Modelltransformationen mit Tripel-Graph-Grammatiken

Selvinaz Karahancer
252 132

Betreut von Dipl.-Inform. Anne-Thérèse Körtgen

Zusammenfassung

Modelltransformationen spielen in der modellbasierten Softwareentwicklung eine besondere Rolle. Sie sollen zunehmend automatisch durchgeführt werden, um Modellintegrationsaufgaben wie Konsistenzüberprüfung, Konsistenzherstellung und Modelltransformation zu erfüllen. Diese Arbeit stellt den Modelltransformationsansatz nach Königs vor. In dem Ansatz werden graphbasierte Modelle betrachtet und Transformationen durch Tripel-Graph-Grammatiken definiert. Die Regeln einer Tripel-Graph-Grammatik sind synchron. Das bedeutet, dass sie nur beschreiben, welche Knotentypen einander entsprechen. Diese synchronen Regeln werden automatisch in asynchrone Graph-Grammatikregeln transformiert, die die Transformation von existierenden Graphen in neue Graphen ermöglichen.

Inhaltsverzeichnis

1 Einleitung

Das Hauptthema des Seminars lautet „Unterstützung modellgetriebener Softwareentwicklung“. Transformationen zwischen Modellen spielen eine bedeutende Rolle bei der modellgetriebenen Softwareentwicklung wie beispielsweise bei der Modellintegration. Bei dem Transformationsprozess eines Quellmodells in ein Zielmodell ist es wichtig, dass Transformationen in beide Richtungen ermöglicht werden, um die Transformation zurückzuverfolgen, und inkrementelle Änderungen auf beiden Seiten vorgenommen werden können. Ein Ansatz zur Modelltransformation sind graphbasierte Transformationen. Für graphbasierte Transformationen werden die Tripel-Graph-Grammatiken verwendet, die im Folgenden näher beschrieben werden.

1.1 Der traditionelle Softwareentwicklungsprozess

Der traditionelle Softwareentwicklungsprozess kann mithilfe des Wasserfall- oder Spiralmodells modelliert werden [4]. Bei diesem Entwicklungsprozess können aber verschiedene Probleme auftreten. Das erste erwähnenswerte Problem ist die *Produktivität*. Bei der Softwareentwicklung wird eine Vielzahl von Dokumenten in Papierform produziert. Dadurch sind der Inhalt und die Konsistenz wegen Zeitdruck nicht mehr prüfbar. Außerdem wird der Abstand zwischen Code und Entwurf von Iteration zu Iteration größer. Dies wird oft von Zeitproblemen getrieben. Das zweite wichtige Problem beim Entwicklungsprozess ist die *Portabilität*. Da die Softwareindustrie ständig neuere, bessere und schnellere Technologien bringt, müssen auch die Softwarehersteller diesen Trends folgen: erstens, weil der Kunde es so will, zweitens, weil Probleme dadurch gelöst werden und zuletzt, weil alte Technologien nicht mehr unterstützt werden. Als Ergebnis muss die Software ständig portiert werden, wobei die eigentliche Funktion nicht geändert wird.
Die *Interoperabilität* ist ein weiteres wichtiges Problem in der Softwareentwicklung. Interoperabilität bedeutet, dass Systeme nur ganz selten allein sind, in der Regel interagieren sie mit anderen existierenden Systemen. Ein System besteht aus vielen Einzelteilen und im Ergebnis müssen alle Einzelsysteme über Technologie hinweg zusammen funktionieren.
Das *Wartungs- und Dokumentationsproblem* ist das letzte erwähnenswerte Problem. Bei der Erstellung der Software glauben die Entwickler, dass es ihre Aufgabe ist, Code zu produzieren. Jedoch bremst die Dokumentation diesen Prozess. Die Softwaresysteme sind folglich schlecht dokumentiert und damit nicht wartbar.

1.2 Lösungsansätze

Modelltransformationen bilden auch die Basis der Model Driven Architecture (MDA) [3]. Die MDA ist eine Initiative der OMG zur modellbasierten Entwicklung von Software-Systemen. Sie bietet einen Entwicklungsansatz, in welchen aus allgemeinen, plattformunabhängigen Modellen (Platform Independent Model, PIM) spezifischere Modelle für eine Plattform (Platform Specific Model, PSM) und letztendlich Code generiert werden. Mit der MDA wird das Modell zum zentralen Element des Softwareentwicklungsprozesses. Neben der inhaltlichen Trennung von Modellen auf unterschiedlichen Abstraktionsebenen definiert die MDA außerdem auch die Transformation der Modelle. MDA basiert auf der Idee der automatischen Transformation von Modellen von der abstrakten Ebene in die konkretere Ebene [7]. Die Transformationen erzeugen aus den Elementen des Quellmodells die Elemente des Zielmodells. MDA bringt also einen neuen Entwicklungszyklus und eine Automatisierung der Entwicklungsschritte. Daher spiegeln die Vorteile von MDA genau die Nachteile der traditionellen Softwareentwicklung (Produktivität, Portabilität, Interoperabilität, Wartung- und Dokumentation). MDA soll die Portabilität und Wiederverwendung von Modellen im Entwicklungsprozess und dadurch die Software selbst verbessern. Damit wird eine Beschleunigung und Kostensenkung in der Entwicklung von Software erhofft. Neben der Spezifikation und Dokumentation der Software werden formale Modelle nun auch zur Definition der Architektur, des Entwurfs und der Implementierung genutzt. Die automatisierte Transformation von Modellen erspart zusätzlichen Entwicklungsaufwand und hilft die Konsistenz der Modelle untereinander zu sichern. Die Transformation zwischen den Modellen wird in den Transformationsregeln festgelegt.
Ein Ansatz verwendet zur Modelltransformation graphbasierte Transformationen. Für graphbasierte Transformationen hat Königs den Ansatz von Tripel-Graph-Grammatiken [7] verwendet. Bei diesem Ansatz werden getrennte Graphgrammatiken zur Definition von zwei Graphklassen verwendet, für deren Regeln jedoch eine paarweise Zuordnung existiert. Der Bezug zwischen den beiden Grammatiken wird durch eine dritte Grammatik hergestellt, in welcher der Zusammenhang von Elementen eines Regelpaars festgelegt wird. Somit erlauben es Tripel-Graph-Grammatiken, Abbildungen zwischen zwei Graphen in beide Richtungen zu interpretieren. Da die Transformationsregeln der Grammatik jedoch in der synchronen Beschreibung nicht sehr sinnvoll und praxisrelevant sind, werden diese Regeln automatisch in asynchrone Regeln umgesetzt, um Modellintegrationsaufgaben wie Konsistenzüberprüfung-, herstellung und die bidirektionale Modelltransformation zu erfüllen.
Für automatisierte Transformationen zwischen MOF (Meta-Object Facility)- Mo-

dellen legt die MDA QVT nahe. QVT steht für Query/Views/Transformation und ist ein OMG Standard. Mit QVT lassen sich Modelle ineinander überführen, deren Metamodelle durch die MOF spezifiziert wurden. Dabei besteht eine QVT-Spezifikation aus den deklarativen Sprachen *QVT-Core* und *QVT-Relations*, während der imperative Teil der Spezifikation die Sprache *Operational Mappings Language* und Vorgaben für *Black Box Implementations* enthält.

1.3 Aufbau der Arbeit

In dieser Seminararbeit wird zunächst in Kapitel 2 eine motivierende Einführung in das Teilgebiet der Softwaretechnik gegeben. Dazu wird erläutert, was Modelltransformation bedeutet und welche Vorteile sie gewährleistet. Zum besseren Verständnis der Tripel-Graph-Grammatiken und deren Funktion bei Modelltransformationen werden in Kapitel 3 die Pair-Graph-Grammatiken eingeführt (PGG), die Graph-zu-Graph Transformationen ermöglichen. Diese Grammatiken werden erweitert um einen dritten Korrespondenzgraphen, so dass die Tripel-Graph-Grammatiken entstehen. Nach einer Beschreibung der Tripel-Graph-Grammatiken wird auf den Aufbau der Tripel-Graph-Grammatiken eingegangen. Daraufhin wird in Kapitel 4 eine Beispieltransformation von ER-Modellen in relationale Schemata repräsentiert. In Kapitel 5 werden verwandte Ansätze zu Tripel-Graph-Grammatiken kurz vorgestellt und auf die Unterschiede zwischen den beiden Modelltransformationssprachen QVT und Tripel-Graph-Grammatiken hingewiesen. Daraufhin werden die Bewertung und Ausblicke von Modelltransformationen mit Tripel-Graph-Grammatiken in Kapitel 6 dargestellt. Zum Abschluss wird die Arbeit in Kapitel 7 zusammengefasst und die Vorteile von Tripel-Graph-Grammatiken vorgestellt.

2 Modelltransformationen

Der Einsatz von Modellen zur Beschreibung von Softwaresystemen ermöglicht es von unwesentlichen Details zu abstrahieren und sich auf wesentliche Aspekte des zu erstellenden Systems zu konzentrieren. Dabei repräsentieren Modelle einen Teil der Funktionalität, Struktur oder des Verhaltens eines Softwaresystems und seiner Umgebung [4]. Sie sind formal definiert, das heißt, dass die jeweilige verwendete Modellierungssprache eine eindeutig definierte Syntax und Semantik hat. Eine Modellierungssprache wird durch ein *Metamodell* definiert. Ein Metamodell [8] ist ein Modell, das die Elemente des Modells beschreibt. Metamodelle und Modelle stehen in einer Instanzierungsbeziehung zueinander. Ein Metamodell

beschreibt die konkrete und abstrakte Syntax und der Semantik einer Sprache. Die *abstrakte Syntax* einer Modellierungssprache legt den Aufbau der beschreibbaren Modelle fest. Dabei abstrahiert sie in der Regel von Details und Informationen, die für die Semantik einer Sprache irrelevant sind. Die abstrakte Syntax definiert jedoch nicht, wie sie dem Endbenutzer präsentiert wird. Die äußere Darstellungsform wird durch die *konkrete Syntax* erreicht, deren Notation textuell oder graphisch ist. Die konkrete Syntax definiert den Aufbau einzelner Sprachkonstrukte und deren Kombination wie zum Beispiel Ausdrücke und Anweisungen.
Das wichtigste Artefakt für die modellgetriebene Softwareentwicklung, die auch das Hauptthema des Seminars ist, sind Modelltransformationen. Eine Modelltransformation transformiert ein Quellmodell auf der linken Seite nach bestimmten festgelegten Vorgaben in ein Zielmodell auf der rechten Seite. Dies kann auf verschiedene Arten passieren:

1. Die Modelltransformation kann mit Hilfe eines Algorithmus durchgeführt werden oder mit einer Metamodelltransformation. Das bedeutet, dass die abstrakte Syntax der Quellmodellsprache in die abstrakte Syntax der Zielmodellsprache transformiert wird.

2. Desweiteren besteht die Möglichkeit, die Transformation zwischen Modellen mit Hilfe von Transformationsregeln zu vollziehen. Diese Transformationsregeln legen fest, welche Elemente des Quellmodells in welche Elemente des Zielmodells transformiert werden.

Wie bereits schon erwähnt, beschreibt der Ansatz von Königs, die Tripel-Graph-Grammatiken, Transformationsregeln für Modelle auf der Grundlage von Graphtransformationen. Die Vorteile dieses Ansatzes sind:

- Graphen sind eine natürliche und sinnvolle Repräsentation von Modellen (vergleiche visuelle Sprachen: Visuelle Sprachen bieten die Möglichkeit, Modelle „visuell“ darzustellen, zum Beispiel mit einem Graphen. Diese visuellen Repräsentationen können besser verarbeitet und verstanden werden als Texte, da diese die Zusammenhänge zwischen Komponenten besser darstellen als textuelle Beschreibungen. [11])

- Graphersetzungsregeln können selbst wieder als Graphen dargestellt werden, wodurch sie selbst wieder verständlicher sind.

3 Tripel-Graph-Grammatiken

In diesem Kapitel werden zunächst die Grundlagen für die Tripel-Graph- Grammatiken erläutert. In 3.1 wird vorerst beschrieben, was eine Graph-Grammatik ist, in 3.2 werden die Pair-Graph-Grammatiken eingeführt und ihre Erweiterung zu Tripel-Graph-Grammatiken wird in 3.3 vorgestellt.

3.1 Graph-Grammatiken

Ein *Graph* G besteht aus einer Menge von Knoten V und Kanten E und aus den Funktionen $s, t : E \rightarrow V$, die den Kanten Quell- und Zielknoten zuordnen. Zudem ist G ein markierter Graph, wenn den Knoten und Kanten Markierungen zugeordnet werden. Eine *Graph-Grammatik* besteht aus einer Menge von Graphproduktionen und beschreibt, wie ein gültiger Graph für diese Grammatik erstellt werden kann. Die linke und rechte Seite einer solchen Produktion stellt jeweils ein Graphmuster, L und R, dar. Bei der Anwendung einer solchen Produktion wird auf einem Instanzgraphen G zunächst eine passende Anwendungsstelle im Graphen gesucht, d.h. einen Untergraph G' von G, so dass es einen Isomorphismus von L nach G' gibt. Sei dazu $G' = (V', E')$. Gesucht wird eine Abbildung $f : V_L \longrightarrow V'$, so dass $(u, v) \in E_L$, wenn $(f(u), f(v)) \in E'$ gilt. Bei der Suche nach einem Graphmuster existieren also mehrere Parameter, die übereinstimmen müssen. Zunächst muss das gefundene Objekt vom gleichen Typ sein wie das entsprechende Objekt des Graphmusters. Auch die über Korrespondenzverbindungen erreichbaren Nachbarobjekte müssen den Nachbarobjekten des Musters entsprechen. Für jedes Objekt können zusätzlich noch bestimmte Attributwerte festgelegt sein, die ebenfalls übereinstimmen müssen. Diese Suche nach einer Übereinstimmung zwischen zwei Teilgraphen wird auch als *Graph-Matching* bezeichnet. War das Matching erfolgreich, wird der Untergraph G' dann durch das Graphmuster R ersetzt. Mit anderen Worten kann eine Produktionsregel angewendet werden, falls im Graphen ein Muster mit derselben Struktur wie der Graph der linken Regelseite gefunden wird. Dieser Untergraph des ursprünglichen Graphen wird durch den Graphen der rechten Regelseite ersetzt. Zur Verknüpfung der rechten Seite mit dem Restgraphen, wird zusätzlich noch eine Einbettungsvorschrift gebraucht, die jeweils die Kantenenden des alten Teilgraphen mit denen des neuen identifiziert. Generell können die Regeln einer Graph-Grammatik in beliebiger Reihenfolge ausgeführt werden, vorausgesetzt es findet sich die von einer Regel jeweils geforderte Struktur im Ursprungsgraphen. Durch diese Ersetzung stellt eine solche Paarregel nichts weiter als eine *Graphersetzungsregel* dar. Oft werden daher auch die Begriffe *Graph-Grammatik* und *Graphersetzungsregel* synonym gebraucht.

3.2 Graphersetzungsregeln mit Pair-Graph-Grammatiken

Pair-Graph-Grammatiken aus [10] stellen eine spezielle Form von Graph- Grammatiken dar. Regeln einer solchen Paargrammatik bestehen aus einem Paar von zwei Regeln und einer Relation. In Abbildung 1 ist eine solche Regel dargestellt. Paarregeln bestehen aus zwei Graphproduktionen S und T, die jeweils auf zwei getrennten Graphen parallel arbeiten und einer Relation m. Die Relation m ist zuständig für die Zuordnung der Knoten aus der einen Produktion zu Knoten der anderen. Dabei ist es möglich, eine $(1:1)$- oder $(1:n)$- Beziehungen zwischen den Knoten der Produktionen S und T darzustellen. In der Abbildung ordnet die Relation m Knoten a aus der Produktion S zu Knoten b der Produktion T zu. Sie bilden den Tupel $((S,T),m)$. Die Zuordnung des Knotens a zu Knoten b bildet die linke Seite L der Paarregel. Nach der Anwendung dieser Paaregel entstehen durch die Ersetzung der linken Regelseite durch die rechte Regelseite zwei neue Knoten c und d, die einander korrespondieren und die zu den Kontextknoten a aus der Produktion S und zu Knoten b der Produktion T in Beziehung stehen. Bei der Anwendung der Paarregel wird in der Regel eine *Ableitungsstruktur* bzw. ein *Ableitungsgraph* [9] benötigt, der bei der Ersetzung der linken Regelseite durch die rechte Regelseite anzeigt, welche Knoten der linken Seite durch welche Knoten der rechten Seite ersetzt werden. In diesem Ableitungsgraphen werden alle Ersetzungen und Transformationsschritte gespeichert.
Eine Paarregel kann zweierlei gelesen werden: Entweder definieren Paarregeln Paare von Modellen, die zusammen gehören. In diesem Fall wird eine Relation zwischen den beiden Modellen definiert. Bei der Ausführung der Paarregel wird die linke Seite der Regel in dem gegebenen Graphen gematcht. Dabei werden die Instanzen beider Graphen gleichzeitig erzeugt. Eine Paarregel kann aber auch als eine Transformation gelesen werden. Das bedeutet, dass jede Regel einer Paargrammatik eine Transformationsregel angibt, die für die linke Seite S die rechte Seite T erzeugt oder umgekehrt. Das heißt, dass die Produktion S bereits ausgeführt wurde, so dass der Quellgraph SG entstanden ist. Nach dem erfolgreichen Matching des Startkontextes und der rechten Seite der Produktion S im gegebenen Graphen wird der Transformationsregel angewendet und der korrespondierende linke Graph TG wird erzeugt.

3.3 Tripel-Graph-Grammatiken

Die Pair-Graph-Grammatiken werden zu Tripel-Graph-Grammatiken erweitert, indem die Paarregeln nicht mehr mit einer einfachen Relation m ausgestattet sind,

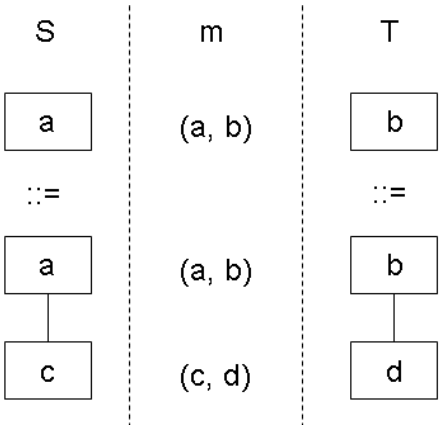

Abbildung 1: Eine PGG-Regel

die die Knoten der einen Produktion denen der anderen zuordnet, sondern mit einer dritten Produktion, die parallel einen dritten Graphen erstellt. Die Erweiterung der Zusammenhänge zwischen den Quell- und Zielgraphen, SG und TG, zu einem eigenständigen dritten Korrespondenzgraphen CG hat den entscheidenden Vorteil, dass jetzt auch komplexe $(m : n)$- Beziehungen zwischen dem Quell- und Zielgraphen dargestellt werden können. Das bedeutet, dass ein Knoten im Korrespondenzgraphen CG mit mehreren Knoten in SG und TG verbunden werden kann. Dadurch erhalten die Tripel-Graph-Grammatiken eine höhere Ausdruckskraft als die PGGs.
Tripel-Graph-Grammatikregeln haben die Form (S, T, C, m_1, m_2), wobei S, T und C Graphproduktionen sind und m_1 eine Relation, die S mit C verknüpft und m_2 eine Relation, die T mit C verknüpft.
Abbildung 2 zeigt ein abstraktes Beispiel für eine Tripel-Graph-Grammatikregel. Die Produktionsregeln S und T für die Graphen SG und TG produzieren verschieden strukturierte Graphmuster. Die korrespondierenden Knoten in der Mitte stellen eine Relation zwischen ein oder mehreren Elementen auf beiden Seiten her.
Die linke Seite dieser Abbildung stellt den *Startkontext* für die Anwendung einer Grammatikregel dar. Der Knoten der linken Seite jeder Graphregel taucht auf der rechten Seite der Regel wieder auf.

Graph-Grammatiken, die keine Knoten in Produktionsregeln löschen, heißen „nicht-löschende“ Regeln. In dieser Arbeit werden nur diese Regeln betrachtet. Aus dieser Tatsache, dass nur noch nicht-löschende Transformationsregeln betrachtet werden, muss eine Ableitungsstruktur nicht explizit angegeben werden.

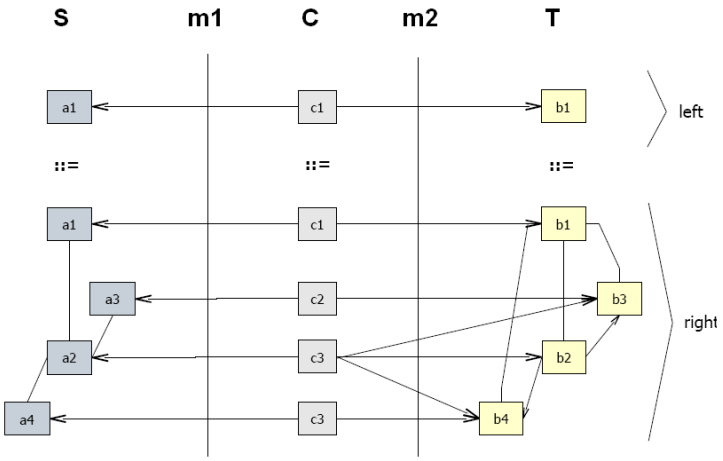

Abbildung 2: Produktionsregeln S, T und C einer Tripel-Graph-Grammatikregel und ihre Relationen m_1 und m_2

Löschende Grammatiken steigern die Komplexität der Kontrolle bzw. der Überprüfung, wie beispielsweise welche Folge von Regelanwendung eine gegebene Graphstruktur produziert. Neben dem Komplexitätsproblem macht das Erlauben von löschenden Grammatikregeln die resultierende Sprache einer Grammatik schwer voraussehbar. Allerdings können Modelltransformationen auch auf Basis von nicht-löschenden Regeln durchgeführt werden. Das erlaubt eine kompaktere Notation der oberen Regel in Abbildung 2:

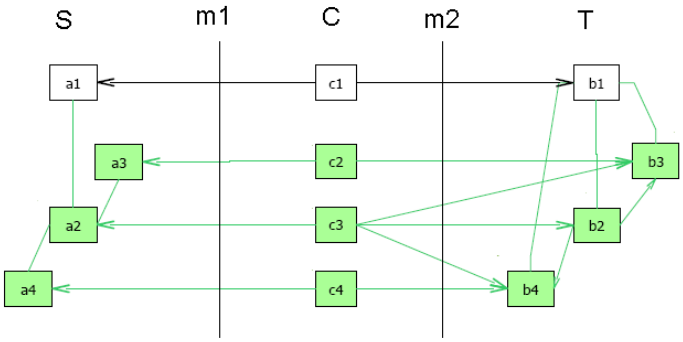

Abbildung 3: Tripel-Graph-Grammatikregel aus Abb.2 in kompakter Notation

In Abbildung 3 ist die Regel aus Abbildung 2 in der kompakten Notation dargestellt. Dort werden nur die Graphmuster der rechten Seite gezeigt und die Knoten

(die weißen), die auf der linken Seite auftreten, treten nun auch auf der rechten Seite auf. Die grünen Knoten repräsentieren die Knoten, die zusätzlich in der Regel entstanden sind. Die letzteren Knoten, wie a2, stehen zu den Kontextknoten, wie a1 (integrierte linke Seite), in Beziehung.
Zwei Graphen, die durch Anwendung von Tripel-Graph-Grammatikregeln als Produktionen parallel entstanden sind, müssen immer semantisch äquivalent sein, das heißt, sie müssen immer einander korrespondieren. Dies kann mit einem Prinzip bewiesen werden, dass dem der Induktion ähnelt: Sind zwei Graphen semantisch äquivalent, dann bleiben sie es auch, wenn auf beide die jeweilige Graphproduktion der gleichen Tripel-Graph-Grammatikregel angewendet wird.

3.3.1 Asynchrone Graphtransformationsregeln

Bisher wurde gezeigt, wie zwei Graphen und ihr Korrespondenzgraph parallel durch Anwendung von synchronen Tripel-Graph-Grammatikregeln entstehen. In modellbasierten Entwicklungsprozessen entstehen Modelle nicht parallel, sondern simultan aus anderen bereits bestehenden Modellen durch Modelltransformation. Die bisher synchrone Transformationsregeln werden automatisch in asynchrone Graphtransformationsregeln übersetzt. Bereits bei den Pair-Graph-Grammatiken wurde erwähnt, dass eine Paaregel als eine Transformation gelesen werden kann. Das bedeutet, dass für eine bestehende linke Seite die rechte Seite erzeugt wird oder umgekehrt. Durch diese asynchronen Transformationsregeln ist es möglich, Modellintegrationsaufgaben wie Konsistenzüberprüfung-, herstellung und die bidirektionale Modelltransformation zu erfüllen.
Die Idee dieses Übersetzungsprozesses ist wie folgt: Die rechte Seite R einer Graphproduktion (S oder T) wird in die linke Seite L der Produktion übernommen. Bei der Ausführung der Transformation wird zunächst der Startkontext im gegebenen Graphen gematcht. Anschließend wird die linke bzw. rechte Seite im Graphen gesucht, und nach dem erfolgreichen Matching werden die entsprechenden Graphmuster von Quell- und Zielgraphen erzeugt.
Im Folgenden werden die aus den synchronen Tripel-Graph-Grammatikregeln abgeleiteten Vorwärts- und Rückwärtsregeln und Konsistenzregeln vorgestellt.

Vorwärtsregel
Abbildung 4 zeigt die abgeleitete Vorwärtsregel aus der Regel aus Abbildung 3. In dieser Regel stellt der obere Teil den Startkontext für die Graphtransformation dar und dient als Kontext für die Anwendung der Regel. Bei der Vorwärtstransformation wird zu dem bereits existierenden Quellgraphen SG der korrespondierende

Zielgraph TG erzeugt. Zusätzlich wird auch der Korrespondenzgraph CG erstellt, der die beiden Graphen miteinander verbindet. Dazu wird bei der Vorwärtsregel die rechte Seite R der Graphproduktion *S* in die linke Seite L übernommen.

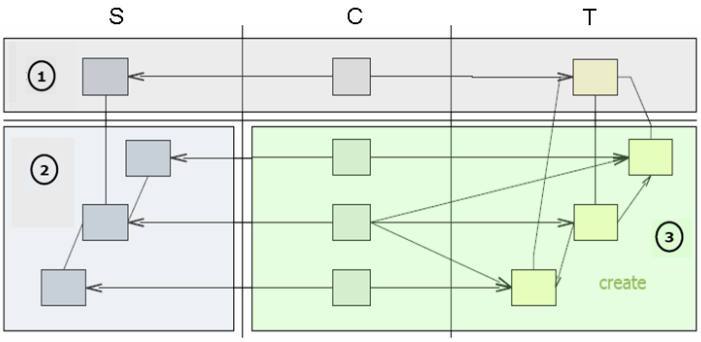

Abbildung 4: Vorwärtsregel von Tripel-Graph-Grammatikregel aus Abb. 3

Damit die Regel angewendet werden kann, muss der Ausgangsgraph zunächst gefunden werden. Das bedeutet, dass die Elemente des Quellgraphen schon existieren müssen. Die erfolgreiche Suche nach dem Ausgangsgraphen stellt die Bedingung zur Anwendung der Tripel-Graph-Grammatikregel dar. Wenn das Graph-Matching erfolgreich ist, werden die Objekte und Korrespondenzverbindungen des *Erweiterungsgraphen* erzeugt. Dies geschieht allerdings nur dann, wenn die entsprechende Graphproduktion im Ausgangsgraphen noch nicht ausgeführt wurde.
Zur genaueren Darstellung der Abläufe während einer Vorwärtstransformation dient die Abbildung 5. In dieser Abbildung wird ein Schritt der Regelanwendung dargestellt. Der Startkontext für die Graphtransformation ist gegeben und dient auch als Kontext für die Anwendung der Graph-Grammatikregel. In dieser Abbildung existiert eine Menge von Tripel-Graph-Grammatikregeln in ihrer kompakten Notation. Der komplette Quellgraph SG ist gegeben. Bei der Ausführung der Vorwärtsregel wird in Schritt ① die linke Regelseite im gegebenen Graphen gematcht. Anschließend wird in Schritt ② das Muster der linken Regelseite der Vorwärtsregel im Graph SG gematcht und folglich werden in Schritt ③ die Muster der Vorwärtsregel T und C im Graphen TG und im Korrespondenzgraphen CG erstellt. Dabei stellen die durch ① und ② markierten Felder und Korrespondenzverbindungen den Ausgangsgraphen und die durch den Stereotype «create» gekennzeichneten Objekte sowie Korrespondenzverbindungen den *Erweiterungsgraphen* dar. Wird also der Ausgangsgraph gefunden, so wird er um den *Erweiterungsgraphen* ergänzt. Der Vorgang des Matchings wird für alle

Muster im Quellgraphen wiederholt, bis der komplette Zielgraph und Korrespondenzgraph entstehen. An dieser asynchronen Vorwärtsregel wird deutlich, dass nicht der komplette Zielgraph TG generiert werden muss, sondern inkrementelle Änderungen an dem Quellgraphen in den Zielgraphen übernommen werden.

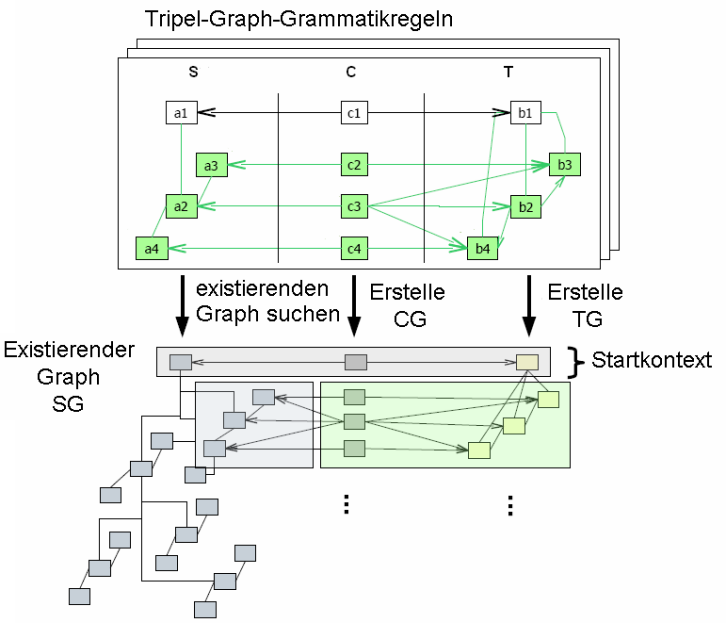

Abbildung 5: Abläufe bei der Ausführung der Vorwärtsregel aus Abbildung 4

Rückwärtsregel
Abbildung 6 zeigt die abgeleitete Rückwärtsregel aus der Regel aus Abbildung 3. Bei dieser abgeleiteten Rückwärtsregel wird die rechte Seite R der Graphproduktion T in die linke Seite L übernommen. In diesem Fall sind der Startkontext und der rechte Zielgraph TG bereits gegeben. Bei der Ausführung der Transformation werden in Schritten ① und ② der Startkontext und die Zielseite von Tripel-Graph-Grammatik gematcht und in Schritt ③ die Quell- und Korrespondenzmuster der Graphen SG und CG werden erstellt. Analog zur Vorwärtsregel wird der Vorgang des Matchings der Muster im Zielgraphen TG wiederholt durchgeführt bis der komplette Quellgraph SG entsteht.

Konsistenzregel
Abbildung 7 zeigt die abgeleitete Konsistenzregel aus Abbildung 3. Sie wird ein-

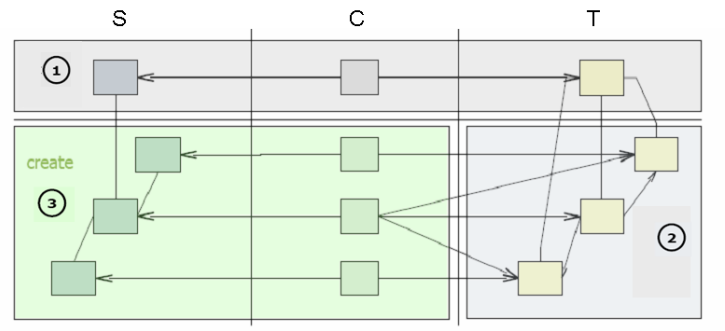

Abbildung 6: Rückwärtsregel von Tripel-Graph-Grammatikregel aus Abb. 3

gesetzt, wenn bereits beide Graphen, Quell- und Zielgraph, vorhanden sind und nur noch der Korrespondenzgraph erstellt werden muss. In diesem Szenario werden also die Tripel-Graph-Grammatikregeln angewendet, um eine gültige Korrespondenz zwischen zwei existierenden Graphen zu errechnen. Dazu werden die rechten Seiten R der Produktionen S und T in die linke Seite L übernommen. Bei der Ausführung der Regel werden in Schritt ① der Startkontext und in Schritt ② die Quell- und Zielseiten der Regeln gematcht und das Korrespondenzmuster des Graphen CG wird in Schritt ③ erstellt. Nachdem die Korrespondenzverbindungen zwischen den Knoten des Quell- und des Zielgraphen erstellt werden, kann überprüft werden, ob beide Graphen miteinander korrespondieren. Das bedeutet, dass es mithilfe der Konsistenzregel möglich ist, die Konsistenz zwischen zwei Graphen zu überprüfen.

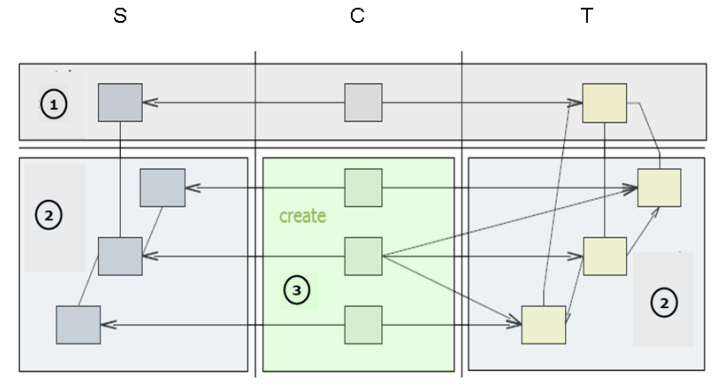

Abbildung 7: Konsistenzregel

4 Transformation von ER-Modellen in relationale Schemata

Für die Modellierung von relationalen Datenbankschemata werden häufig *Entity-Relationship-Modelle* (ER-Modelle) verwendet [6]. Dabei stellen *Entities* (Gegenstände) wohlunterscheidbare physisch oder gedanklich existierende Konzepte der zu modellierenden Welt dar. Entities haben *Attribute*, die dazu dienen, Gegenstände bzw. Beziehungen zu charakterisieren. Ähnliche Gegenstände werden zu Gegenstandstypen (Entitytypen) abstrahiert. Ein *Relationship* (Beziehung) zwischen den Entities kann als eine Relation im mathematischen Sinn betrachtet werden. Demnach ist eine Relation eine endliche Teilmenge vom Kreuzprodukt von Entitytypen, die an der Beziehung beteiligt sind.
Im Folgenden wird die Transformation von ER-Modellen in relationale Schemata mit Tripel-Graph-Grammatiken erklärt. Dazu wird in 4.1 die synchronen Regeln und in 4.2 die asynchronen Regeln vorgestellt.

4.1 Synchrone Regeln

Bei der Umsetzung eines ER-Modells in ein relationales Datenbank-Schema, welches aus Relationen besteht, werden folgende Transformationen durchgeführt:

- Entities werden in Tabellen bzw. Relationen umgewandelt.
- Attribute werden in Spalten transformiert.
- Attributwerte werden in Zeilen übertragen.

Transformation von Entities
Bei der Transformation von Entities wird zwischen persistenten und nicht-persistenten Entities unterschieden. Persistente Entities entsprechen den Tabellen des relationalen Schemas mit demselben Namen, während nicht-persistente Entities zu bereits bestehenden Tabellen korrespondieren. Das bedeutet, dass für sie keine neuen Tabellen erzeugt werden.
Abbildung 8 beschreibt eine Regel, die definiert, dass ein Entity e einer Tabelle t mit dem gleichen Namen entspricht. Eine Korrespondenzverbindung mit dem Typ EntityTabelleRel wird eingefügt, die das Entity e und die Tabelle t verbindet.

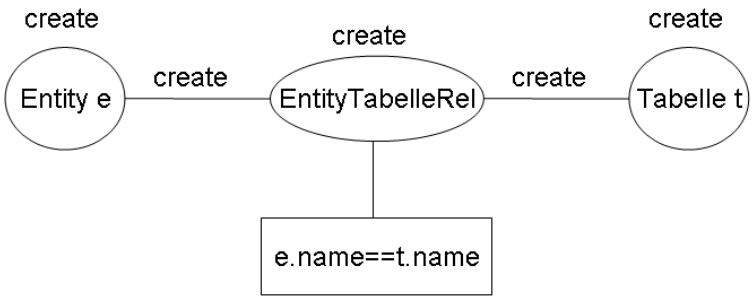

Abbildung 8: Tripel-Graph-Grammatikregel für Entity und Tabelle

Transformation von Attributen
Attribute von Entities mit primitiven Datentypen (z.B. Zeichenketten, Zahlen) entsprechen den Spalten der korrespondierenden Tabelle mit gleichem Datentyp.

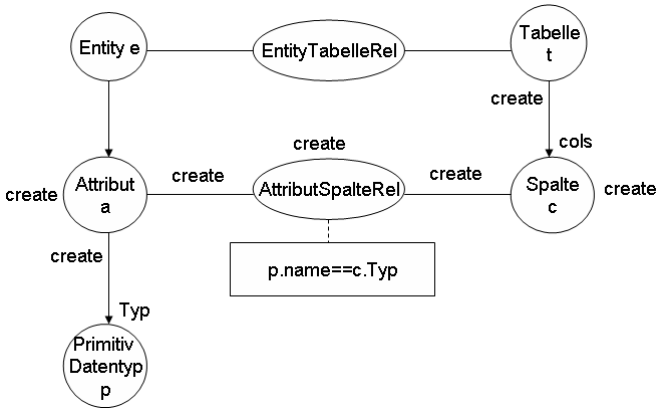

Abbildung 9: Transformation von Attributen mit primitiven Datentypen

Abbildung 9 zeigt eine Regel, die definiert, dass ein Attribut a des Entities e mit einem primitiven Datentyp p der Spalte c mit demselben Namen entspricht. Die bereits existierenden Knoten und schon durchgeführte Transformation vom Entity e in Tabelle t stellen den Startkontext für die parallele Erzeugung des Attributs und der Spalte dar. Die neue Spalte c wird der Tabelle t hinzugefügt, die dem Entity e entspricht, das das Attribut a besitzt.

Transformation von Relationen
Bei der Transformation von Relationen müssen zwei Fälle unterschieden werden: Zum einen kann eine Relation mit einem persistenten Entity verbunden

sein oder aber mit einem nicht-persistenten Entity. Abhängig davon wird bei der Transformation unterschiedlich vorgegangen: Relationen, deren Zielknoten nicht-persistente Entities sind, werden nicht explizit transformiert. Vielmehr entspricht das nicht-persistente Entity gerade der Tabelle, die mit dem persistenten Entity der Relation korrespondiert.
In Abbildung 10 entsprechen die Tabelle t und das Entity e1 einander und repräsentieren den Startkontext. Die neue Relation r zeigt auf ein nicht-persistentes Entity e2 (Zielknoten) mit der Kantenmarkierung *dest*. Dieses nicht-persistente Entity e2 wird mit der Tabelle t über die Korrespondenzverbindung EntityTabelleRel verbunden.

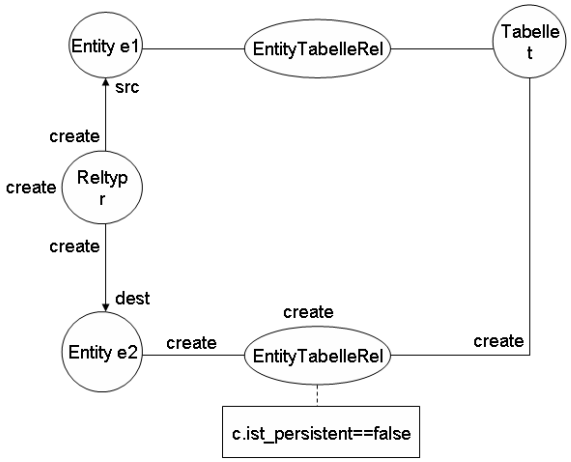

Abbildung 10: Transformation von Relationen mit nicht-persistentem Ziel-Entity

Bei Relationen, deren Zielknoten ein persistentes Entity ist, entsprechen beide persistente Entities schon zwei verschiedenen Tabellen. Die Relation r zwischen den Entities e1 und e2 wird durch ein Fremdschlüssel f in der Tabelle t1, welches auf Tabelle t2 referenziert, abgebildet. Die Markierung dieses Knotens f in Kombination mit create bedeutet, dass dieses Element nur geschaffen wird, wenn es nicht schon existiert. Damit wird garantiert, dass nur ein Element f geschaffen wird. Desweiteren fügt die Regel der ersten Tabelle t1 eine neue Spalte nc für jede Spalte hinzu, die als Primärschlüssel in der zweiten Tabelle t2 markiert ist. Das Fremdschlüssel-Element f zeigt auf diese neuen Spalten nc.

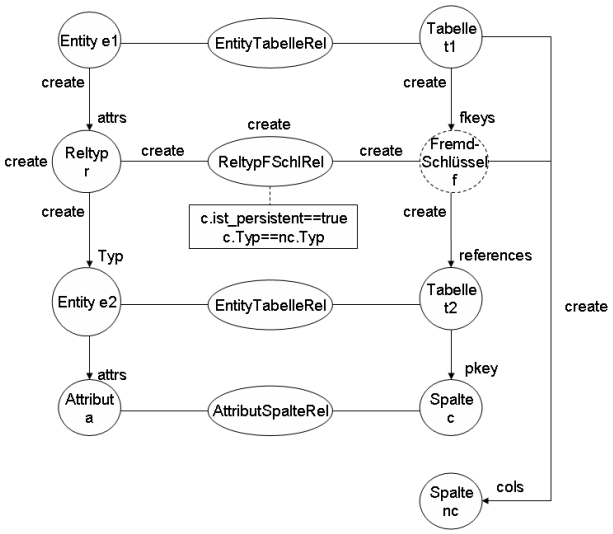

Abbildung 11: Transformation von Relationen mit persistentem Ziel-Entity

4.2 Asynchrone Regeln

Nun werden die asynchronen *Vorwärts- Rückwärts- und Konsistenzregeln* an diesem Beispiel dargestellt.
Abbildung 12 stellt die Transformation von einem existierenden Entity zu einer neuen Tabelle dar. Diese Regel erzeugt den Relationsknoten EntityTabelleRel, den korrespondierenden rechten Knoten Tabelle t zu dem gegebenen linken Knoten Entity e und die Kanten zwischen diesen Knoten. Dabei erhält die Tabelle t den Namen des Entities e.

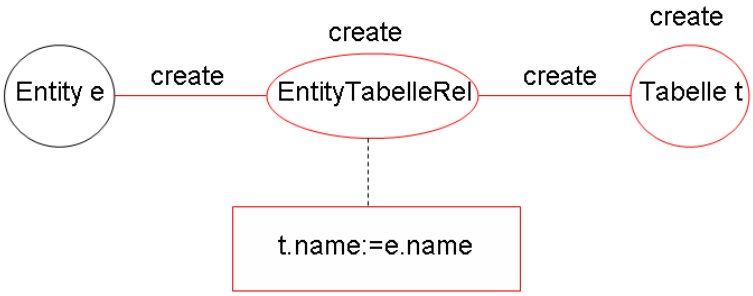

Abbildung 12: Vorwärtsregel

Abbildung 13 zeigt die Transformation von einer existierenden Tabelle zu einem neuen Entity. Diese Regel erzeugt den Relationsknoten EntityTabelleRel, den

korrespondierenden linken Knoten Entity e zu dem gegebenen rechten Knoten Tabelle t und die Kanten zwischen diesen Knoten. Der Name des Entities e entspricht dem Namen der Tabelle t.

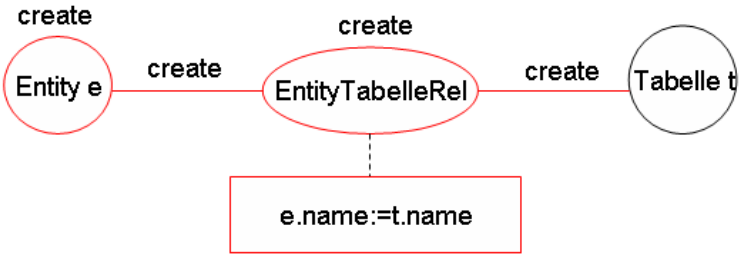

Abbildung 13: Rückwärtsregel

Abbildung 14 zeigt die Konsistenzregel zwischen einem existierenden Entity und einer existierenden Tabelle mit demselben Namen. Diese Regel erzeugt den Relationsknoten bzw. die Korrespondenzverbindung EntityTabelleRel zwischen den zwei Knoten Entity e und Tabelle t.

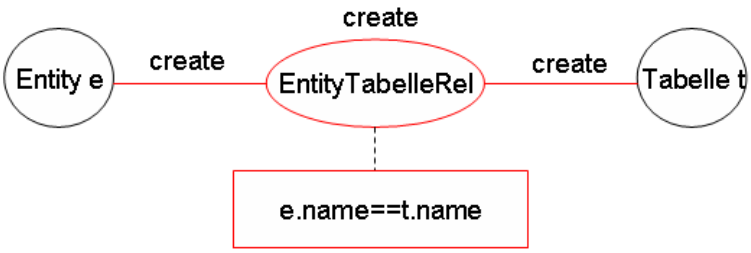

Abbildung 14: Konsistenzregel

5 Verwandte Ansätze

Neben Tripel-Graph-Grammatiken existieren auch andere Modelltransformationssprachen wie QVT (Query/ Views/ Transformations), die die deklarative, implementierungsunabhängige Definition von Sichten und Transformationen ermöglicht. QVT ist ein OMG-Standard und wurde für die Transformation von MOF-Metamodellen vorgesehen. Ziel ist die Definition einer einheitlichen Sprache, mit

deren Hilfe sich Modelle ineinander überführen lassen, deren Metamodelle durch die MOF spezifiziert wurden.
Eine QVT-Spezifikation setzt sich aus zwei Arten von Sprachen zusammen: aus einem deklarativen und einem imperativen Teil [1]. Die QVT-Spracharchitektur ist in Abbildung 15 dargestellt. Die Sprachen *QVT-Relations* und *QVT-Core* sowie eine Abbildung *RelationsToCoreTransformation* der *Relations*- auf die *Core*-Sprache bilden gemeinsam den deklarativen Teil. Die Sprache *QVT-Relations* unterstützt komplexes „Objekt-Muster-Matching“. QVT-Core ist eine Sprache, die nur das „Muster-Matching“ über einer flachen Menge von Variablen unterstützt, indem es Bedingungen über jenen Variablen gegen eine Menge von Modellen auswertet.
Der imperative Teil beinhaltet die Sprache *Operational Mappings Language* sowie Vorgaben für *Black Box Implementations*, also eine Schnittstelle für Transformationssprachen, die außerhalb von QVT implementiert sind. Mit Black-Box Transformationen können Mengen von Elementen des Quell- und Zielmodells in Beziehung gesetzt werden, ohne genau zu beschreiben, wie die einzelnen Elemente in Beziehung stehen. *Operational Mappings* können zur Implementierung einer oder mehrerer Relationen aus einer Relationsspezifikation benutzt werden. Um eine Transformation durchzuführen, werden die Regeln in *QVT-Relations* in *QVT-Core*-Regeln transformiert.

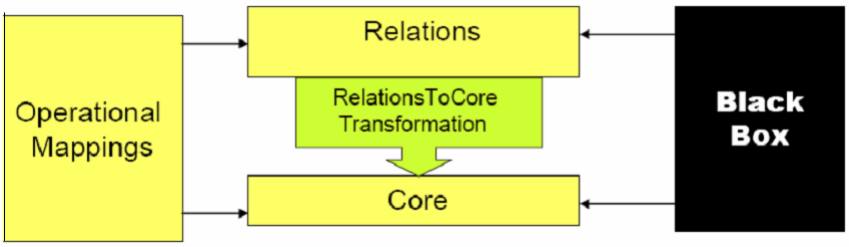

Abbildung 15: QVT-Spracharchitektur

Die deklarativen und imperativen Sprachen können auch gemeinsam verwendet werden, indem einzelne Relationen einer relationalen Transformation einer der imperativen Sprachen implementiert und in die relationale Transformation als „plug-in“ eingeklinkt werden. Diese gemeinsame Verwendung von deklarativen und imperativen Sprachen könnte zur Implementierung von wesentlich komplexeren Algorithmen eingesetzt werden.
In Abbildung 16 sind eine QVT-Core-Abbildung und eine Tripel-Graph- Grammatikregel gegenübergestellt. Der Vergleich von Tripel-Graph-Grammatiken und

QVT zeigt, dass beide Ansätze synchrone Transformationsregeln auf einer Konstellation von Modellelementen anwenden. Es lässt sich leicht erkennen, dass QVT-Core-Abbildungen und Tripel-Graph-Grammatikregeln sehr ähnlich sind: die Muster in der Abbildung guard und bottom entsprechen den Mustern auf der linken Kontext und rechten Seite create von Tripel-Graph-Grammatikregeln. Ein erwähnenswerter Vorteil von Tripel-Graph-Grammatiken ist, dass hier multiple Korrespondenzknoten existieren, in QVT ist dies nicht der Fall.

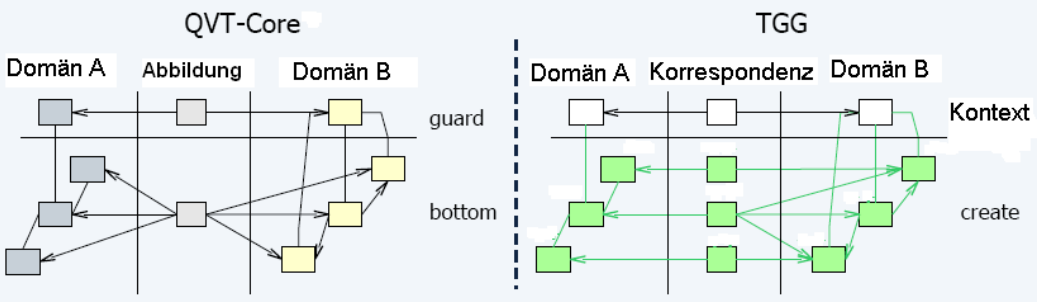

Abbildung 16: Gegenüberstellung einer QVT-Core-Abbildung und einer Tripel-Graph-Grammatikregel [5]

Nach einer Gegenüberstellung einer QVT- und einer Tripel-Graph-Grammatikregel wird Modelltransformation mit Tripel-Graph-Grammatiken mit QVT verglichen. Dabei fallen folgende Gemeinsamkeiten auf [5]:

- Abbildungen in QVT-Core und Tripel-Graph-Grammatikregeln spezifizieren Kontextmuster und gematchte Muster im Quell- und Zielmodell.
- QVT- Relationen und Tripel-Graph-Grammatikregeln setzen synchron Modellmuster in Beziehung.
- Modelltransformation ist in beiden Fällen Metamodell-basiert.
- QVT-Relationen und Tripel-Graph-Grammatikregeln ermöglichen die Spezifikation von Kontexten, in denen die Regel gültig ist.
- QVT-Relations, QVT-Core und Tripel-Graph-Grammatiken beschreiben Objektmuster in Quell- und Zielmodellen.

Neben den Gemeinsamkeiten weisen beide Ansätze zur Modelltransformation auch Unterschiede auf:

- Tripel-Graph-Grammatiken ermöglichen eine bidirektionale Transformation, bei QVT hingegen ist die Transformation nur in eine Richtung, also unidirektional. Mit der bidirektionalen Transformation ist es möglich, die vollzogene Transformation zurückzuverfolgen, d.h. für ein Element eines Zielmodells sind die Elemente eines Quellmodells identifizierbar, aus denen das Element entstanden ist.

- QVT-Relations und QVT-Core sind textbasierte Sprachen, jedoch spezifiziert QVT auch eine graphische Syntax für QVT-Relations, während in Tripel-Graph-Grammatiken die Notation graphisch ist.

- QVT- und Tripel-Graph-Grammatikregeln beziehen sich auf einen gewissen Kontext, in dem Modellelemente miteinander verbunden sind. Diese Regel wird in QVT als eine Startregel der Transformation betrachtet. Tripel-Graph-Grammatiken brauchen einen Startkontext, um die Transformation zu starten.

- QVT- und Tripel-Graph-Grammatikregeln nehmen Bezug auf einen Kontext. Aber QVT-Relationen rufen andere Relationen mit Parametervariablen auf. Tripel-Graph-Grammatikregeln jedoch markieren gewisse Knoten als Kontextknoten.

6 Bewertung und Ausblick

Eine Modelltransformation mit Tripel-Graph-Grammatikregeln wird durchgeführt, um einen Graphen *SG* in den Graphen *TG* zu transformieren. Dabei wird gleichzeitig der Korrespondenzgraph *CG* generiert. Dies erlaubt das Übertragen von inkrementellen Änderungen zwischen zwei Graphen *SG* und *TG*. Diese Übertragung von inkrementellen Änderungen ist bidirektional und somit besonders wichtig für iterative Softwareentwicklungsprozesse, in denen ein Modell sich kontinuierlich und simultan entwickelt.
Die praktische Anwendbarkeit von Tripel-Graph-Grammatiken für Graph-zu-Graph Transformationen ist in Kapitel 4 an einem praxisrelevanten Beispiel der Transformation von ER-Modellen in relationale Schemata präsentiert worden. Der Hauptvorteil von Tripel-Graph-Grammatiken ist die Fähigkeit, Transformationsregeln graphisch zu spezifizieren, denn die graphische Spezifikation von Modellen oder Regeln erleichtert die Verständlichkeit (vergleiche visuelle Sprachen [11]).
Ein weiterer Vorteil besteht darin, dass Verifikationstechniken eingesetzt werden können, um die Korrektheit solcher Tripel-Graph-Grammatiken zu beweisen, wodurch wiederum sichergestellt werden kann, dass die bei der Synchronisation

entstehenden Modelle ebenfalls korrekt sind. Ein letzter erwähnenswerter Vorteil der Tripel-Graph-Grammatiken ist, dass eine Transformation zwischen zwei Graphenpaaren möglich ist, die kein gemeinsames Metamodell aufweisen. Dadurch ist die Möglichkeit gegeben, von einem Modell zum anderen und wieder zurück zu gelangen.
Neben all diesen Vorteilen von Tripel-Graph-Grammatiken existieren dennoch Probleme bei der Anwendbarkeit von Tripel-Graph-Grammatikregeln in der Praxis. Bei der Transformation eines Quellgraphen in einen Zielgraphen müssen alle Transformationsregeln ausgeführt werden. Dieser Prozess erfordert viel Zeit. Die meiste Zeit wird vor allem bei dem Finden aller Muster auf der linken Seite benötigt. In der Praxis ist es jedoch unakzeptabel, mehrere Stunden zu warten, bis alle Transformationsregeln abgearbeitet und alle Muster der linken Regelseite gefunden sind. Daher wäre es sinnvoll, den „Matching-Algorithmus“ zu optimieren und zu beschleunigen.
Ein weiterer Nachteil ist, dass die Tripel-Graph-Grammatiken nur in graphbasierten Transformationen einsetzbar sind. In realen Modelltransformationssystemen liegen die Modelle jedoch oft nicht als Graphen vor und somit sind die Daten auch nicht direkt zugreifbar.
In der Praxis sind manchmal nur Teile eines Modells relevant und sollen transformiert werden. In diesem Fall sollten die Tripel-Graph-Grammatikregeln nur für die relevanten Teile der Transformation konstruiert werden. Wenn jedoch die weniger relevanten Teile des Modells die Transformation beeinflussen, können sie als Kontextknoten in Tripel-Graph-Grammatikregeln auftreten. Das würde bedeuten, dass obwohl nur einige Teile des Modells transformiert werden sollen, die Tripel-Graph-Grammatiken eine komplette Grammatik für alle teilnehmenden Modellteile spezifizieren müssen.
Außerdem ist das Verfahren der Tripel-Graph-Grammatiken zur Integration und Konsistenzüberprüfung zwischen Modellen nur dann anwendbar, wenn die kleinste mögliche Änderung an einem Modell das Löschen oder Erzeugen von Objekten ist. Wenn die Attributwerte eines Objektes, wie beispielsweise der Name eines Entities, verändert werden, so wird dies jedoch durch die vorgestellten Tripel-Graph-Grammatikregeln nicht erkannt.
Zudem existiert noch der Nachteil, dass beide Modelle im Speicher geladen sein müssen, damit eine Transformation zwischen den einzelnen Modellen vorgenommen werden kann. Dies ist besonders speicherintensiv. Desweiteren sind für die Transformation zwischen zwei Modellen sehr komplexe und umfangreiche Transformationsregeln aufzustellen.
Bisher wurde beobachtet, dass die Tripel-Graph-Grammatiken sowohl bei der Definition als auch bei der Ausführung von Graph-zu-Graph Transformationen sehr nützlich sind. Jedoch sind sie erweiterungsfähig und verbesserungsbedürftig. Daher werden im Folgenden Verbesserungsvorschläge bzw. Möglichkeiten zur Er-

weiterung von Tripel-Graph-Grammatiken in Anlehnung an [2] vorgestellt.
Tripel-Graph-Grammatiken spezifizieren eine Relation zwischen zwei Graphen-Modellen und erlauben somit $(1:1)$- Modelltransformationen. In der Praxis könnte es jedoch notwendig sein, dass mehrere Modelle in ein oder mehrere Modelle transformiert werden. Daher ist es sinnvoll, Tripel-Graph-Grammatiken in MGGs (Multi-Graph-Grammars) [5] zu erweitern, so dass $(m:n)$- Beziehungen zwischen Modellen dargestellt werden können. Eine MGG-Regel könnte wie in Abbildung 17 dargestellt aussehen. Die Knoten im Korrespondenzgraphen verbinden Knoten aus n Modellen mit Knoten aus m Modellen.

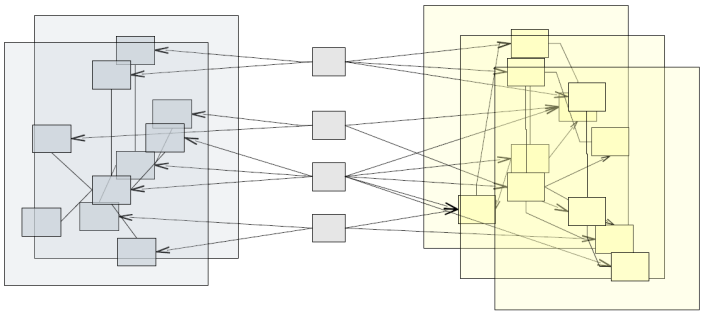

Abbildung 17: MGG-Regel

In dieser Arbeit wurden nur monotone Produktionen betrachtet, in denen die linke Seite L einer Produktion in der rechten Seite R enthalten sein muss. Das bedeutet, dass die Produktionen keine Knoten und Kanten löschen. Für die Zukunft wäre es allerdings interessant, die Tripel-Graph-Grammatikregeln auf Regeln zu generalisieren, die erlauben, Knoten und Kanten zu löschen. Das Erlauben von löschenden Tripel-Graph-Grammatikregeln könnte jedoch zu Entscheidbarkeitsproblemen führen.
Erwähnenswert ist auch die Erweiterung der Tripel-Graph-Grammatiken um imperative Sprachelemente mit dem Ziel, auch bestimmte Verhaltensweisen von Tripel-Graph-Grammatikregeln spezifizieren zu können, wie beispielsweise das Auslösen von anderen Transformationen. Bisher wurde versucht, auf einem gegebenen Graphen die gesamte Regelmenge anzuwenden. Die Reihenfolge der Regelausführung war unerheblich. Für die Tripel-Graph-Grammatiken würde der Einsatz von imperativen Sprachelementen bedeuten, dass die Grammatikregeln in einer explizit festgelegten Reihenfolge ausgeführt werden.
Die Integration von OCL (Object Constraint Language) ist eine weitere Möglichkeit zur Erweiterung oder Verbesserung der Tripel-Graph-Grammatiken. OCL ist eine Ergänzung der UML und soll die Modellierung von Software noch präziser

gestalten. In OCL werden zusätzliche Randbedingungen eines Modells spezifiziert, zum Beispiel die Beschränkung eines Attributs auf einen Wertebereich oder einzuhaltende Vorschriften zwischen Objekten. Die zwei deklarativen Sprachen von QVT, QVT-Relations und QVT-Core, benutzen die OCL, um die Relation von Objektstrukturen und zusätzlichen Bedingungen zu spezifizieren. OCL erlaubt die Formulierung komplexer Ausdrücke.
Mit einem optimierten Algorithmus für das Matching-Problem der linken Seite einer Regel, einer Verbesserung der oben genannten Nachteile und den möglichen Erweiterungen können Tripel-Graph-Grammatiken ein sehr nützliches Konzept für die Spezifikation und Anwendung von Graph-zu-Graph Transformationen in der modellbasierten Softwareentwicklung werden.

7 Zusammenfassung

In dieser Ausarbeitung wurde ein Überblick über Modelltransformation für die Modellintegration gegeben. Dabei bestand der Schwerpunkt auf graphbasierten Transformationen mit Tripel-Graph-Grammatiken. Zum besseren Verständnis der Tripel-Graph-Grammatiken wurden zunächst die Pair-Graph-Grammatiken vorgestellt, auf denen die Tripel-Graph-Grammatiken basieren. PGG ermöglichen die Graph-zu-Graph Transformationen. Dieser Ansatz wurde erweitert um einen Korrespondenzgraphen, der die Beziehungen zwischen dem Quell- und Zielmodell darstellt. Diese Erweiterung hatte den großen Vorteil, dass nun nicht mehr nur $(1:1)$- Beziehungen zwischen Knoten dargestellt werden konnten, sondern auch echte $(m:n)$- Beziehungen.
Mit Tripel-Graph-Grammatiken lassen sich Transformationsregeln für Modelle auf der Grundlage von Graphtransformationen beschreiben. Die Transformationsregeln sind synchron und definieren, welche Knotentypen einander entsprechen. Diese Regeln werden automatisch in asynchrone Graphtransformationsregeln (Vorwärtsregeln, Rückwärtsregeln, Konsistenzregeln) übersetzt, um verschiedene Modellintegrationsaufgaben wie Konsistenzüberprüfung, Konsistenzherstellung und Modelltransformation zu erfüllen.

Mit den Tripel-Graph-Grammatiken besteht also die Möglichkeit,

- Modelle bidirektional in andere Modelle zu überführen (Vorwärts- und Rückwärtsregel)

- die Konsistenz zwischen zwei Modellen zu überprüfen und herzustellen (Konsistenzregel)
- inkrementelle Änderungen an einem Dokument in das andere zu übernehmen.

Außerdem haben die Modelltransformationen mit Tripel-Graph-Grammatiken nach [12] folgende wichtige Vorteile:

- Die vorliegenden Datenmodelle sind Graphen und keine Bäume.
- Die gleiche Spezifikation kann als Beschreibung eines unidirektionalen sowie eines bidirektionalen Transformationsprozesses benutzt werden.
- Die Korrespondenzen werden explizit modelliert und sind nicht auf $(1:1)$-Beziehungen zwischen Objekten beschränkt, es können echte $(m:n)$- Beziehungen dargestellt werden.

Literatur

[1] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Final Adopted Specification, http://www.omg.org, 2005.

[2] *TGG Workshop-Presentations*. TU-Darmstadt, Real-Time Systems Lab, Institute of Computer Engineering, www.es.tu-darmstadt.de/index2.php?page=3130.

[3] ANNEKE KLEPPE, JOS WARMER und WIM BAST: *MDA Explained: The Model Driven Architecture–Practice and Promise (Paperback)*. Addison - Wesley, 2003.

[4] BALZERT, HELMUT: *Lehrbuch der Software-Technik, Bd. 1.: Software-Entwicklung*. Lehrbücher der Informatik, Spektrum, Akad. Verlag, Heidelberg, 1996.

[5] GREENYER, JOEL: *Reconciling TGGs with QVT*. Diploma Thesis, Faculty for Computer Science, Software-Engineering-Group, Universität Paderborn, 2006.

[6] KEMPER, ALFONS und ANDRÉ EICKLER: *Datenbanksysteme - Eine Einführung*. Oldenbourg Verlag München Wien, 2004.

[7] KÖNIGS, ALEXANDER: *Model Transformation with Triple Graph Grammars*. Real-Time Systems Lab, 2005.

[8] MARC BORN, ECKHARDT HOLZ und OLAF KATH: *Softwareentwicklung mit UML 2*. Addison-Wesley, 2004.

[9] MAYR, ERNST W.: *Einführung in die Informatik IV, SS 2005*. Fakultät für Informatik, TU München, http://www14.in.tum.de/lehre/2005SS/info4/index.html.de, 2005.

[10] PRATT, TERRENCE W.: *Pair grammars, graph languages and string-to-graph translations*. Journal of Computer and System Sciences 5, 5:560–595, 1971.

[11] SCHIFFER, STEFAN: *Visuelle Programmierung-Grundlagen und Einsatzmöglichkeiten*. Addison Wesley, 1998.

[12] SCHÜRR, ANDY: *Specification of Graph Translators with Triple Graph Grammars*. In Mayr, E. W., Schmidt, G., Tinhofer, G., eds.: Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science WG 1994, Seiten 151–163, 1994.