Seminar Formale Software-Entwicklung SS '08: Towards Verified Model Transformations

Martin Küster

Institut für Theoretische Informatik, Logik und Formale Methoden
Prof. Peter H. Schmitt, 76131 Karlsruhe, Germany
martin.kuester@stud.uni-karlsruhe.de
http://lfm.iti.uni-karlsruhe.de/

Stichworte: MDA, modellgetriebene Softwareentwicklung, Modelltransformation, Triple Graph Grammatiken, Verifikation

1 Einleitung

Modelle spielen spätestens seit der Verbreitung und Standardisierung der Unified Modeling Language (UML) eine zentrale Rolle in der Softwareentwicklung. Objekt-orientierter Enwurf wird sogar oftmals gleichgesetzt mit Klassenmodellierung in UML. Der gewünschte Effekt, dass die Betrachtungsweise vom Code gelöst und auf eine höhere Abstraktionsebene gehoben wird, erfüllt sich jedoch nicht immer: nach Fertigstellung der Software dienen die Modelle oft als bloße Dokumentation der Software. Dies resultiert aus der Tatsache, dass stets eine Umsetzung der Modelle in Code von Hand nötig ist und Änderungen direkt im Code erfolgen, nicht aber rückwirken auf die zu Grunde liegende Modellierung. Kürzer werdende Entwicklungszyklen (u.a. durch agilere Entwicklungsprozesse) verschärfen diese Problem sogar und führen dazu, dass ein hoher Bedarf an Wartung und Modernisierung bestehender Software besteht, bei dem die bereits entwickelten Modelle nicht (oder nur in geringem Maße) dienlich sind. Dies wird auch als *legacy crisis* bezeichnet.

Modell*getriebene* Softwareentwicklung versucht, dieses grundlegende Problem zu beheben, indem sie das Modell ins Zentrum des Entwicklungsprozesses rückt. Änderungen an der Software bedingen so stets Änderungen am Modell, die wiederum (automatisch) den Code anpassen. Um bestehende Lösungen wiederverwenden zu können, müssen Modelle ggf. in die existierende Modellstruktur (das sog. Metamodell) transformiert werden. Der Transformation als Brücke zwischen Modellen kommt eine zentrale Rolle zu, und sie verdient daher besondere Betrachtung.

Genau wie im Übersetzerbau, wo die Überführung einer Sprache in eine zweite die Semantik erhalten muss, so ist bei der Modell-zu-Modell-Transformation wünschenswert (notwendig?), dass das Quellmodell in ein semantisch äquivalentes Zielmodell überführt wird. Verifikation verfolgt das Ziel, dies formal zu beweisen. Während die Semantik dynamischer Systeme wie endlichen Automaten exakt durch mathematische Konstrukte (Eingabe-/Ausgaberelationen, partielle Funktionen,...) beschrieben werden kann, fehlt eine solche Möglichkeit der

Formalisierung oft für Software, in denen modellgetriebene Softwareentwicklung zum Einsatz kommt.

Einen Ansatz in Richtung automatischer Verifikation von Modelltransformationen stellt Giese et al. in [1] vor. Diese Veröffentlichung ist aus einer Diplomarbeit von Leitner [2] am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe (Prof. Goos) / Institut für Softwaretechnik und Theoretische Informatik der TU Berlin (Prof. Glesner) hervorgegangen und bildet die wesentliche Grundlage dieser Seminararbeit. Die Idee, Modelltransformationen möglichst automatisch darauf zu überprüfen, ob sie bestimmte Korrektheitskriterien erfüllen, ist sofort faszinierend. Inwieweit die Arbeit von Leitner et al. ein generisches Vorgehen präsentiert, wird in dieser Seminararbeit versucht zu beleuchten.

Die Arbeit gliedert sich wie folgt: In Abschn. 2 werden die Grundlagen der (Meta-)Modellierung beschrieben. In Abschn. 3 wird der in [2] verwendete Transformationsansatz der Triple-Graph-Grammatiken (TGGs) vorgestellt. Abschn. 4 widmet sich detaillierter dem Verifikationsverfahren. In Abschn. 5 wird eine Fallstudie präsentiert, die einen Anwendungsfall für das Verifikationsverfahren darstellt. Abschn. 6 schließt mit einer Bewertung und kurzem Ausblick.

2 (Meta-)Modellierung

Für die modellgetriebene Softwareentwicklung ist das Konzept der Metamodellierung essenziell. Während ein Modell ein Element der Echtwelt beschreibt, legt ein Metamodell eine 'Spache' fest, wie solche Modelle aussehen. Das beinhaltet insbesondere die Definition folgender Teile:

- **Abstrakte Syntax**: Modellelemente und deren Beziehung untereinander (unabhängig von der tatsächlichen Repräsentation)
- **Konkrete Syntax**: mind. eine Beschreibung, wie Modellelemente tatsächlich repräsentiert werden
- Statische und dynamische **Semantik**

Ein gängiges (weil in der Informatik allgegenwärtiges) Beispiel, das die verschiedenen Teile eines Metamodells illustriert, ist die Beschreibung eines endlichen Automaten: während die abstrakte Syntax nur festlegt, dass ein endlicher Automat aus (ggf. Start- oder End-)Zuständen und Übergängen besteht und dass zwischen Zuständen Übergänge bestehen, besteht eine konkrete Syntax z.B. aus einem Zustandsübergangsdiagramm mit Kästchen und Pfeilen. Eine andere konkrete Syntax ist z.B. die Formulierung durch textuelle Angabe der Mengen (Zustände, Startzustand, Finalzustände und Übergänge) sowie der Übergangsrelation.

Der Zusammenhang zwischen Realwelt, Modellen und Metamodellen wird in Abbildung 1 illustriert. Zentral ist dabei, dass Modelle stets Instanzen eines Metamodells sind. Während es wie oben beschrieben stets mehrere Formalisierungsmöglichkeiten für Metamodelle gibt, wird im Folgenden die von der

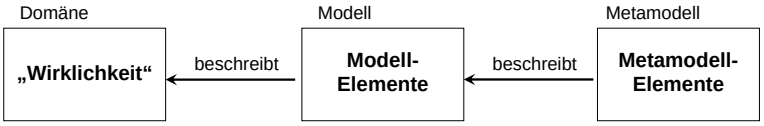

Abbildung 1. Beziehung zwischen Wirklichkeit, Modellebene und Metamodellebene (nach [3])

OMG[1] entwickelte Beschreibungssprache MOF (Meta Object Facility) benutzt, die selbstbeschreibend[2] ist und damit das obere Ende der Modellhierarchie darstellt. Mit ihr können alle Metamodelle beschrieben werden. Dazu bedient sie sich UML-ähnlicher Konstrukte wie Klassen und Assoziationen sowie insbesondere Stereotypen. Als konkrete Syntax wird die von UML-Klassendiagrammen bekannte Kästchen-Pfeil-Repräsentation gewählt.

3 Modelltransformationen mit Triple-Graph-Grammatiken

Modelltransformationen können auf verschiedene Arten beschrieben werden. Eine neue und vielversprechende Spezifikation zur Formalisierung solcher Transformationen wurde kürzlich von der OMG herausgegeben. Dieser Standard legt sowohl eine deklarative als auch eine operative Sprache QVT[3] fest, mit der Modelltransformationen ausgedrückt werden können. Eine erste, freie Implementierung eines Übersetzers für diese Sprache wurde von ikv++[4] entwickelt.

Ein gänzlich anderer Ansatz zur Beschreibung von Modelltransformationen fußt auf den *Triple Graph Grammars* (TGGs), die von Schürr [4] zur Spezifikation von Graph-Übersetzern vorgeschlagen wurden. Dazu werden das Quell- und Zielmodell als gerichtete, typisierte Graphen aufgefasst, in denen durch eine Transformation parallel Ersetzungen vorgenommen werden. Ein dritter Graph (der sog. *Korrespondenzgraph*) stellt eine Verbindung zwischen Quell- und Zielgraphen her. Dieser erlaubt, komplexe Transformationen (also m:n-Beziehungen zwischen Quell- und Zielgraphen) zu beschreiben und Änderungen (Updates) der beiden Graphen zu verfolgen.

[1] OMG: Object Management Group, ein Konsortium, das internationale Standards entwickelt, darunter CORBA und UML
[2] Selbstbeschreibend: Konstrukte wie Klasse, Operation, Referenz etc. sind durch Elemente ausdrückbar, die im selben Metamodell definiert werden.
[3] QVT: Query / View / Transformation
[4] ikt++ technology ag. `http://www.ikv.de`

Eine TGG-Transformation besteht also aus den drei Graphen (Quell-, Ziel- und Korrespondenzgraph) sowie aus einer Menge von TGG-Produktionen. Diese setzen sich je aus drei Teilen[5] zusammen:

- Linke-Seite-Ersetzungsregel: Vorschrift, welche Struktur im Quellgraph (bzw. -modell[6]) ersetzt werden soll
- Rechte-Seite-Ersetzungsregel: Vorschrift, welche Struktur im Zielgraph ersetzt werden soll
- Korrespondenz-Regel: Überprüft (und setzt ggf.) Verbindungen zwischen dem Quellgraphen und dem Korrespondenzgraphen sowie zwischen dem Zielgraphen und dem Korrespondenzgraphen.

Dahinter steckt die Vorstellung, dass sich Quell- und Zielmodell simultan entwickeln, wobei die Ersetzungen der TGG-Regeln angewendet werden und die Korrespondenz überprüft und mitgeschrieben wird. Im Gegensatz dazu können sowohl Quell- als auch Zielmodell ggf. bereits vorliegen. Die intuitive Vorstellung eines Ersetzungsverfahren, das die ensprechenden linken Seiten aufsucht und mit den rechten Seiten der Ersetzungsregeln vertauscht, muss schon deshalb scheitern, weil für die Überprüfung der Graphenisomorphie der Teilgraphen kein effizienter Algorithmus existiert[7].

Da die textuelle Beschreibung von TGG-Regeln nicht besonders intuitiv ist, hat sich für die Beschreibung von bidirektionalen Modelltransformationen mit Hilfe von TGGs ein grafisches Verfahren eingebürgert, welches TGG-Regeln in einem neuen, gemeinsamen Modell darstellt, bei dem das Quell-, das Ziel- sowie das Korrespondenz-Metamodell enthalten und mit besonderen Stereotypen versehen ist:

- `«left»` an Modellelementen, die an der linken Ersetzungsregel teilnehmen
- `«right»` an Modellelementen, die an der rechten Ersetzungsregel teilnehmen
- `«create»` an Modellelementen (Klassen und Assoziationen), die bei Anwendung einer der Ersetzungsregeln erzeugt werden
- `«destroy»` an Modellelementen, die bei Anwendung einer der Ersetzungsregeln entfernt werden. Es gilt in vielen Verfahren jedoch die Einschränkung auf *monotone*, also vergrößernde, Produktionen, die i.A. keine Modellelemente löschen.
- `«map»` an Modellelementen des Korrespondenzmetamodells,

Dabei ist es natürlich zulässig, die Stereotypen zu kombinieren. In Abbildung 2 findet sich ein Beispiel, wie eine TGG-Regel zwischen einem Automatenmodell und dem Modell einer Steuerungssprache mit Hilfe der Stereotypen dargestellt werden (vgl. detaillierter dazu die Besprechung der Fallstudie in Abschnitt 5). Mittig finden sich die Elemente des Korrespondenzmetamodells mit dem Suffix `-'Corr'`.

[5] daher wohl der Name *Triple* Graph Grammars

[6] ab hier wird von *Modellen*, nicht mehr von (typisierten) *Graphen* die Rede sein

[7] Das Problem, ob zwei Graphen isomorph sind, liegt in der Klasse NP. Das Teilgraphisomorphieproblem ist sogar NP-vollständig. Vgl. dazu z.B. [5]

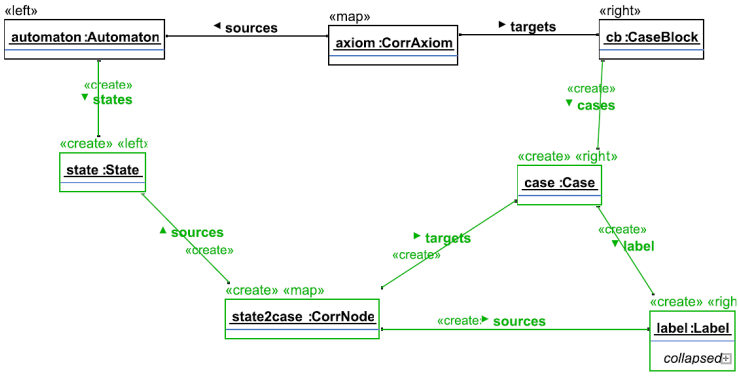

Abbildung 2. Formulierung einer einfachen TGG-Regel für die Übertragung eines Endlichen Automaten in eine Steuerungssprache (aus [2])

4 Verifikation

Verifikation von Programmen wird trotz guter Erfahrungen in sicherheitskritischen Bereichen wie Raumfahrt oder Medizintechnik und im Übersetzerbau immer noch nicht verbreitet in der Softwareentwicklung eingesetzt. Verschiedener Faktoren (u.a. fehlende formale Spezifikation, kurze Entwicklungszyklen, ...) lassen oftmals den grundsätzlichen Wunsch nach einem fehlerfreien Produkt zu 'relativer Fehlerfreiheit' degradieren - also dem Einstellen der Fehlersuche, wenn zuletzt hinreichend wenige Fehler gefunden wurden.

In der modellgetriebenen Softwareentwicklung, als relativ jungem Feld der ingenieursmäßigen Softwareentwicklung, könnte das Vorhandensein von semiautomatischen oder generischen Verfahren zur Verifikation von Modelltransformationen ein neues Kapitel aufschlagen. Mit dem Aufstieg von Programmcode zu abstrakteren Modellen besteht die Chance, einmal modellierte und im Modell formal spezifizierte Eigenschaften über Modelltransformationen hinweg zu erhalten und nicht durch Plattform-spezifische Implementierungen oder Änderungen am Code wieder zu verlieren.

4.1 Isabelle/HOL

Automatische Beweiser (engl. *Theorem Prover*) stellen für automatisierte Korrektheitsbeweise ein mächtiges Hilfmittel dar. So wird z.B. im Übersetzerbau davon Gebrauch gemacht. Unter vielen verfügbaren Beweisern ist *Isabelle/HOL* ein weit verbreitetes interaktives Beweiser-Framework, das an der University of Cambridge und der TU München entwickelt wird. In seiner Diplomarbeit verwendet Leitner [2] diese Software für die Korrektheitsbeweise in der Fallstudie und stellt anhand der Syntax der zu Grunde liegenden HOL[8] die generische For-

[8] HOL: higher-order logic

malisierung der Modelltransformation dar. In Tabelle 1 ist die Übersicht über das Isabelle/HOL-Framework dargestellt.

Isabelle/HOL	Isabelle-Instanz für HOL
Isabelle	Generischer Beweiser
StandardML (SML)	Modulare, Funktionale Implementierungssprache

Tabelle 1. Systemarchitektur von Isabelle und Isabelle/HOL (nach [7])

4.2 Übersicht Beweisverfahren

Die generelle Idee des Formalisierungs- und Beweisverfahrens ist die folgende:

> **Eine Modelltransformation ist dann korrekt, wenn zwei Modelle, die *vor* Anwendung einer TGG-Regel äquivalent waren, auch *nach* Anwendung der Regel äquivalent sind.**

Der zentrale Punkt ist dabei die zu definierende Äquivalenzrelation auf der Semantik von Modellen. Was heißt es, dass zwei Modellinstanzen semantisch äquivalent sind? Diese und weitere Fragestellungen, die sich bei einem allgemeinen Korrektheitsbeweis ergeben, sind in folgendem Schema zusammen zu fassen:

1. Syntaxdefinition: Umsetzung des Metamodells in einen induktiven Isabelle/HOL-Datentyp
2. TGG-Regeln: Überführung der Graphersetzungsregeln in sog. Modifikatoren
3. Korrespondenzen: Identifizierung und Zusammenfügen von Korrespondenzvoraussetzungen
4. Semantik: Festlegung einer formalen Semantik und einer Äquivalenzrelation darauf

Im Folgenden werden die einzelnen Schritte erläutert:

4.3 Syntaxdefinition

Zunächst müssen die im Metamodell repräsentierten Elemente in einen Datentypen aus Isabelle/HOL übersetzt werden. Die zur Verfügung stehenden Konstrukte sind:

- Datentypen (Basisdatentypen, Typvariablen, Totale Funktionen, Listen, benutzerdefinierte Datentypen)
- Rekursive Funktionen
- Logische Operatoren

Metamodelle werden auf Typen abgebildet und Modelle auf Instanzen dieses Typs. Für primitive Datentypen im Metamodell (Integer etc.) ist dies trivial, allerdings erfordern komplexe Datentypen einige Überlegung. Es müssen benutzerdefinierte Typen (`records`) angelegt werden, die dem Metamodell entsprechen. Außerdem sind Zyklen im Metamodell zu eliminieren, damit der induktive HOL-Datentyp als Liste von Konstruktoraufrufen erstellt werden kann. Folgendes generische Übertragungsschema wird verwendet:

- **Klassen** werden auf `records`abgebildet
- **Kollektionen** werden auf Listen abgebildet. Ggf. muss im Metamodell eine Ordnung eingeführt werden
- **Vererbung** wird durch das `datatype` Konstrukt (also einen abstrakten Datentypen) in HOL realisiert. Ein abstrakter Supertyp repräsentiert die Auswahl der konkreten `records`.

4.4 Modifikatoren

Den TGG-Regeln, genauer den Ersetzungsregeln für die beiden Modelle, werden in Isabelle/HOL Funktionen (sog. Modifikatoren) gegenübergestellt. Änderungen an einem Modell entsprechen also Anwendungen von Funktionen auf Elementen des entsprechenden HOL-Datentyps.

4.5 Korrespondenzen

Wie im Abschnitt 3 beschrieben, sind die Änderungen am Quell- und Zielmodell durch das Korrespondenzmodell verbunden. Eine TGG-Regel wird nur angewendet, wenn der entsprechende Zusammenhang zwischen dem Quell-, dem Korrespondenz- und dem Zielmodell besteht. Für die Formalisierung bedeutet dies: Die Korrespondenzen sind Vorbedingungen für die Korrektheits-Lemmata der entsprechenden Regeln. In Isabelle/HOL lässt sich dies durch ein Prädikat darstellen, dessen Inhalt jedoch von der Formalisierung des Metamodells abhängt.

4.6 Semantikdefinition

Wie jede Programmverifikation basiert auch die Verifikation von Modelltransformationen auf der formalen Definition der Semantik von Ein- und Ausgabe. Die verwendete *Operationale Semantik* fasst ein Modell als Menge von Eigenschaften auf. Belegungen mit Werten führen zu sog. Konfigurationen, die ein Modell durch Erzeugung und Änderungen durchläuft. Sieht man Änderungen im Modell durch eine Zustandsübergangsfunktion s beschrieben, so erhält man die formale Semantik des Modells als Ergebnis des wiederholten, rekursiven Aufrufs der Funktion s auf einem definierten Startzustand.

Es wird sofort klar, dass zwei Modelle, die bis auf Benennung identisch sind, nach dieser Semantikdefinition nicht äquivalent sind, wenn man als Kriterium die

(konfigurationsweise) Gleichheit fordert. Entsprechend müssen je nach Anwendungsfall geeignete Abstraktionen vorgenommen werden. Die erfordert jedoch Spezialistenwissen und ist nicht generisch.

Ein zentraler Begriff ist dabei die *Bisimulation*, welcher mit der Vorstellung der gleichzeitigen Ausführung einer Regel auf beiden Seiten einer Regel zu tun hat. Sind die Konfigurationen Zustände in einem Zustandsübergangssystem $(S, \rightarrow)$ mit Zustandsmenge S und $\rightarrow \in S \times S$ als Übergangsrelation.

Bisimulation. Eine Relation $R \in S \times S$ heißt *Bisimulation* gdw. $\forall p, q \in S$ mit $(p, q) \in R$ gilt: $\forall p' \in S : (p \rightarrow p' \Rightarrow \exists q' \in S : q \rightarrow q'$ und $(p', q') \in R)$ $\forall q' \in S : (q \rightarrow q' \Rightarrow \exists p' \in S : p \rightarrow p'$ und $(p', q') \in R)$

Hiermit lässt sich die folgende (semantische) Äquivalenzrelation definieren: Zwei Zustände $r, s \in S$ sind semantisch äquivalent, $p \approx q$, wenn es eine Bisimulation R gibt mit $(p, q) \in R$. Wenn im Folgenden von Äquivalenz die Rede ist, ist diese Definition gemeint.

4.7 Beweisdarstellung

Benennt man die Modifikatoren für die beiden Metamodelle M und N als o_M und o_N, sowie die Korrespondenzvoraussetzungen mit $C_1, ..., C_k$, zusätzliche Prädikate für die Regel, die nicht in der grafischen Darstellung der TGG enthalten sind, mit $P_1, ..., P_l$, sowie die Semantik eines Modells m im Metamodell M mit $sem_M(m)$, so erhält man insgesamt für jede Regel einen Satz der Form:

$$\bigwedge_{i=1}^{k} C_i \wedge \bigwedge_{j=1}^{l} P_j \wedge sem_M(m) \approx sem_N(n) \Rightarrow sem_M(o_M(m)) \approx sem_N(o_N(n)) \quad (1)$$

Dieser Satz muss mit Isabelle/HOL für jede TGG-Regel bewiesen werden.

5 Fallstudie

Einen wesentlichen Teil in der Diplomarbeit von Leitner [2] stellt die Anwendung des vorgestellten generischen Verfahrens zur Transformationsverifikation auf ein konkretes Bespiel nicht-trivialer Größe. Es handelt sich um die Überführung eines deterministischen endlichen Automaten in eine Steuerungssprache in strukturiertem Text [9] für die Ansteuerung von PLCs[10]. Diese Controller steuern ein Schienensystem und damit einen sicherheitskritischen Bereich. Daher lässt sich die Notwendigkeit einer Verifikation der Modelltransformation gut begründen.

Anhand der Überführung des Automatenmodells in ein Modell der Steuerungssprache (mit elementaren Konstrukten wie `if-then-else`, `case`, `statement`,

[9] SCL: Structured Control Language
[10] PLC: programmable logic controller

`function`) wird erläutert, wo die Herausforderungen und Möglichkeiten eines solchen Ansatzes bestehen. Insgesamt umfasst die Formalisierung der zum Beweis nötigen Konstrukte (Datentypen, Funktionen, Lemmata) ca. 1500 Zeilen, obwohl lediglich vier Regeln zu beweisen sind. Zusammengefasst stellten sich als besonders kritisch die folgenden Bereiche heraus:

- **Kongruenzbedingungen**, also der Erhalt der semantischen Äquivalenz bei Anwendung einer Regel, sind nur für triviale Regeln schlicht. Komplexe Regeln sind in der Formulierung schnell sehr groß
- **Korrespondenzen** wurden nicht explizit ausgedrückt, nur teilweise in die Modifikatoren eingebaut
- **Datentypen** konnten nur nach Eingriff in das Metamodell realisiert werden

Zumindest an der letzten Stelle könnte eine Semiautomatisierung helfen. Die Abbildung ist an den meisten Stellen generisch und die Formalisierung der Datentypen stellt fast ein Drittel des Beweiscodes dar. Die übrigen Punkte erfordern meist konkretes Expertenwissen, ohne das eine Übertragung in Beweiscode nicht möglich ist.

6 Zusammenfassung

Insgesamt wird in der besprochenen Arbeit ein grundlegender Schritt in Richtung Verifikation von Modelltransformationen gemacht. Einige der Ansätze zeigen, dass mit entsprechenden Hilfsmitteln eine deutliche Erleichterung bei der Erstellung des Beweisercodes erzielbar wäre. Das besprochene Anwendungsproblem, das insbesondere durch ein sehr einfaches Quellmetamodell als mittel-komplex einzuordnen ist, zeigt, dass leider beim Verifizieren von aufwändigeren Transformationen (bsp. das gut bekannte UML-RDBMS-Mapping[11]) mit erheblich umfangreicherem Beweisercode zu rechnen ist.

Zuletzt bleibt die Hoffnung, dass mit weiterem Fortschreiten der Modellorientierung in der Softwaretechnik auch das formale Spezifizieren der Semantik solcher Modelle gefördert wird. Dies hätte einen erheblichen Einfluss auf die Durchführbarkeit von Verifikationen von Modelltransformationen (ob in TGGs oder QVT formuliert).

Literatur

1. Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards Verified Model Transformations. Proc. of the 3rd Workshop on Model design and Validation (MoDeV2a), Genua, Italien (2006)
2. Leitner, Johannes: Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken. Diplomarbeit, Karlsruhe/Berlin, Germany (2006)
3. Stahl, Th., Völter, M.: Modellgetriebene Softwareentwicklung. dpunkt.verlag, Heidelberg, Germany (2005)

[11] Überführung eines UML-Modells in ein Schema einer relationalen Datenbank

4. Schürr, Andy: Specification of Graph Translators with Triple Graph Grammars. Lecture Notes in Computer Science (LNCS), Vol. 903, Springer Verlag, Heidelberg, Germany (1994)
5. Krumke, S., Noltemeier, H.: Graphentheoretische Konzepte und Algorithmen. Teubner Verlag, Wiesbaden, Germany (2005)
6. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electronic Notes in Theoretical Computer Science (113-150), Elsevier (2006)
7. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science (LNCS), Vol. 2283, Springer Verlag, Heidelberg, Germany (2002)