werden zu GrGen.NET-Knotenklassen abgebildet. Der Name der Knotenklasse ergibt sich aus dem Namen des entsprechenden UML-Elements. Sie entsprechen genau den Bezeichnungen die auch in der (ablesbaren) UML-Aufbauspezifikation verwendet werden. Da die UML-Elemente in allen Normierungsebenen der OMG eindeutige Namen besitzen, ist auch die Namensgebung für CMOFClassen in der Modelldefinition eindeutig, siehe Abbildungen 4.1 und 4.2. Dies führt zu einer eleganten Namensgebung für Knotenklassen.

Ein CMOFProperty definiert einen Namen für einen Wert im Zusammenhang mit einer CMOFClass. Es ist vergleichbar mit einem Attribut in einer objektorientierten Programmiersprache. Der Wert wird üblicherweise durch ein anderes CMOFClass-Element verkörpert, welches den aktuellen Attributwert darstellt. Ein CMOFProperty kann jedoch

15.3.10 Region (from BehaviorStateMachines)

Generalizations

- "Namespace (from Kernel)" on page 101
- "RedefinableElement (from Kernel)" on page 132

Description
A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.

Attributes
No additional attributes

Associations
[...]

- transition:Transition[*] The set of transitions owned by the region. Note that internal transitions are owned by a region, but applies to the source state. {Subsets *Namespace::ownedMember*}
- subvertex: Vertex[*] The set of vertices that are owned by this region. {Subsets *Namespace::ownedMember*}

Abbildung 4.2.: Auszug aus: "Figure 15.2 – State Machines" der UML-Aufbauspezifikation bzgl. Region

optional sein oder auch mehrfach in einer vorhandenen Instanz vorkommen. Daher ist es nicht möglich, ein CMOFProperty als ein GrGen.NET Klassenattribut darzustellen. Eine CMOFProperty wird deswegen als Kantenklasse dargestellt, die ausgehend von ihrem Besitzertyp ihren zugehörigen Werttyp referenziert.

Unglücklicherweise ist die Namensgebung für ein CMOFProperty nicht eindeutig, so wie es bei CMOFClassen der Fall ist. Da mehrfache Kantenklassen mit selben Namen in GrGen.NET nicht zulässig sind erfordert dies die Aufsummierung der mehrfach vorkommenden Verbindungsbedingungen der jeweiligen Kantenklassen. Auf diese Art kann man den einfachen Namen der CMOFProperty verwenden.

Jedes CMOFPrimitiveType-Element wird durch eine Knotenklasse repräsentiert. Diese Knotenklassen erben einen gemeinsamen, abstrakten Supertyp welches ein abstraktes Attribut value deklariert: es wird ein Attribut ohne Angabe eines speziellen Typs deklariert. Jedes konkrete CMOFPrimitiveType initialisiert das Attribut mit einem entsprechenden Typ von GrGen.NET's verfügbaren primitiven Typen. CMOFEnumerationen werden zu enum's und CMOFEnumerationLiterale zu entsprechenden enum::item's abgebildet. CMOFEnumerationen werden wie CMOFPrimitiveType-Elemente behandelt

CMOF	GrGen.NET (③, *.gm-Datei)
CMOFAssociation	–
CMOFClass	`node class`
CMOFComment	*// Kommentar*
CMOFConstraint	–
CMOFEnumeration	`enum`
CMOFEnumerationLiteral	`enum::Literal`
CMOFOpaqueExpression	–
CMOFOperation	–
CMOFParameter	–
CMOFPrimitiveType	verpackt in `node class`
CMOFProperty	`edge class`

Tabelle 4.1.: Verwendete CMOF-Strukturen in UML2 und entsprechende Abbildung in GrGen.NET

und bearbeitet. CMOFCommentare werden zu Kommentaren in der Modellsyntax von GrGen.NET abgebildet.

Im XMI-Quelldokument der UML-Normierungsebenen geben alle UML2 Modellelemente ihre Instanzbeziehung zu der darüber liegenden Metaebene an. Über das zusätzliches XMI-Attribut xmi:type wird bekannt gegeben, von welchem Metaelement es instanziiert wurde. In GrGen.NET haben wir keine äquivalenten Ausdruck in der Syntax um diese Instanzbeziehung auszudrücken. Wir konstruieren uns daher zusätzliche Knotensuperklassen und -kanten, mit denen wir unsere Instanzbeziehungen formulieren wollen. Jedes Element der Modelldefinition in GrGen.NET erbt von einer entsprechenden Superklasse. Eine Enumklasse beispielsweise erbt von einer CMOFEnumeration-Knotenklasse. Auf diese Weise können die Instanzbeziehung der verschiedenen UML-Elemente abgelesen werden. Weiter erbt eine Klasse von CMOFClass, die Attributkanten von CMOFProperty und primitive Typen von CMOFPrimitiveType. Diese Formulierung ist, wie in Kapitel 4.2 bereits besprochen, zulässig und gültig.

Aktuell wird dynamisches Verhalten in UML (und/oder anderen Sachen die mit CMOF spezifiziert sind) in unserem Ansatz nicht unterstützt. CMOFConstraint-, CMOFOpaqueExpression-, CMOFOperation- sowie CMOFOperation-Elemente werden nicht auf das Modell abgebildet. Dennoch sind all diese Elemente über einen optionalen Reflexionsmechanismus erreichbar[1]. So scheint eine Unterstützung für in CMOFConstraint und CMOFOpaqueExpression vorkommende OCL Ausdrücke machbar zu sein. Möglich machen würde das ein auf Graphersetzung basierender Interpretierer. Dasselbe gilt natürlich auch für CMOFOperationen und deren CMOFParameter.

Um CMOFAssociationen zu unterstützen, bräuchte man eine Ausdrucksstärke wie sie in dem Ersetzungssystem für Omnigraphen (Ogre) [DGG08] geboten wird: CMOFAssociati-

[1] Der Reflexionsmechanismus wird standardmäßig nicht verwendet.

onen sind binäre Beziehungen zwischen CMOFPropertyElementen (siehe Abbildung 4.3). Assoziationen würden durch Omnikanten repräsentiert werden. Diese wiederum würden an anderen, CMOFProperty darstellenden Omnikanten, anliegen. CMOFAssociation–Elemente beschreiben, anders als man es erwarten würde, keine weiteren Aktionen wenn sie eingefügt oder entfernt werden. Die Komplexität würde sich durch den Einsatz des Ogre Werkzeuges erhöhen. Daher wird auf dessen Verwendung und eine Unterstützung für CMOFAssociationen verzichtet. Ähnlich verhält es sich auch mit CMOFOperationen. Da die Operationen (bzw. Methoden) lediglich auf der Metaebene und nicht in ihrer darunterliegenden Instanz operieren, werden sie nicht in GrGen.NET's Modelldefinition abgebildet.

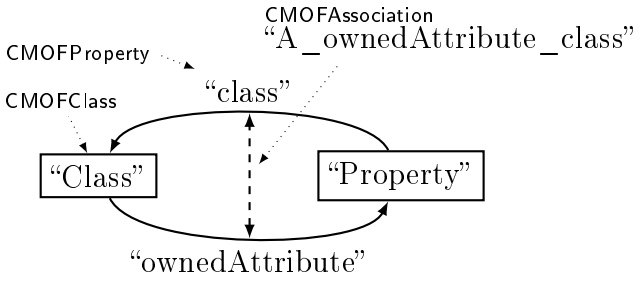

Abbildung 4.3.: Die Rolle der CMOFAssociationen

Das bedeutet aber nicht, dass man nicht mit UML-Assoziationen umgehen kann. Im Gegenteil, Assoziationen sind in UML-Modellen Instanzen der CMOFClass mit dem Namen „Association“. CMOFAssociationen sind im Gegenzug Assoziationsgebilde auf einer abstrakteren (und damit höheren) Metaebene. Hiermit wird nocheinmal betont, dass die unterstützten CMOF-Modellelemente ausreichen um alle derzeitigen UML-Diagrammtypen und deren Elemente zu formulieren.

4.3.2. Anwendungsbeispiel

Codebeispiel 4.1 zeigt einen Teil der von der UML2 in GrGen.NET abgebildeten Modelldefinition. Der gezeigte Teil beinhaltet die Elemente zur Modellierung eines Zustandsautomaten, siehe Abbildung 4.1 und 4.2. Die ersten beiden Zeilen kommentieren die Knotenklasse Region. Der Kommentar wird direkt aus dem Quelldokument für die Modelldefinition (L3) in die Syntax von GrGen.NET eingefügt. Die Knotenklasse Region erbt von den beiden Superknotenklassen RedefinableElement und Namespace (Zeile 3). Zusätzlich enthält die Region ein String-Attribut uuid (Zeile 4), welches für jede Instanz dieser Klasse mit dem Namen der Knotenklasse initialisiert wird: So kann für jede Knotenklasse die gespeicherten reflexive Informationen bestimmt werden.

Die beiden Kantenklassen subvertex (Zeile 7 bis 10) und transition (Zeile 12 bis 15) einer Region geben ihre Beziehungen zu anderen Klassen über Verbindungsbedingungen

```
1  // A region is an orthogonal part of either a composite state
2  // or a state machine. It contains states and transitions.
3  node class Region extends RedefinableElement, Namespace {
4     uuid = "Region";
5  }
6
7  edge class subvertex extends CMOFProperty
8     connect Region[0:*] -> Vertex[*] {
9        name = "subvertex";
10 }
11
12 edge class transition extends CMOFProperty
13    connect Region[0:*] -> Transition[*] {
14       name = "transition";
15 }
```

Auszug 4.1: Ein Auszug der automatisch generierten Modelldefinition

und Bereichsangaben an, siehe Zeile 8 bzw. 13. Eine subvertex-Kante kann beliebig viele Verbindungen zu Vertex-Klassen aufnehmen, genau wie eine transition-Kante beliebig viele Transition-Klassen aufnehmen kann. Ähnlich wie die Knotenklassen haben auch die Kantenklasse ein Attribut name um Zugriff auf reflexive Informationen zu unterstützen.

4.4. XMI importieren

Das Einlesen von XMI-Dokumenten folgt dem offiziellen Entwurfsmodell[2] für XMI-Dokumente in der OMG MOF/XMI Abbildungsspezifikation [OMG07b]. Das Entwurfsmodell unterscheidet die Verwendung von eingebetteten XMI-Elementen oder XMI-Attributen während der Modellierung. Wir modellieren dabei XMI-Attribute, da Altova UModel die XMI-Dokumente nur auf diese Weise lesen kann (Man beachte, dass es sich immer noch um ein standardisiertes Entwurfsmodell der OMG handelt). Jedes XMI 2.1-konforme Dokument kann so als Graph importiert werden – vorausgesetzt, die korrekte(n) Modelldefinition(en) mit den entsprechenden Knoten- und Kantenklassen liegen vor. Da der Importfilter MOF-spezifizierte Dokumente liest, ist der hier vorgestellte Ansatz generisch und nicht an die UML gebunden, sondern kann jedes durch MOF beschriebene Modell importieren.

Ein XMI-Dokument besteht aus verschiedenen XMI-Strukturelementen, siehe Abbildung 4.4. XMIObjectElemente können genau wie XMI-Elemente weitere Attribute und Kindelemente enthalten. XMIValueElemente haben zwar einen Wert, aber keine weiteren

[2] Kapitel 6.5.2 „EMOF Package“ sowie Kapitel 6.4 „EBNF Rules Representation“ beschreiben das verwendete Entwurfsmodell.

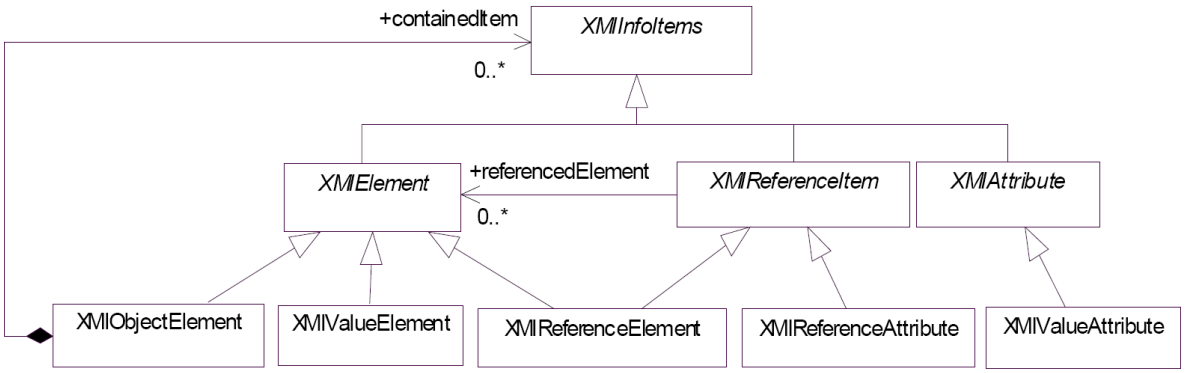

Abbildung 4.4.: Modell der XMI-Strukturen

Attribute oder Kindelemente. Elemente mit einem idref oder href Attribut werden als XMIReferenceElement bezeichnet. Sie referenzieren ein anderes XMIElement über eine id, URI oder URI und XPointer. Ein XMIReferenceAttribute ist ein Attribut, dass ein XMI-Element über seine id referenziert. Ein XMIValueAttribute ist ein einfaches Wertattribut.

Das XMI-Dokument hat immer ein festgelegtes Initial- bzw. Wurzelelement vom Typ Package. In GrGen.NET wird die entsprechende Knotenklasse instanziiert. Für alle weiteren im Dokument vorkommenden XMI-Elemente und -Attribute gilt folgende Entwurfsregel:

- Tagnamen und Attribute des Standard-XML-Namensraums sind Attribute: Sie werden als Kanten dargestellt (siehe Kapitel 4.3). Der Element- bzw. Attributname gibt gleichzeitig den Kantentyp an.

- Die durch die Kanten zu verbindenden Knotenklassen stehen in dem type-Attribut des XMI-Namensraumes (*xmi:type*). Knotenklassen sind Unterknotenklassen von CMOFClass.

- Ist ein Wert keine Unterknotenklasse der CMOFClass, dann handelt es sich entweder um einen primitiven Datenwert (String, Boolean, Integer, UnlimitedNatural), ein Literal einer Enumeration oder ein XMIReferenceAttribute. Wir bestimmen die Information über den zu verwendenden Typ des Wertes an Hand einer in das XMI-Dokument eingefügten Metadatenliste. Wird kein entsprechender Typ in der Liste gefunden, dann handelt sich um ein XMIReferenceAttribute.

Die eingefügte Metadatenliste ist eine Liste aller notwendigen Informationen über jedes in der Normierungsebene vorkommende UML-Element. Die gesammelten Informationen liegen im Quelldokument "L3.merged.cmof", siehe Kapitel 4.1 als Attribute vor und werden unverändert übernommen. Die Liste wird für den Importvorgang vorher in

das XMI-Diagramm eingefügt. Die während dem Import benötigten Daten können so direkt aus dem XMI-Diagramm ausgelesen werden, wobei die Hauptaufgabe in der Typbestimmung eines UML-Elements besteht. Die Liste wird genau wie der Importer oder die automatische Modellabbildung mit einem XSL Transformationsskript generiert.

Das Diagramm wird von unserem XSL Transformationsskript automatisch in eine Modelldefinition für GrGen.NET abgebildet (siehe Codebeispiel 4.2). Dabei wird das Dokument rekursiv abgearbeitet bis es auf die Blattelemente trifft. Die Blattelemente garantieren die Terminierung des im Transformationsskript angewandten Algorithmus.

```
<!-- This template links the parent-node to its child-nodes
         with its corresponding edge type. -->
<xsl:template name="link-parentNode-with-childNodes">
<xsl:param name="parent-node" />
<xsl:param name="child-nodes" />
  <!-- For each NON-empty child node ... -->
  <xsl:for-each select="$child-nodes[@xmi:type]">
    <!-- ... create the child node, ... -->
    <xsl:text>new </xsl:text>
    <xsl:value-of select="@xmi:id" />
    <xsl:text> : </xsl:text>
    <xsl:value-of select="substring(@xmi:type,5)" />
    <xsl:text>()</xsl:text>
    <!-- ... create the edge ... -->
    <xsl:text>new </xsl:text>
    <!-- ... beginning at the parent ... -->
    <xsl:value-of select="$parent-node/@xmi:id"/>
    <xsl:text> -: </xsl:text>
    <!-- ... with the according edge type ... -->
    <xsl:value-of select="name()"/>
    <!-- ... referencing its child  (actual context) node -->
    <xsl:text> -> </xsl:text>
    <xsl:value-of select="@xmi:id"/>
    <!-- Apply this template recursively to the child element until it encounters
            leafs. The recursive application of the template will then stop. -->
    <xsl:call-template name="link-parentNode-with-childNodes">
      <xsl:with-param name="parent-node" select="."/>
      <xsl:with-param name="child-nodes" select="./child::node()"/>
    </xsl:call-template>
  </xsl:for-each>
</xsl:template>
```

Auszug 4.2: Ein Auszug der XSL Transformationen: Verbindet Vaterelement mit seinem Kindelement

```
1  <uml:Package xmi:id="u01" name="Root">
2     <packagedElement xmi:type="uml:Package" xmi:id="ux8"
3              name="StateMachinePackage" visibility="public">
4           <packagedElement xmi:type="uml:StateMachine" xmi:id="u34" name="StateMachine">
5                 <region xmi:type="uml:Region" xmi:id="u87" name="Region">
6                 <subvertex xmi:type="uml:Pseudostate" xmi:id="u9f">
7                    <outgoing xmi:idref="f90" />
8                 </subvertex>
9                 <transition xmi:type="uml:Transition" xmi:id="f90" source="u9f"
10                                                                  target="u7e" />
11                <subvertex xmi:type="uml:Pseudostate" xmi:id="u7e">
12                   <incoming xmi:idref="f90" />
13                </subvertex>
14          </region>
15       </packagedElement>
16    </packagedElement>
17 </uml:Package>
```

Auszug 4.3: UML2 Zustandsautomat als XMI-Dokument

4.4.1. Anwendungsbeispiel

Die vorher definierten Begriffe des XMI-Modells werden hier an einem Beispiel ausführlich erläutert. Folgendes XMI-Beispiel in Codebeispiel 4.3 dokumentiert einen Teil des Zustandsautomaten in Kapitel 3. Zwei Zustände werden mit einem Zustandsübergang verbunden. Abbildung 4.5 demonstriert das gezeigte XMI-Dokument als UML-Graph in GrGen.NET.

Das erste vorkommende Element packagedElement in Zeile 2 ist ein XMIObjectElement. Es enthält weitere XMI-Elemente und -Attribute. Alle darin enthaltenen Attribute sind XMIValueAttribute die über den jeweiligen namengebenden Kantentyp mit dem zugehörigen Knoten verbunden sind. Die Auflösung des Knotentyps erfolgt über die Metadatenliste, siehe Kapitel 4.4. Weitere XMIObjectElemente sind packagedElement, region, subvertex und transition. XMIValueAttribute sind name, visibility, xmi:type und xmi:id.

Neben XMIValueAttributen gibt es auch XMIReferenceAttribute. Das in dem transition Element auftauchende source-Attribut (Zeile 9) ist ein solches XMIReferenceAttribute. Es verbindet den Transition-Knoten über eine source-Kante mit einem Element welches das id-Attribut mit dem Wert „u9f“ besitzt. In diesem Fall ist es das Pseudostate-Element in Zeile 6. Ein weiteres XMIValueAttribute ist das target-attribut.

Das outgoing Element in Zeile 7 ist ein XMIReferenceElement: Eine outgoing-Kante verbindet den Vaterknoten Pseudostate mit dem Knoten Transition. Die Verbindung wird dabei an Hand der id aufgelöst. Ein weiteres XMIReferenceElement ist incoming.

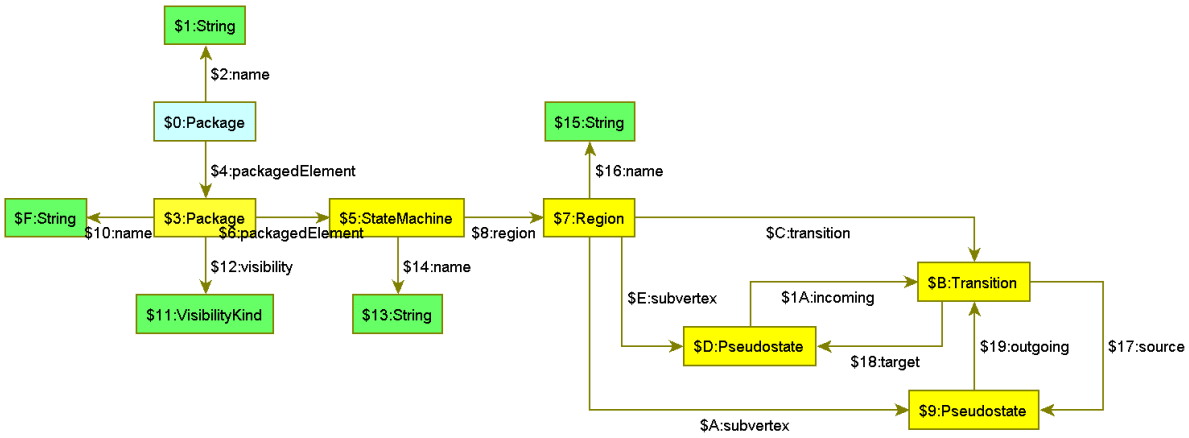

Abbildung 4.5.: Teil des UML2 Zustandsautomaten als Graph in GrGen.NET

4.5. XMI exportieren

Gemäß dem Entwurfsmodell aus Kapitel 4.4 ist das Exportieren eines XMI-Dokuments ein transparenter Vorgang. Der Graph wird vor dem Exportvorgang serialisiert: Er wird ausgehend von einem festgelegten Wurzelknoten (das *Root-Package*) in Hierarchieebenen unterteilt. Speziell für die Serialisierung wurde in die Knotensuperklasse CMOFNODE[3] ein integer-Attribut level deklariert. Die hierarchische Struktur des entstehenden XMI-Dokuments verlangt nach einer Unterteilung des Graphen in Hierarchieebenen. Jeder Folgeknoten eines Knotens, ausgenommen solche vom Typ CMOFValueType-Knotenklassen, stellen in der XMI-Struktur ein Kindelement dar. Für die Serialisierung bedeutet das die Hierarchieebene des Folgeknotens um eins zu erhöhen. Die oberste Hierarchieebene beginnt dabei bei Null. Der Vorgang wird solange rekursiv wiederholt bis alle Knoten einer Ebene zugeteilt wurden.

Nach erfolgreicher Serialisierung des Graphen beginnt das Exportieren in ein XMI-Dokument. Genau wie der Serialisierer, beginnt auch der Exporter bei dem selben festgelegten Wurzelknoten und arbeitet sich rekursiv auf die nächsthöhere Hierarchieebene

[3] Jede Knotenklasse im abgebildeten UML-Modell ist eine direkte oder indirekte Knotenunterklasse von CMOFNODE.

weiter. Das level-Attribut wird nun verwendet um das XMI-Dokument in einer korrekten hierarchischen Struktur auszugeben. Um Zyklen und somit endlos laufende Rekursionen zu vermeiden, wird nach jedem Durchlaufen eines Elements die vorher benutzte Kante aus dem UML-Graphen als *besucht* markiert. So wird ein terminierender Algorithmus garantiert. Ein Element mit einem Eigentümer (*owned elements*) wird dabei als XMIElement ausgegeben. Elemente, die solche bereits deklarierten Elemente referenzieren, werden als XMIReferenceElemente in den Ausgabestrom geschrieben.

Codebeispiel 4.4 zeigt einen Auszug des Regelsatzes für den XMI-Exporter. Instanzen von CMOFClass werden der Regel als Parameter übergeben. Das Suchmuster in Zeile 5 verlangt nach einem Knoten das ein Attribut besitzt. Das Attribut ist über eine CMOFProperty-Kante verbunden (falls vorhanden). Die Regel wird nur dann auf dem gefundenen Graphen ausgeführt, wenn das gefundene Attribut tiefer in der Hierarchiestruktur liegt als das besitzende Vaterelement und die gefundene Eigenschaft noch nicht besucht wurde (um Wiederholungen und endlose Rekursionen zu vermeiden), siehe if-Bedingung in Zeile 7 bis 10. Treffen alle diese Bedingungen zu, dann wird die Transformation auf dem Graphen ausgeführt (Zeile 12 bis 31). Die Transformation erzeugt hier lediglich einen Ausgabestrom zur Ausgabe eines XMI-Dokuments führt aber keine graphersetzenden Schritte durch. Ausgabeströme werden durch emit-Befehle ausgegeben. Bei der Ausgabe wird darauf geachtet auch die Attribute sowie weitere Kindelemente auszugeben. Bei Attributen kann es sich um XMIValueattribute handeln (Zeile 19). Kindelemente können dabei XMIElemente und XMIReferenceElemente sein die in den Zeilen 23 bis 26 ausgegeben werden. Bei XMIElementen werden zuerst Elemente mit Eigentümern (ownedElements) ausgegeben. Nach erfolgreicher Ausgabe wird das XMI-Element geschlossen und die prop-Kante zur Verhinderung von Zyklen als *besucht* markiert.

```
/* Emits an XMIElement to the XMI document. An XMIElement can contain
 * XMIAttributes and/or other XMIElements. */
rule writeXmiElement(parent:CMOFClass) {
   // Declare the search pattern.
   parent -prop:CMOFProperty-> child:CMOFClass\(CMOFValueType);
   // Consider only patterns meeting the following condition.
   if {
      parent.level < child.level;
      prop.VISITED == false;
   }
   // The graph rewriting statement
   modify {
      // Set VISITED flag of traversed edge - breaks possible cycles
      eval { prop.VISITED = true; }
      // Emit the XMIElement's tag-name, its xmi:type and xmi:id.
      emit("\t<",  prop.name,"␣xmi:type=", "\"uml:", child.uuid,
                  "\"", "␣xmi:id=\"G-", child.id, "\"");
      // Emit value attributes, if present ...
      exec(writeXmiValueAttribute(child)[*]);
      // ... and close the elements tag.
      emit(">\n");
      // First emit owned elements ...
      exec(writeOwnedElement(child)[*]);
      // ... then emit reference elements ...
      exec(writeBackReference(child)[*]);
      exec(writeForwardReferenceIfOwnedByElement(child)[*]);
      // ... last but not least emit the children.
      exec(writeXmiElement(child)[*]);
      // Emit the closing tag of XMIElement.
      emit ("\t</",  prop.name,">\n");
   }
}
```

Auszug 4.4: Eine Regel des XMI-Exporters

5. Verwandte Arbeiten

MOF QVT der Object Management Group [OMG07c] ist die wohl bekannteste Spezifikation zur Beschreibung einer Sprache für Modell-zu-Modell-Transformationen. QVT steht für „queries“ (*Abfragen*), „views“ (*Sichten*) und „transformations“ (*Transformationen*). Abfragen sind Ausdrücke mit denen einzelne Elemente eines Modells beschrieben werden. Sichten sind komplexe Abfragen um Abschnitte eines Modells auszuwählen. Mit Transformationen werden die Beziehungen der Modelle untereinander dargestellt. QVT ist eine Mischform aus deklarativer und imperativer Sprache mit einer auf Dreifachgrammatik für Graphen aufbauenden Implementierung. „Spuren“ (*traces*) sind Verbindungen zwischen den verschiedenen Modellen. Diese Spuren werden auch nach einer Modelltransformation beibehalten, um Auswirkungen bei einer Veränderung des Modells auf das andere Modell sichtbar und nachvollziehbar zu machen. Eine im Sinne von QVT vollständige Implementierung würde einen Abbildungs- und Transformationsmechanismus in nur einem einzigen Softwaresystem anbieten. Es gibt allerdings bis heute keine vollständige Implementierung der Spezifikation. Verschiedene Implementierungen existieren zwar, weichen aber oft von der Spezifikation und ihren Anforderungen ab. Und selbst bei einer existierenden QVT-Implementierung müssten die notwendigen Sichten und Transformationen für das UML2 Modell erst implementiert werden. Der Einsatz von QVT ist somit genauso geeignet oder ungeeignet wie der Einsatz von GrGen.NET oder einem anderen Graphersetzungssystem.

GREAT [Chr04] ist ein regelbasiertes Werkzeug zur Transformation von Modellen die auf der selben oder verschiedenen Metaebenen liegen. GREAT ermöglicht UML-Modell-zu-Modell-Transformationen, wie Strukturverbesserungen, Neuorganisation oder Modelloptimierung. Einfache Anwendungsbeispiele wären das strukturverbessernde Umbenennen einer Klasse (refactoring) oder das Anlegen von Entwurfsmustern (restructuring). Die Modelltransformationen werden durch das Graphersetzungssystem OPTIMIX [Ass99] verarbeitet. Ähnlich wie bei unserem Ansatz wird das UML-Modell auch von einer XMI-Datei importiert und bietet ebenfalls einen XMI-Exportfilter für einen erfolgreichen Im- und Export des transformierenden Modells. Die Menge der verwendbaren UML-Modelle ist jedoch stark eingeschränkt, da GREAT lediglich UML-Klassendiagramme verarbeitet und nicht die komplette UML abdeckt.

Mit GReAT [Agr03] wurde ein spezielles Graphersetzungssystem zur schnellen Entwicklung von domänenspezifischen Sprachen (DSL – domain specific language) entworfen. Mit der allgemeinen Modellierungsumgebung (GME – generic modeling environment) baut es auf einer visuellen Eingabesprache mit textuellen Bezeichnungen auf die eine eingegebene Sprache in Form eines Graphen wiedergibt. GReAT wird so ermög-

licht, eine in der GME eingegebenen, Domänensprache zu verarbeiten. Textuelle Ausgaben können nur durch Implementierung spezieller Besuchermuster (Entwurfsmuster) in C++ erreicht werden. Im Gegensatz dazu bietet GrGen.NET eine nahtlose Integration der Ausgabe in den Regelsätzen. Vergleiche [TBB+08] zeigen außerdem, dass eine gemischte textuelle und visuelle Eingabe eines Modells sehr langatmig sein kann. Die Ausführungsgeschwindigkeit von GReAT ist ziemlich gering. Es existiert zudem keine *bekannte* Domänensprache der kompletten UML. Unser Ansatz könnte eine Lösung zur Modellierung der UML in GReAT bieten.

VIATRA2 [VB07] ist ein speziell für Modelltransformationen entwickeltes Graphersetzungssystem. Es baut auf einem abstraktem Zustandsautomaten zur Verarbeitung der Graphersetzungsregeln auf. Modelle für VIATRA2 werden mit VPM, ein Werkzeug zur Beschreibung visueller, präziser und mehrschichtiger Metamodelle für mathematische Arbeitsbereiche und die UML, definiert. Damit sind Modelle auf verschiedensten Metaebenen möglich und sind generischer als MOF spezifizierte Modelle. VIATRA2 bezahlt seine hohe Generizität allerdings mit hohen Leistungseinbußen: eine Vielzahl der Regelwerke bieten reflexive Bearbeitungen an, die resultierenden Regeln können nur schwer zu effizienten Implementierungen abgebildet werden. Durchgeführte, abhängige und unabhängige, Analysen und deren Bewertungen [CHM+02, SNZ08] zur Performanz deuten an: echte Transformationen an nicht-trivialen Modellen dauern zu lange für eine nahtlose Integration in einen bestehenden Softwareprozess.

EMF Tiger (EMT) [Tae07] ist ein auf EMF/GMF (Eclipse/Graphical Modeling Framework) basierendes Werkzeug zur Erstellung von visuellen Editoren verschiedener Modelle für das Graphersetzungssystem AGG. Das Eclipse Modellierungswerkzeug (EMF) erzeugt dabei Java Code aus einem gegebenen XMI-Dokument. EMT ist zwar fähig XMI-Dokumente zu lesen, aber das einzige genutzte Modell um die Elemente des Dokuments darzustellen ist EMOF, eine Untermenge von CMOF. Man kann also die UML-Spezifikation lesen und zu etwas anderem transformieren. Ebenso kann man ein im XMI-Dokument gespeichertes UML-Modell einlesen und transformieren. Eine Verbindung zwischen dem Modell und dem Metamodell kann aber nicht hergestellt werden. Daraus folgt logischerweise Typunsicherheit, da die Verbindung zum Modell für die Instanz nicht hergestellt werden kann. Genau wie GReAT und VIATRA2 bietet auch EMT eine unzureichende Performanz für echte, nichttriviale Modelltransformationen.

6. Fazit

Graphersetzungssysteme wie GrGen.NET brauchen eine Anpassung an die Anwendungsdomäne. Im Fall der Modelltransformationen brauchen wir ein an die UML angepasstes Graphmodell. Da die UML in einem maschinenlesbaren XMI-Quelldokument vorliegt, waren wir in der Lage XSL Transformationsskripte zur Verfügung zu stellen um die UML-Modelldefinition automatisch erzeugen zu lassen.

Wir haben zusätzliche Im- und Exportfilter entwickelt um die transformierten Modelle mit der Außenwelt austauschen zu können. Der Erfolg dieses Ansatzes wurde an einem beispielhaften Im- und Export eines transformierten Modells in und von Altova UModel 2008 gezeigt. Eine Serialisierung des Graphen bot dabei die Möglichkeit den Graphen in ein XMI-Dokument zu exportieren ohne auf spezielle reflexive Informationen zugreifen zu müssen. Der Exporter kann XMI-Dokumente auch von Grund auf erzeugen. Die Möglichkeit XMI-Dokumente aus einem manuell erzeugten Graphen zu erzeugen schafft neue kreative Einsatzmöglichkeiten. Andere Systeme könnten einen UML-Graphen erzeugen und mit Hilfe des Exporters das zugehörige UML-Diagramm in einem UML-Werkzeug laden. Ein System, dass den Exporter erfolgreich nutzt ist Sale [GT07]. Sale liest ein speziell annotiertes Anforderungsdokument um daraus einen Graphen zu erstellen. Aus diesem Graphen kann dann ein ein UML-Diagramm erzeugt werden ohne vorher ein XMI-Quelldokument importieren zu müssen.

Bei dem Exportvorgang wird das XMI-Dokument gemäß dem beschriebenen standardisierten Entwurfsmodell für XMI-Dateien generiert. Die geschriebenen Skripte sind – wie gezeigt – dabei nicht auf die UML2 beschränkt. Eine automatische Modellgenerierung und das Im- und Exportieren von Instanzen des Modells ist für jedes beliebige MOF basierte Modell möglich.

Der vorgestellte Ansatz hat auch gezeigt, dass eine Implementierung des Modelltransformationsskripts und der Im- und Exportfilter nicht überdurchschnittlich aufwändig ist. Wir haben mit unseren Erweiterungen für GrGen.NET [GDG08] gezeigt, dass es möglich ist eine passende Modelldefinition für jedes beliebige typisierte Graphersetzungssystem automatisch zu generieren. Wir haben uns hier auf den statischen Teil der UML2 konzentriert und alle Diagrammtypen abgedeckt. Dies konnte mit nur unter 1000 Zeilen Code[1] implementiert werden.

[1] Transformationsskript zur Modelldefinition: 476 Zeilen, Importer: 350 Zeilen, Exporter: 173 Zeilen.

A. Regeln

A.1. epsilon–Befreiung

Die folgenden Regeln sind alle in einem Regelsatz (eine Regeldatei *.grg) geschrieben worden. Um die folgenden Regeln auszuführen sollten sie in einer Datei, beispielsweise removeEps.grg geschrieben stehen und mit folgender Befehlssequenz aufgerufen werden.

- new graph removeEps
- include myStatemachineWithEpsilons.grs
- xgrs (sm)=findStateMachine && ((%computeEpsilonHull(sm)+ | (forwardTransition3States(sm)|forwardTransition2States(sm))+)* | removeEpsilonTransition(sm)*)

Abbildung A.1.: Befehlssequenz zur Graphtransformation

```
1 /* Finds a statemachine in the graph and returns it. */
2 rule findStateMachine : (StateMachine) {
3    sm : StateMachine;
4    modify { return (sm); }
5 }
```

Auszug A.1: Findet einen Zustandsautomaten

```
/* Computes an epsilon hull of a given state machine. */
rule computeEpsilonHull(sm:StateMachine) {
   // A region  must have at least three states x, y and z. x and y are
   // connected by a transition xy, y and z by a transition yz.
   sm -:region-> reg:Region;
   reg -:subvertex-> x:State;
   reg -:subvertex-> y:State;
   reg -:subvertex-> z:State;
   reg -:transition-> xy:Transition;
   reg -:transition-> yz:Transition;
   hom(x,z); // x and z can be the same -> homomorph.
   // The connection sequence of the states and transitions.
   x <-:source- xy -:target-> y <-:source- yz -:target-> z;
   // xy must not have a trigger -> It must be a epsilon transition.
   negative { xy -:trigger->; }
   // The transition yz must not have a trigger
   //    => it must be an epsilon transition.
   negative { yz -:trigger->; }
   // A transitive connection between the states x and z must not occur,
   // where the transition xz may not be epsilon transition.
   negative {
      reg -:transition-> xz:Transition;
      x <-:source- xz -:target-> z;
      negative { xz -:trigger->; }
   }
   // If such a pattern is found, we compute a transitive transition xz
   // and add it to the graph.
   modify {
      reg -:transition-> xz:Transition;
      x <-:source- xz -:target-> z;
      z -:incoming-> xz <-:outgoing- x;
   }
}
```

Auszug A.2: Berechnet eine transitive Hülle

```
/*
 * Creates a forward transition for a two-transition sequence within three
 * states. First transition is an epsilon, second transition is a
 * triggered transition. A forward transition will be created to avoid
 * using the epsilon transitions.
 */
rule forwardTransition3States(sm:StateMachine) {
   // A region  must have at least three states x, y and z, where z
   // is no final state. x and y are connected by a transition xy,
   // y and z by a transition yz.
   sm -:region-> reg:Region;
   reg -:subvertex-> x:State;
   reg -:subvertex-> y:State;
   reg -:subvertex-> z:State\(FinalState);
   reg -:transition-> xy:Transition;
   reg -:transition-> yz:Transition;
   // The names of the states.
   x -:name-> xname:String;
   y -:name-> yname:String;
   z -:name-> zname:String;
   hom(x,z); // x and z can be the same -> homomorph.
   // The connection sequence of the states and transitions.
   x <-:source- xy -:target-> y <-:source- yz -:target-> z;
   yz -:trigger-> trig:Trigger; // yz is no epsilon transition.
   trig -:name-> trigName:String;
   // xy must be an epsilon transition.
   negative { xy -:trigger->; }
   // Prohibit multiple triggered transitions with same name.
   negative {
      hom(x,z);
      reg -:transition-> xz:Transition;
      x <-:source- xz -:target-> z;
      xz -:trigger-> trig2:Trigger;
      trig2 -:name-> trig2Name:String;
      if{ trigName.value == trig2Name.value; }
   }
   // Create a new transitive forward transition with a trigger.
   modify {
      reg -:transition-> xz:Transition;
      x <-:source- xz -:target-> z;
      z -:incoming-> xz <-:outgoing- x;
      xz -:trigger-> trigNew:Trigger;
      trigNew -:name-> trigNewName:String;
      eval { trigNewName.value = trigName.value; }
   }
}
```

Auszug A.3: Führt eine transitive Transition ein

```
/*
 * Creates a forward transition for a two-transition sequence within
 * two states. First transition is an epsilon, second transition is
 * a triggered transition. A forward transition will be created to
 * avoid using the epsilon transitions.
 */
rule forwardTransition2States(sm:StateMachine) {
   // A region must have at least two states connected by transitions...
   sm -:region-> reg:Region;
   reg -:subvertex-> x:State;
   reg -:subvertex-> y:State;
   reg -:transition-> xy:Transition;
   reg -:transition-> yx:Transition;
   x -:name-> xname:String;
   y -:name-> yname:String;
   // ... with a particular sequence.
   x <-:source- xy -:target-> y <-:source- yx -:target-> x;
   yx -:trigger-> trig:Trigger; // yx is no epsilon transition.
   trig -:name-> trigName:String;
   // xy must be an epsilon transition.
   negative { xy -:trigger->; }
   // Prohibit multiple triggered transitions with same name.
   negative {
      reg -:transition-> xx:Transition;
      x <-:source- xx -:target-> x;
      xx -:trigger-> trig2:Trigger;
      trig2 -:name-> trig2Name:String;
      if{ trigName.value == trig2Name.value; }
   }
   // Create a new transitive forward transition with a trigger.
   modify {
      reg -:transition-> xx:Transition;
      x <-:source- xx -:target-> x;
      x -:incoming-> xx <-:outgoing- x;
      xx -:trigger-> trigNew:Trigger;
      trigNew -:name-> trigNewName:String;
      eval { trigNewName.value = trigName.value; }
   }
}
```

Auszug A.4: Führt eine transitive Transition ein

```
/* Creates a forward transition to a final state. */
rule propagateFinalState(sm:StateMachine) {
   // A region must have at least three states. Two of them are no
   // final states and one has to be a final state. They are
   // connected by transitions...
   sm -:region-> reg:Region;
   reg -:subvertex-> x:State\(FinalState);
   reg -:subvertex-> y:State\(FinalState);
   reg -:subvertex-> f:FinalState;
   reg -:transition-> xy:Transition;
   reg -:transition-> yf:Transition;
   // ... with a particular sequence.
   x <-:source- xy -:target-> y <-:source- yf -:target-> f;
   // Transition yf must not have a trigger to be an epsilon transition.
   negative { yf -:trigger->; }
   // Transition xy must not have a trigger to be an epsilon transition.
   negative { xy -:trigger->; }
   // Prohibit multiple triggered transitions with same name.
   negative {
      reg -:transition-> xf:Transition;
      x <-:source- xf -:target-> f;
   }
   // Create a new transitive triggered forward transition
   // leading to a final state.
   modify {
      reg -:transition-> xf:Transition;
      x <-:source- xf -:target-> f;
      f -:incoming-> xf <-:outgoing- x;
   }
}
```

Auszug A.5: Führt eine transitive Transition für Endzustände ein

```
/* Removes an epsilon transition in the given state machine. */
rule removeEpsilonTransition(sm:StateMachine) {
   // A transition must not be connected to a final state.
   sm -:region-> reg:Region;
   reg -:transition-> xy:Transition;
   xy -:target-> :State\(FinalState);
   // The transition xy must not leave an initial pseudostate.
   negative {
      reg -:subvertex-> x:Pseudostate;
      x <-:source- xy;
   }
   // Transition xy must not have a trigger to be an epsilon transition.
   negative { xy -:trigger->; }
   // Delete the epsilon transition.
   modify { delete(xy); }
}
```

Auszug A.6: Entfernt eine ϵ–Transition

Literaturverzeichnis

[Agr03] Agrawal, Aditya: Metamodel Based Model Transformation Language. In: Crocker, Ron (Hrsg.) ; Jr, Guy L. S. (Hrsg.): *OOPSLA Companion*, ACM, 2003. – ISBN 1–58113–751–6, 386–387 27

[AKL03] Agrawal, Aditya ; Karsai, Gabor ; Lédeczi, Ákos: An End-to-End Domain-Driven Software Development Framework. In: Crocker, Ron (Hrsg.) ; Jr, Guy L. S. (Hrsg.): *OOPSLA Companion*, ACM, 2003. – ISBN 1–58113–751–6, 8–15

[Alt08] Altova: *UModel – UML tool for software modeling and application development*. http://www.altova.com/products/umodel/uml_tool.html, Januar 2008 10

[Ass99] Assmann, Uwe: OPTIMIX – a tool for rewriting and optimizing programs. In: *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools* (1999), S. 307–318. ISBN 981–02–4020–1 27

[BG07] Blomer, Jakob ; Geiss, Rubino ; Universität Karlsruhe (TH), Institute for Program Structures and Data Organization (IPD) (Hrsg.): *The GrGen.NET User Manual - Refers to GrGen.NET Release 1.0*. Karlsruhe: Universität Karlsruhe (TH), Institute for Program Structures and Data Organization (IPD), July 2007. http://www.grgen.net

[CHM+02] Csertan, G. ; Huszerl, G. ; Majzik, I. ; Pap, Z. ; Pataricza, A. ; Varró, D.: VIATRA – visual automated transformations for formal verification and validation of UML models. In: *Proc. 17th IEEE International Conference on Automated Software Engineering ASE 2002*, 2002, S. 267–270 3, 28

[Chr04] Christoph, Alexander: Describing Horizontal Model Transformations with Graph Rewriting Rules. In: Assmann, Uwe (Hrsg.) ; Aksit, Mehmet (Hrsg.) ; Rensink, Arend (Hrsg.): *MDAFA* Bd. 3599, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–28240–8, 93–107 3, 27

[CS03] Caplat, Guy ; Sourrouille, Jean L.: Considerations about Model Mapping. In: *Workshop in Software Model Engineering (WiSME@UML'2003)*. San Francisco, USA, Oktober 2003 14

[Den07] Denninger, Oliver: *Erweiterung des Kantenkonzepts deklarativer Graphersetzungssysteme von Einfachkanten über Hyperkanten zu „Superkanten“*, Universität Karlsruhe, IPD Tichy, Diplomarbeit, 2007

[DGG08] Denninger, Oliver ; Gelhausen, Tom ; Geiss, Rubino: Applications and Rewriting of Omnigraphs – Exemplified in the Domain of MDD. In: Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, Springer-Verlag, 2008 (LNCS) 17

[Fra03] Frankel, David S. ; Hudson, Theresa (Hrsg.): *Model Driven Architecture*. Wiley Publishing, 2003. – Official ÖMG Press"Publisher, ISBN-10: 0-471-319-20-1

[GBG+06] Geiss, Rubino ; Batz, Gernot V. ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, Andrea (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Montanari, Ugo (Hrsg.) ; Ribeiro, Leila (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *ICGT* Bd. 4178, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–38870–2, 383–397 3

[GDG08] Gelhausen, Tom ; Derre, Bugra ; Geiss, Rubino: *The GrGen.NET MOF-Suite*. http://www.grgen.net/mof, Januar 2008 29

[Gei08] Geiss, Rubino: *GrGen.NET*. http://www.grgen.net/. http://www.grgen.net/. Version: Januar 2008

[GT07] Gelhausen, Tom ; Tichy, Walter F.: Thematic Role based Generation of UML Models from Real World Requirements. In: *First IEEE International Conference on Semantic Computing (ICSC 2007)* Bd. 0. Irvine, CA, USA : IEEE Computer Society, September 2007, 282-289 29

[HWS00] Holt, Richard C. ; Winter, Andreas ; Schürr, Andy: GXL: Toward a Standard Exchange Format. In: Winter, A. (Hrsg.): *Seventh Working Conference on Reverse Engineering*, 2000, S. 162–171 3

[Kay01] Kay, Michael: *XSLT: Programmer's Reference*. 2nd. Wrox Press, 2001. – ISBN-10: 1-861-005-06-1

[Kay07] Kay, Michael ; World Wide Web Consortium (W3C) (Hrsg.): *XSL Transformations (XSLT)*. 2.0. World Wide Web Consortium (W3C), January 2007. http://www.w3.org/TR/xslt20/

[OMG01] OMG ; Object Management Group (Hrsg.): *Model Driven Architecture (MDA)*. Object Management Group, July 2001. http://www.omg.org/mda/. – Document number: ormsc/2001-07-01

[OMG03] OMG ; Object Management Group (Hrsg.): *Common Warehouse Metamodel (CWM) Specification*. 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: Object Management Group, März 2003. http://www.omg.org/docs/formal/03-03-02.pdf 7

[OMG06a] OMG ; Object Management Group (Hrsg.): *Meta Object Facility (MOF) Core Specification*. 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: Object Management Group, Januar 2006. http://www.omg.org/spec/MOF/2.0/ 5

[OMG06b] OMG ; Object Management Group (Hrsg.): *MOF Core specification*. 2.0. Object Management Group, January 2006. http://www.omg.org/spec/MOF/2.0/. – Document number: formal/2006-01-01

[OMG06c] OMG ; Object Management Group (Hrsg.): *New XMI 2.1.1 specification – UML 2.1.1 XMI file(s)*. 2.1.1. Needham, MA 02494, USA: Object Management Group, Oktober 2006. http://www.omg.org/cgi-bin/doc?ptc/2006-10-06 13

[OMG07a] OMG ; Object Management Group (Hrsg.): *Infrastructure specification*. 2.1.2. Object Management Group, November 2007. http://www.omg.org/spec/UML/2.1.2/. – Document number: formal/2007-11-04 5

[OMG07b] OMG ; Object Management Group (Hrsg.): *MOF 2.0/XMI Mapping, Version 2.1.1*. formal/2007-12-01. 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: Object Management Group, Dezember 2007. http://www.omg.org/spec/XMI/2.1/ 3, 19

[OMG07c] OMG ; Object Management Group (Hrsg.): *MOF Query/View/Transformation Specification*. 2.0. Object Management Group, July 2007. http://www.omg.org/. – Document number: ptc/07-07-07 3, 27

[OMG07d] OMG ; Object Management Group (Hrsg.): *Systems Modeling Language (OMG SysML)*. 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: Object Management Group, September 2007. http://www.omg.org/docs/formal/07-09-01.pdf 7

[OMG07e] OMG ; Object Management Group (Hrsg.): *Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. 140 Kendrick Street, Building A Suite 300 Needham, MA 02494, U.S.A.: Object Management Group, November 2007. http://www.omg.org/spec/UML/2.1.2/ 9, 13, 14

[SNZ08] Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *AGTIVE - Applications of Graph Transformation 2007*. Bd. *Proc. 3rd Intl.*

Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07). Springer-Verlag, 2008 (LNCS) 3, 28

[Tae07] Taentzer, Gabriele: *Tiger EMF Transformation*. http://tfs.cs.tu-berlin.de/emftrans, 2007 28

[TBB+08] Taenzer, Gabriele ; Biermann, Enrico ; Bisztray, Dénes ; Bohnet, Bernd ; Boneva, Ivonka ; Boronat, Artur ; Geiger, Leif ; Geiss, Rubino ; Horvath, Acos ; Kniemeyer, Ole ; Mens, Tom ; Ness, Benjamin ; Plump, Detlef ; Vajk, Tamás: Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In: Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, Springer-Verlag, 2008 (LNCS) 28

[VB07] Varró, Dániel ; Balogh, András: The model transformation language of the VIATRA2 framework. In: *Sci. Comput. Program.* 68 (2007), Nr. 3, S. 187–207. http://dx.doi.org/http://dx.doi.org/10.1016/j.scico.2007.05.004. – DOI http://dx.doi.org/10.1016/j.scico.2007.05.004. – ISSN 0167–6423 28

[VVP02] Varró, Dániel ; Varró, Gergely ; Pataricza, András: Designing the automatic transformation of visual languages. In: *Sci. Comput. Program* Bd. 44 Department of Measurement and Information Systems, Budapest University of Technology and Economics, H-1521, Budapest, Magyar tudósok körútja 2, Hungary, 2002, S. 205–227