Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Inkrementelle Modellsynchronisation

Schriftliche Arbeit
zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften“
(Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Robert Wagner
Am Coesfeld 1
33334 Gütersloh

Paderborn, im Februar 2009

Zusammenfassung

Software wird immer komplexer. Gleichzeitig nehmen die Anforderungen an die Leistungsfähigkeit und die Qualität von Software beständig zu. Die steigende Komplexität der Software stellt die Softwareentwicklung, die durch die Globalisierung der Märkte zudem unter einem hohen Kosten- und Zeitdruck stattfindet, vor immer größere Probleme.

Die modellbasierte Softwareentwicklung ist ein viel versprechender Ansatz, um den Problemen, die mit der steigenden Komplexität bei der Softwareentwicklung einhergehen, zu begegnen, die Softwarequalität zu erhöhen und gleichzeitig den Entwicklungsaufwand signifikant zu reduzieren. Hierzu wird ein Softwaresystem mit unterschiedlichen Modellen beschrieben. Die unterschiedlichen Modelle sind notwendig, um verschiedene Gesichtspunkte und Sichtweisen auf ein Softwaresystem adäquat zu beschreiben. Die verwendeten Modelle beruhen zwar häufig auf verschiedenen Formalismen mit unterschiedlichen Notationen und Konzepten, aber aufgrund der Tatsache, dass sie ein und dasselbe Softwaresystem beschreiben, überlappen sie sich in ihrem Informationsgehalt.

Ein Problem ist, dass diese Überlappungen zu widersprüchlichen Aussagen über das Softwaresystem führen können. Um ein fehlerfreies Softwaresystem zu erhalten, müssen die Widersprüche zwischen den Modellen beseitigt werden, das heißt, die Modelle müssen miteinander abgeglichen werden. Insbesondere bei großen und komplexen Modellen ist ein Abgleich von Hand aber nicht nur mühsam und fehleranfällig, sondern zeitaufwändig und damit auch unwirtschaftlich.

In dieser Arbeit wird ein Ansatz zur *automatischen Modellsynchronisation* vorgestellt. Die Modellsynchronisation gleicht in Beziehung stehende Modelle miteinander ab und löst damit vorhandene Widersprüche zwischen den Modellen auf. Die Modellsynchronisation kann dabei sowohl vollständig in einem einzigen Schritt, d. h., *batch-artig*, als auch Schritt für Schritt, d. h., *inkrementell*, durchgeführt werden. Damit ist dieser Ansatz auch für *große Modelle* geeignet.

Darüber hinaus werden in dieser Arbeit eine *Methode* und dazugehörige *Softwarewerkzeuge* zur *modellbasierten* und *automatisierten Entwicklung von*

Modellsynchronisationswerkzeugen vorgestellt. Der in dieser Arbeit vorgestellte Ansatz ist dabei nicht auf die Modellsynchronisation beschränkt. Er eignet sich ebenso zur *Modellintegration*, *Modelltransformation* und *Codegenerierung*. Mit der prototypischen *Realisierung* der in dieser Arbeit dargestellten Konzepte konnte anhand verschiedener Beispiele und durchgeführter Leistungsmessungen gezeigt werden, dass die inkrementelle Modellsynchronisation auch bei großen Modellen *effizient* durchführbar ist.

Danksagung

Diese Arbeit wäre wohl niemals ohne die Mitwirkung von vielen netten und mir wohlgesonnenen Menschen entstanden. An dieser Stelle möchte ich mich bei all den Menschen ganz herzlich bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Ich danke meinem Doktorvater Wilhelm Schäfer für das mir entgegengebrachte Vertrauen, die damit verbundene Möglichkeit, in seiner Arbeitsgruppe an spannenden Themen aus dem Bereich der Softwaretechnik zu forschen, sowie dafür, dass er mir die Chance zur Promotion eröffnet hat. Ich möchte mich insbesondere für die Förderung meiner wissenschaftlichen und persönlichen Entwicklung in den Jahren meiner Mitarbeit in seiner Arbeitsgruppe bedanken. Die Arbeit unter seiner Leitung hat mir viel Freude gemacht. Neben Wilhelm waren aber auch Ekkart Kindler und Holger Giese an der wissenschaftlichen Betreuung meiner Arbeit beteiligt. Dafür bedanke ich mich ganz herzlich auch bei ihnen.

Bei den Mitgliedern meiner Prüfungskommission Andy Schürr, Ekkart Kindler, Gregor Engels und Heike Wehrheim bedanke ich mich dafür, dass sie sich mit meiner Arbeit auseinandergesetzt haben. Insbesondere danke ich Andy Schürr und Ekkart Kindler dafür, dass sie das Gutachten zu meiner Dissertation übernommen haben. Ferner danke ich Andy Schürr für die Entwicklung der Tripel-Graph-Grammatik, ohne die diese Dissertation ganz anders ausgesehen hätte oder womöglich gar nicht erst zustande gekommen wäre.

Ein ganz besonderer Dank gilt meinen Kollegen Björn Axenath, Matthias Meyer, Vladimir Rubin und Lothar Wendehals, die durch ihre Ideen sowie ihre hilfreiche und konstruktive Kritik zu der vorliegenden Dissertation beigetragen haben. An dieser Stelle danke ich aber auch allen anderen (ehemaligen) Kollegen, die sowohl mit wissenschaftlichen als auch mit privaten Gesprächen zu einer angenehmen Arbeitsatmosphäre beigetragen haben: Sven Burmester, Matthias Gehrke, Stefan Henkler, Martin Hirsch, Florian Klein, Ahmet Mehic, Ulrich Nickel, Jörg Niere, Daniela Schilling, Matthias Tichy, Dietrich Travkin und Jörg Wadsack.

Vielen Dank auch an Jutta Haupt für ihre Hilfe bei der Überwindung

manch einer bürokratischen Hürde, sowie an Jürgen Maniera für die technische Unterstützung. Danke auch an alle Studenten, die an der Umsetzung meiner Ideen als studentische Hilfskräfte und/oder im Rahmen ihrer Studien- und Diplomarbeiten mitgewirkt haben.

Schließlich möchte ich mich bei meinen Freunden und meiner Familie für ihre Unterstützung bedanken. Meinen Eltern danke ich insbesondere dafür, dass sie mir meine Ausbildung ermöglicht haben. Ein ganz besonderer und lieber Dank gilt aber meiner Frau Martina für ihre Geduld und Nachsicht, die sie während der Fertigstellung meiner Dissertation aufgebracht hat. Martina hat mir immer den notwendigen Rückhalt gegeben und für die erforderlichen Freiräume gesorgt, ohne die diese Arbeit wahrscheinlich immer noch nicht fertig gestellt wäre. Danke!

Inhalt

Kapitel 1

Einleitung

Die Einsatzbereiche für Software reichen heutzutage von der klassisch betriebswirtschaftlichen Anwendung, über die Anwendung in der Steuerungs- und Regelungstechnik im Maschinenbau und der Automobilindustrie, bis hin zur Multimediaanwendung in der Unterhaltungsbranche. Mit dem steigenden Einsatz und der weiten Verbreitung von Software wachsen aber gleichzeitig auch die Qualitätsansprüche. Insbesondere in sicherheitskritischen Anwendungen mit einem hohen Gefährdungspotenzial, wie zum Beispiel Werkzeugmaschinen, Produktionsanlagen oder Transportmitteln, muss die Korrektheit und Fehlerfreiheit der Software zwingend gewährleistet werden. Die steigende Komplexität der Software führt mit den zunehmenden Anforderungen an die Qualität zu einem immer größeren Aufwand bei der Entwicklung und damit auch zu höheren Entwicklungskosten. Einen Ansatz um der steigenden Komplexität und den damit einhergehenden Problemen zu begegnen, sowie den hohen Anforderungen an die Qualität gerecht zu werden, stellt die *modellbasierte Softwareentwicklung* dar.

1.1 Modellbasierte Softwareentwicklung

Die Grundlage der modellbasierten Softwareentwicklung bilden *Modelle*. Modelle sind zu einem integralen Bestandteil vieler wissenschaftlicher Methoden und Werkzeuge geworden. Im Allgemeinen ermöglichen Modelle eine

> *„...vereinfachte Darstellung der Funktion eines Gegenstands oder des Ablaufs eines Sachverhalts, die eine Untersuchung oder Erforschung erleichtert oder erst möglich macht.“* [Dud06].

Modelle werden in nahezu allen Bereichen genutzt, um auf relevante Eigenschaften eines betrachteten Systems zu fokussieren und von weniger wichtigen Details zu abstrahieren. Die Modelle helfen, die Komplexität des zu

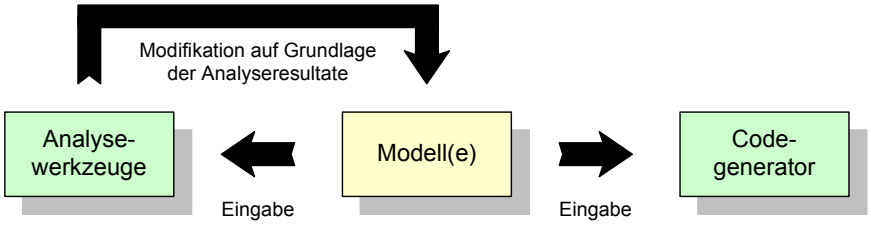


Abbildung 1.1: Modellbasierte Softwareentwicklung

entwickelnden Softwaresystems zu beherrschen. Zusätzlich verbessern sie die Kommunikation über das zu entwickelnde Softwaresystem und ermöglichen den Einsatz automatischer Analysetechniken, wie zum Beispiel Validierung durch Simulation, modellbasiertes Testen oder Verifikation durch Model-Checking [CGP00].

Der Ansatz der modellbasierten Softwareentwicklung ist in der Abbildung 1.1 schematisch dargestellt. Ausgangspunkt ist ein Modell[1], das als Eingabe für Analysewerkzeuge dient. Die Analysewerkzeuge überprüfen das Modell auf zuvor festgelegte Eigenschaften, die unbedingt einzuhalten sind. Wird eine Verletzung einer oder mehrerer dieser Eigenschaften bei der Analyse entdeckt, so muss das Modell korrigiert und die Überprüfung erneut ausgeführt werden. Sind hingegen alle Eigenschaften erfüllt, so kann aus dem Modell zumindest ein Teil der Implementierung mit Hilfe eines Codegenerators automatisch erzeugt werden.

Gegenüber der Implementierung von Hand besitzt die automatische Codegenerierung mehrere Vorteile [Her03]. Ein Vorteil gegenüber der manuellen Implementierung ist die erhöhte Produktivität. So kann ein Codegenerator den Code automatisch aus einem Modell erzeugen. Notwendige Änderungen an dem zu entwickelnden Softwaresystem können im Modell vorgenommen werden. Aufgrund der Abstraktion sind Änderungen im Modell deutlich einfacher durchzuführen als in der Implementierung. Diese Änderungen kann der Codegenerator dann wieder automatisch in Code umsetzen. Zusätzlich kann durch den Austausch des Codegenerators sehr einfach Code für eine weitere Zielplattform erzeugt werden. Auf diese Weise ist eine Portierung auf eine andere Zielplattform mit sehr geringem Aufwand möglich.

[1] Das (Gesamt-)Modell eines komplexen Softwaresystems setzt sich meistens aus mehreren Modellen zusammen. Je nach Sichtweise können wir hier also von einem oder von mehreren Modellen sprechen.

Ein weiterer Vorteil ergibt sich aus der Tatsache, dass durch den Einsatz eines Codegenerators die Softwarequalität gesteigert wird. Dies liegt daran, dass die Qualität der Implementierung von der Qualität des eingesetzten Codegenerators abhängig ist. Verbessert man den Codegenerator, so verbessert sich auch die Qualität des generierten Codes. Verwendet der Codegenerator nur bereits getestete und bewährte Codefragmente, so sinkt die Wahrscheinlichkeit für Softwarefehler.

Zusammenfassend kann man feststellen, dass die modellbasierte Softwareentwicklung viele Vorteile bietet. Sie kann insbesondere zu einer höheren Softwarequalität sowie Produktivität und damit zu geringeren Entwicklungskosten beitragen. Für einen praktikablen Einsatz der modellbasierten Softwareentwicklung muss allerdings zunächst ein Problem gelöst werden, das den Ausgangspunkt dieser Arbeit bildet. Das Problem wird im folgenden Abschnitt erläutert.

1.2 Problembeschreibung

Die Entwicklung eines komplexen Softwaresystems zeichnet sich dadurch aus, dass verschiedene Gesichtspunkte des Systems berücksichtigt werden müssen. Zur Beschreibung eines solchen Systems reicht daher nur selten ein einziges Modell aus. Die unterschiedlichen Gesichtspunkte und Sichtweisen auf das Softwaresystem erfordern den Einsatz verschiedener Modelle, denen typischerweise unterschiedliche Notationen zugrunde liegen. So eignen sich zur Beschreibung der statischen Struktur eines Softwaresystems andere Modelle als zur Beschreibung der dynamischen Anteile.[2]

Dieser Umstand wird auch durch heutige Modellierungssprachen berücksichtigt. Ein prominentes Beispiel hierfür ist die *Unified Modeling Language* (UML) [UML05]. Diese graphische Modellierungssprache definiert in der aktuellen Version insgesamt dreizehn verschiedene Diagrammarten, die jeweils als einzelne Modelle aufgefasst werden können. Sechs der Diagrammarten dienen zur Modellierung der statischen Struktur; die übrigen sieben Diagrammarten werden zur Modellierung des Verhaltens benutzt. Zwar müssen zur Beschreibung eines Softwaresystems nicht alle Diagrammarten zwangsläufig eingesetzt werden – allerdings ergeben erst die eingesetzten Modelle zusammen das Gesamtmodell des zu entwickelnden Softwaresystems. Ein anderes Beispiel sind *domänenspezifische Sprachen* (engl. Domain Spe-

[2] Hinzu kommt, dass auch in den verschiedenen Phasen der Softwareentwicklung unterschiedliche Modelle eingesetzt werden können.

cific Languages, kurz DSL), in denen unterschiedliche Sprachen aufeinander abgestimmt und zu einer einzigen Modellierungssprache integriert werden [WR99].

Zusätzlich können Modelle auf unterschiedlichen Abstraktionsebenen existieren. Während auf höheren Abstraktionsebenen auf wesentliche Artefakte eines Softwaresystems fokussiert wird, erfolgt in den darunter liegenden Abstraktionsebenen eine Verfeinerung dieser Artefakte. Die Modelle der unterschiedlichen Abstraktionsebenen sind somit voneinander abhängig, d. h., zwischen den Modellelementen der beteiligten Modelle existieren Beziehungen, die eingehalten werden müssen, um ein fehlerfreies Softwaresystem zu erhalten.

Hinzu kommt, dass an der Entwicklung großer und komplexer Softwaresysteme häufig viele Entwickler aus zum Teil unterschiedlichen Domänen beteiligt sind. Sie beschreiben das Softwaresystem aus unterschiedlichen Sichten und setzen dafür verschiedene Werkzeuge ein. Dies liegt daran, dass die heutzutage verfügbaren Modellierungswerkzeuge auf ein Anwendungsgebiet spezialisiert sind. Ein Werkzeug eignet sich damit für eine Aufgabe besonders gut, ein anderes für eine andere Aufgabe. In den meisten Fällen verwenden die Werkzeuge ein werkzeugspezifisches Modell, das inkompatibel zu allen anderen Werkzeugen ist. Damit muss für jedes Werkzeug ein eigenes Modell erstellt werden, wodurch die Anzahl der eingesetzten Modelle zusätzlich erhöht wird.

Das Problem stellen nicht die vielen Modelle an und für sich dar. Wie zuvor dargestellt, ermöglicht häufig erst der Einsatz unterschiedlicher Modelle die Beherrschung der Komplexität eines zu entwickelnden Softwaresystems. Das Problem besteht vielmehr darin, dass die Modelle voneinander abhängig sind. Die Modelle beschreiben ein und dasselbe Softwaresystem aus unterschiedlichen Perspektiven. Daher enthalten die Modelle (zumindest teilweise) gleiche Information über das zu entwickelnde Softwaresystem. Wenn die gleiche Information in mehreren Modellen repräsentiert ist, so sagt man, dass die Modelle sich *überlappen*. Durch die überlappenden Teile stehen die Modelle zueinander in Beziehung. Diese Überlappung der Information kann zu widersprüchlichen Aussagen über das zu entwickelnde Softwaresystem führen. Ein Widerspruch zwischen zwei Modellen wird auch *Inkonsistenz* genannt. Nuseibeh et al. definieren in [NER00] die Inkonsistenz ganz allgemein als

> *„…any situation in which a set of descriptions does not obey some relationship that should hold between them“.*

Diese Definition ist sehr generisch. Beziehen wir jedoch die in der Definition genannten Beschreibungen auf einzelne Modellelemente, so liegt eine Inkonsistenz vor, wenn die Modellelemente eine Beziehung verletzen, in der sie zueinander stehen sollten. Wird also eine geforderte Beziehung zwischen den Modellelementen zweier Modelle nicht eingehalten, so kann dies zu widersprüchlichen Aussagen über das Softwaresystem führen. Aufgrund der Tatsache, dass die automatische Codegenerierung aus inkonsistenten Modellen in fehlerhafter Software resultieren kann, sollten Inkonsistenzen zwischen Modellen beseitigt werden.

Die Beseitigung der Inkonsistenzen kann durch eine manuelle Überprüfung und Korrektur der beteiligten Modelle stattfinden. Bei großen und komplexen Softwaresystemen ist eine Überprüfung und Korrektur der Modelle von Hand aber nicht nur mühsam und fehleranfällig, sondern auch zeitaufwändig und damit unwirtschaftlich. Die manuelle Beseitigung der Inkonsistenzen erschwert somit den Entwicklungsprozess und macht die Vorzüge der modellbasierten Softwareentwicklung wieder zunichte. Für einen erfolgreichen und praktikablen Einsatz der modellbasierten Softwareentwicklung ist daher eine geeignete Werkzeugunterstützung nicht nur wünschenswert, sondern unabdingbar.

1.2.1 Modelltransformation

Lange Zeit wurden *automatische Modelltransformationen* als Lösung des Konsistenzproblems propagiert. Bei einer Modelltransformation wird auf der Grundlage von Transformationsregeln ein Quellmodell automatisch in ein Zielmodell übersetzt. Durch den automatischen Übersetzungsvorgang entsteht ein Zielmodell, das zum Quellmodell konsistent ist.

In der heutigen Praxis haben sich zur Softwareentwicklung weitestgehend iterativ-inkrementelle Entwicklungsprozesse durchgesetzt. Bei der iterativ-inkrementellen Softwareentwicklung wird ein Softwaresystem in aufeinander folgenden Ausbaustufen erstellt. In jeder Ausbaustufe wird das Softwaresystem um weitere Funktionen erweitert, bis schließlich das gesamte Softwaresystem realisiert ist. Dabei müssen die Modelle häufig auch nach einer bereits durchgeführten Modelltransformation angepasst und geändert werden, was erneut zu Inkonsistenzen zwischen den Modellen führen kann.

Die geänderten Modelle können durch eine erneute Modelltransformation wieder miteinander abgeglichen werden. Während am Anfang der iterativ-inkrementellen Softwareentwicklung die Modelle meistens noch relativ klein und überschaubar sind, werden die Modelle mit jedem Iterationszyklus im-

mer größer. Im fortgeschrittenen Stadium der Entwicklung können die Modelle sogar so groß werden, dass eine Modelltransformation mehrere Stunden dauert und damit für den Softwareentwickler – insbesondere nach geringfügigen Änderungen – nicht mehr zumutbar ist.

Ein weiteres Problem entsteht dadurch, dass bei einer Modelltransformation ein bereits bestehendes Zielmodell verworfen und ein komplett neues Zielmodell erzeugt wird. Häufig ist das Zielmodell jedoch mit zusätzlichen Informationen angereichert, die im Quellmodell nicht vorhanden sind. Weil die zu dem Zielmodell manuell hinzugefügten Informationen nicht automatisch aus dem Quellmodell erzeugt werden können, gehen sie durch eine erneute Modelltransformation unwiderruflich verloren.

Ein anderes Problem liegt vor, wenn beide Modelle unabhängig voneinander erstellt worden sind. In diesem Fall sind beide Modelle bereits gegeben – sie müssen lediglich auf Konsistenz überprüft und vorhandene Inkonsistenzen durch einen Abgleich der Modelle beseitigt werden. Für ein solches Szenario sind die meisten Ansätze zur Modelltransformation nicht ausgelegt.[3]

Heutige Ansätze zur Modelltransformation eignen sich somit zwar, um ein Quellmodell in ein Zielmodell zu übersetzen, aber weniger, um die Modelle miteinander abzugleichen und die Konsistenz zwischen ihnen über ihren gesamten Lebenszyklus sicherzustellen. Sie können Änderungen an einem bereits übersetzten Quellmodell nicht inkrementell an das Zielmodell weiterleiten (vergleiche Kapitel 8). Stattdessen erzeugen sie das Zielmodell bei einer erneuten Übersetzung immer wieder komplett neu, was häufig zu Informationsverlusten im Zielmodell führt. Hinzu kommt, dass eine komplette Übersetzung bei vielen Änderungen und großen Modellen ineffizient ist.

1.2.2 Codegenerierung

Ein Ziel der modellbasierten Softwareentwicklung besteht darin, die manuelle Programmierung überflüssig zu machen. Die heutige Praxis sieht allerdings anders aus. In den meisten Fällen müssen nach der Codegenerierung noch Änderungen und Ergänzungen am Code durchgeführt werden. Dies liegt daran, dass die Modelle aufgrund ihres hohen Abstraktionsniveaus nicht genügend Informationen enthalten, um daraus automatisch eine vollständige und lauffähige Implementierung zu generieren. Daher erzeugt der Codegenerator häufig nur ein Codegerüst, das weiter verfeinert werden muss.

[3] Einen Ansatz, mit dem sowohl dieses Szenario als auch die Modelltransformation realisiert werden können, lernen wir mit der in dieser Arbeit verwendeten Technik der Tripel-Graph-Grammatik [Sch94] kennen.

Hierbei ergeben sich Probleme, die wir auch schon bei der Modelltransformation identifiziert haben. So müssen Änderungen im Modell auch im Code umgesetzt werden, ohne dass manuell hinzugefügte Codefragmente verloren gehen. Umgekehrt müssen Änderungen im Code auch an das Modell propagiert werden. Genauso wie ein Abgleich zwischen zwei Modellen muss auch ein Abgleich zwischen einem Modell und dem dazugehörigen Code stattfinden. Dies ist nicht weiter überraschend, wenn man bedenkt, dass Code auch als ein Modell des zu entwickelnden Softwaresystems aufgefasst werden kann.

Natürlich gibt es bereits Werkzeuge, die eine Unterstützung zur Synchronisation von Modell und Code anbieten (vergleiche dazu auch Abschnitt 8.2). Diese Synchronisation wird in den meisten Werkzeugen mit Hilfe von Forward-Engineering und Reverse-Engineering realisiert [CC90]. Beim Forward-Engineering wird Code aus einem Modell generiert. Beim Reverse-Engineering wird ein Modell aus bereits vorhandenem Code erstellt. Dabei wird entweder der Code oder das Modell komplett neu erzeugt. Abhängig von der Richtung ersetzt dieser Vorgang somit die alte Version des Codes oder des Modells. Enthält der Code oder das Modell Informationen, die nicht in dem jeweils anderen Artefakt vorhanden sind, so führt ein Forward- bzw. Reverse-Engineering zum Verlust dieser Informationen. Der Ansatz ist somit nur für bijektive Abbildungen zwischen Modell und Code praktikabel. Diese sind in der Praxis aber nur selten gegeben [SK04].

1.2.3 Nachverfolgbarkeit

Der Ansatz der modellbasierten Softwareentwicklung geht davon aus, dass ein Softwaresystem nur noch mit Modellen beschrieben und die manuelle Programmierung nicht notwendig sein wird. Viele Werkzeuge arbeiten aber immer noch auf der Basis von Code. Ein Beispiel hierfür sind Debugger, die den Code zur Fehlersuche in Einzelschritten ausführen. Wird ein Fehler erkannt, so wird dieser im Code angezeigt. Aufgrund der fehlenden Zuordnung ist ein Übersetzungsvorgang nicht nachverfolgbar, das heißt, es ist nicht sofort ersichtlich, aus welchen Modellelementen der fehlerhafte Code generiert wurde. Es ist daher häufig einfacher, den Fehler direkt im Code zu korrigieren als die Ursache für den Fehler im Modell zu beheben.

Die Nachverfolgbarkeit (engl. Traceability) solcher Übersetzungsvorgänge ist nicht nur bei der Codegenerierung nützlich, sondern ganz allgemein bei allen Modelltransformationen [ANRS06]. So ist es häufig notwendig, ein Modell zum Zwecke der Analyse in den Formalismus des entsprechenden Analysewerkzeugs zu übersetzen. Die Analyse wird somit nicht auf dem

Modell durchgeführt, das dem Entwickler vertraut ist, sondern auf einem semantisch äquivalenten Modell, das dem Formalismus des Analysewerkzeugs genügt. Das Analysewerkzeug untersucht dieses Modell und bezieht sich bei der Präsentation der Analyseresultate natürlich nur auf dieses Modell. Folglich muss der Softwareentwickler die Analyseresultate im Formalismus des Analysewerkzeugs interpretieren. Hier wäre es von Vorteil, wenn die Analyseresultate im ursprünglichen Modell angezeigt werden könnten, da dieses Modell dem Entwickler vertraut ist.

1.2.4 Validierung und Verifikation

In diesem Kontext stellt der Übersetzungsvorgang ein weiteres Problem dar. Dieses Problem beruht auf der Tatsache, dass es zurzeit sehr schwierig ist, eine Übersetzung formal zu verifizieren, um ihre Korrektheit nachzuweisen. Insbesondere bei der Entwicklung sicherheitskritischer Softwaresysteme muss aber sichergestellt werden, dass bei der Übersetzung eines Modells in ein Modell des Analysewerkzeugs keine Fehler gemacht wurden. Ansonsten kann nicht gewährleistet werden, dass das überprüfte Modell dem tatsächlich vom Softwareentwickler erstellten Modell entspricht.

Dieses Problem gilt auch für die Codegenerierung. Der Nachweis von Eigenschaften in einem Modell ist nur dann wirklich nützlich, wenn garantiert werden kann, dass der generierte Code korrekt ist, das heißt, dass die im Modell überprüften Eigenschaften auch im Code eingehalten werden. Hierzu muss gewährleistet sein, dass der generierte Code semantisch äquivalent zu dem überprüften Modell ist.

Der formale Nachweis einer solchen Äquivalenz ist sehr aufwändig. Dies liegt daran, dass heutige Codegeneratoren nicht formal und abstrakt genug spezifiziert sind, sondern direkt in einer Programmiersprache implementiert werden. Die vielen Details in der Implementierung eines Codegenerators verhindern einen formalen Nachweis mit den heute verfügbaren Techniken. Daher wird die Korrektheit heutiger Codegeneratoren meistens nur empirisch überprüft. Eine formale Spezifikation dieser Übersetzung könnte eine geeignete Basis für einen solchen Korrektheitsnachweis darstellen und ihn deutlich vereinfachen.

1.3 Ziele und Beiträge

Ziel der vorliegenden Arbeit ist die Entwicklung einer Technik zur *Modellsynchronisation*. Die Modellsynchronisation soll zwei zueinander in Beziehung stehende Modelle *automatisch* miteinander abgleichen und dadurch Widersprüche zwischen den Modellen auflösen. Mit der Technik soll insbesondere eine *inkrementelle Modellsynchronisation* möglich sein, um auch sehr große Modelle *effizient* miteinander abgleichen zu können.

Die Modellsynchronisation zwischen zwei Modellen soll in beide Richtungen funktionieren, d. h., die Modellsynchronisation muss *bidirektional* ausführbar sein. Dabei sollen auch die zur Modellsynchronisation benötigten Spezialfälle der *Modelltransformation* und der *Modellintegration* berücksichtigt werden. Darüber hinaus soll es möglich sein, eine Synchronisation zwischen einem Modell und dem daraus generierten Code durchzuführen.

Eine weitere Anforderung ist, dass Beziehungen zwischen zwei Modellen explizit repräsentiert werden. Durch die explizite Angabe der Beziehungen soll ein Nachweis der semantischen Äquivalenz der in Beziehung stehenden Modelle ermöglicht werden. Beim Einsatz dieser Technik zur Synchronisation eines Modells mit dem daraus generierten Code wird dadurch die Möglichkeit geschaffen, die Korrektheit des generierten Codes einfacher zu überprüfen oder sogar formal nachzuweisen.

Bei der zu entwickelnden Technik soll eine Modellsynchronisation allerdings nicht von Hand programmiert, sondern modellbasiert entwickelt werden. Hierzu soll eine geeignete Methode entwickelt werden, mit der benötigte Modellsynchronisationswerkzeuge automatisiert erstellt werden können, so dass Softwareentwickler von der Komplexität einer manuellen Entwicklung weitestgehend befreit werden.[4]

Diese Ziele erreichen wir auf der Grundlage von Tripel-Graph-Grammatiken (TGGs) [Sch94]. Diese visuelle, deklarative und formale Spezifikationstechnik ist nicht neu – sie wurde bereits zur Modelltransformation und zur inkrementellen Modellintegration eingesetzt (z. B. in [Lef95, Bec07, Kön08], vgl. Abschnitt 8.1.1). Allerdings nutzen die technischen Ansätze das Potenzial der TGGs bisher nicht aus: Die Modelltransformationen und Modellintegrationen kann *entweder* batch-artig *oder* inkrementell ausgeführt werden. Zudem muss bei der inkrementellen Modelltransformation bzw. Modellintegration das *gesamte* Quell- und Korrespondenzmodell untersucht werden,

[4] Die hier nur überblicksartig dargestellten Anforderungen werden zusammen mit der entwickelten Methode in Abschnitt 2.4 noch genauer erläutert.

d. h., die eingesetzten Algorithmen arbeiten nur im Zielmodell inkrementell. In dieser Arbeit wurde hingegen ein Algorithmus entwickelt, der sowohl batch-artig als auch inkrementell ausgeführt werden kann. Bei der inkrementellen Ausführung wird nicht das gesamte Quell- und Korrespondenzmodell untersucht, sondern nur die von den Änderungen tatsächlich betroffenen Modellelemente. Durch die lokale Arbeitsweise lassen sich daher selbst große Modelle *schnell* und *effizient* miteinander synchronisieren.

Die in dieser Arbeit vorgestellte Technik ist nicht auf die Modellsynchronisation beschränkt. Sie eignet sich ebenso zur *Modelltransformation* und *Codegenerierung*. Damit können sowohl Modell-zu-Modell Beziehungen als auch Modell-zu-Text Beziehungen durchgängig *in einer Notation* spezifiziert werden, um auf dieser Grundlage eine Modelltransformation bzw. Codegenerierung automatisch auszuführen. Nach einer Codegenerierung erlaubt die Modellsynchronisation, das Modell und den dazu in Beziehung stehenden Code miteinander abzugleichen, wenn diese nach der Codegenerierung geändert wurden.

Bei sehr umfangreichen Metamodellen hat sich die Spezifikation von Modell-zu-Modell und Modell-zu-Text Beziehungen als schwierig und zeitaufwändig erwiesen. Um die Spezifikation der Beziehungen zu vereinfachen, haben wir daher einen Ansatz entwickelt, bei dem die Beziehungen durch die Angabe von zueinander korrespondierenden Beispielen definiert werden. Dabei werden die Beispiele in der Notation der beteiligten Sprachen angegeben. Aus den gegebenen Beispielen wird anschließend die zur Synchronisation benötigte Tripel-Graph-Grammatik automatisch synthetisiert. Mit diesem Ansatz konnte die Spezifikation einer Tripel-Graph-Grammatik signifikant vereinfacht werden.

Die zur Modellsynchronisation benötigten Werkzeuge müssen nicht von Hand programmiert sondern können automatisch generiert werden. Hierzu wurden im Rahmen dieser Arbeit eine *Methode* und dazugehörige *Softwarewerkzeuge* entwickelt. Bei der entwickelten Methode werden die Regeln zur Modellsynchronisation nicht fest in einem Werkzeug codiert, sondern aus der formalen Spezifikation mithilfe der realisierten Softwarewerkzeuge in ausführbaren Code übersetzt. Dieser Code wird zur Parametrisierung eines im Rahmen dieser Arbeit entwickelten Frameworks verwendet. Das Framework mit den ausführbaren Regeln ergibt ein Synchronisationswerkzeug, dass in andere Modellierungswerkzeuge integriert werden kann. Damit wird eine Modellsynchronisation in Anwendungsdomänen ermöglicht, die durch *heterogene Werkzeuglandschaften* geprägt sind.

Mit der prototypischen *Realisierung* der in dieser Arbeit dargestellten

Konzepte konnte anhand verschiedener Beispiele und durchgeführter Leistungsmessungen gezeigt werden, dass die *inkrementelle Modellsynchronisation* auch bei großen Modellen *effizient* durchführbar ist und somit kontinuierlich, das heißt, sofort nach jeder Änderung der Modelle, stattfinden kann. Zudem wurde gezeigt, dass mit der in dieser Arbeit vorgestellten Methode Modellsynchronisationswerkzeuge mit sehr geringem Aufwand realisiert werden können.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit befasst sich mit einer Technik zur automatischen und insbesondere inkrementellen Synchronisation von Modellen. Einen wesentlichen Schwerpunkt dabei bilden das erarbeitete Konzept zur Modellsynchronisation und die Methodik, die zur weitestgehend automatischen Erstellung von Modellsynchronisationswerkzeugen eingesetzt wird. Die hierzu benötigten Grundlagen werden – sofern nötig – dem jeweiligen Kapitel vorangestellt, so dass auf ein einführendes Grundlagenkapitel verzichtet wird. Die vorliegende Dissertation besitzt daher den folgenden Aufbau:

Kapitel 2 enthält ein Beispiel, an dem die Modellsynchronisation motiviert und die damit verbundenen Probleme näher vorgestellt werden. Anschließend werden relevante Begriffe und Kriterien für die Modellsynchronisation aufgestellt, anhand derer diese Arbeit eingeordnet wird. Im letzten Teil dieses Kapitels werden die Anforderungen an die Modellsynchronisation aufgestellt. Das Ziel dieses Kapitels ist, den Leser in das Thema der Modellsynchronisation und die damit verbundenen Probleme einzuführen.

Kapitel 3 stellt die Technik der Tripel-Graph-Grammatiken (TGG) zur Spezifikation von Korrespondenzregeln vor. Hierzu werden zunächst die notwendigen Grundlagen erläutert. Anschließend wird die grundlegende Syntax und Semantik der TGGs vorgestellt, die dann um nützliche Konzepte erweitert wird. Darauf aufbauend stellen wir typische Anwendungsszenarien für TGGs vor, zu denen auch die Modellsynchronisation gehört. Das Ziel dieses Kapitels ist es, den Leser mit der Spezifikationstechnik vertraut zu machen und mögliche Anwendungsszenarien aufzuzeigen.

Kapitel 4 zeigt, wie die in dieser Arbeit verwendete Spezifikationstechnik auch zur Beschreibung von Modell-zu-Text Beziehungen eingesetzt

werden kann. Diese Spezifikation gestaltet sich allerdings ohne eine angemessene Werkzeugunterstützung als recht umständlich. Daher präsentieren wir in diesem Kapitel zwei Spezifikationsvarianten mit denen auch Modell-zu-Text Beziehungen einfach und dennoch formal definiert werden können.

Kapitel 5 beschreibt den entwickelten Synchronisationsmechanismus. Der Synchronisationsmechanismus arbeitet sowohl batch-artig als auch inkrementell. Zunächst stellen wir die grundlegende Arbeitsweise vor. Anschließend zeigen wir notwendige Erweiterungen, die zur inkrementellen Modellsynchronisation benötigt werden. Die Realisierung mit sogenannten Storydiagrammen wird im letzten Teil dieses Kapitels vorgestellt.

Kapitel 6 gibt einen Überblick über Möglichkeiten der Validierung und Verifikation von Modelltransformationen. Dieser Forschungszweig ist noch nicht sehr weit fortgeschritten. Daher stellen wir im ersten Teil dieses Kapitels einige existierende Ansätze zur Überprüfung der syntaktischen Korrektheit vor und untersuchen, inwiefern diese Ansätze auf TGGs übertragbar sind. Im zweiten Teil dieses Kapitels beschäftigen wir uns mit Ansätzen zur formalen Verifikation der semantischen Korrektheit von Modelltransformationen.

Kapitel 7 präsentiert die im Rahmen dieser Arbeit entstandene Werkzeugunterstützung. Die prototypische Realisierung dieser Werkzeuge wurde auf der Basis bereits existierender Entwicklungsumgebungen durchgeführt. Daher werden diese Werkzeuge ebenfalls kurz vorgestellt. Zusätzlich werden in diesem Kapitel die Ergebnisse der durchgeführten Evaluation der prototypischen Implementierung präsentiert. Die Ergebnisse der Evaluation belegen, dass die inkrementelle Modellsynchronisation effizient durchführbar ist.

Kapitel 8 untersucht verwandte Arbeiten und vergleicht sie mit dem in dieser Arbeit vorgestellten Ansatz. Hierzu stellen wir zunächst Arbeiten vor, die TGGs bereits zur Modelltransformation und Modellintegration genutzt haben. Anschließend betrachten wir andere Ansätze zur Modelltransformation und Modellintegration und untersuchen, inwieweit diese Ansätze zur Modellsynchronisation geeignet sind. Wir schließen dieses Kapitel mit einigen Ansätzen, die sich mit der Vereinfachung von Spezifikationen im Rahmen von Modelltransformationen beschäftigen.

Kapitel 9 fasst die Ergebnisse der Arbeit zusammen und schließt mit einen Ausblick auf mögliche Erweiterungen die vorliegende Arbeit ab.

Kapitel 2

Modellsynchronisation

In der Literatur zur modellbasierten Softwareentwicklung wird hauptsächlich die Notwendigkeit zur Modellsynchronisation betont – Anforderungen, mögliche Probleme sowie konkrete Ansätze und Vorschläge zur Umsetzung werden jedoch nicht behandelt. Ziel dieses Kapitels ist es daher, dem Leser einen ersten Überblick über das Thema der Modellsynchronisation zu vermitteln, die damit verbundenen Probleme aufzuzeigen, sowie den in dieser Arbeit entwickelten methodischen Ansatz zur Erstellung von Modellsynchronisationswerkzeugen kurz vorzustellen. Hierzu betrachten wir zunächst ein Beispiel zur Modellsynchronisation und zeigen anhand einiger Szenarien typische Probleme, die bei der Modellsynchronisation auftreten können. Anschließend befassen wir uns mit Begriffen, die wir im Kontext der Modellsynchronisation verwenden und die für das weitere Verständnis dieser Arbeit hilfreich sind. In dem darauf folgenden Abschnitt stellen wir einige Kriterien vor, die zu einer Klassifikation von Ansätzen zur Modellsynchronisation herangezogen werden können. Anschließend geben wir einen Überblick über den in dieser Arbeit verfolgten Ansatz und klassifizieren ihn auf Grundlage der zuvor festgelegten Kriterien. Im letzten Abschnitt fassen wir die Ergebnisse dieses Kapitels zusammen.

2.1 Ein Beispiel

In diesem Abschnitt wird die Modellsynchronisation an einem Beispiel aus der Domäne der automatisierten Fertigungssysteme vorgestellt. Hierzu geben wir zunächst einen Überblick über die Domäne und den damit verbundenen Synchronisationsbedarf. Anschließend erläutern wir exemplarisch einige Synchronisationsszenarien.

2.1.1 Hintergrund zur Domäne

In produzierenden Unternehmen werden seit geraumer Zeit automatisierte Fertigungssysteme eingesetzt, um die Fertigung mit einer hohen Qualität und mit niedrigen Kosten durchzuführen. Durch die Globalisierung der Absatzmärkte ist allerdings ein Käufermarkt[1] entstanden, der immer kürzere Innovationszyklen und eine steigende Variantenvielfalt erwartet. In dieser Marktsituation ist es für den Erfolg eines Unternehmens wichtig, schnell auf Marktänderungen reagieren und ein bestehendes Fertigungssystem schnell und kostengünstig an neue Anforderungen anpassen zu können.

In der Industrie werden daher immer häufiger *Flexible Fertigungssysteme* eingesetzt. Ein flexibles Fertigungssystem ist modular aufgebaut und besteht typischerweise aus mehreren Werkzeugmaschinen, die über ein automatisches *Materialflusssystem* miteinander verbunden sind. Zusätzlich können in das Fertigungssystem Handarbeitsplätze sowie Material- und Werkstücklager integriert sein. Die Handarbeitsplätze sind für Aufgaben vorgesehen, die nicht automatisiert durchgeführt werden können. Die Lager sorgen für eine unterbrechungsfreie Versorgung der Produktion mit den benötigten Roh-, Halbfertig- und Fertigteilen.

Ein Beispiel für ein solches flexibel automatisiertes Fertigungssystem ist das an der Universität Paderborn im Labor für Rechnerintegrierte Produktion aufgebaute Fertigungssystem zur Produktion von Flaschenöffnern. Der experimentelle Aufbau ist schematisch in der linken Hälfte der Abbildung 2.1 dargestellt. Das Fertigungssystem besteht aus vier Stationen, die über ein automatisiertes Materialflusssystem miteinander verbunden sind. Bei dem verwendeten Materialflusssystem handelt es sich um ein schienengebundenes Transportsystem mit selbst fahrenden Förderfahrzeugen, die *Shuttles* genannt werden. Zu den Grundelementen des Transportsystems gehören unterschiedliche Schienen, Weichen und Stationen. Aus diesen Grundelementen können – je nach Anwendung und räumlichen Gegebenheiten – verschiedene Topologien aufgebaut werden. In unserem Beispiel besteht die Topologie des Materialflusssystems aus einer Haupt- und einer Nebenschleife. Die Nebenschleife kann über Weichen erreicht und wieder verlassen werden. Für den Transport der Werkstücke zwischen den einzelnen Stationen werden mehrere Shuttles eingesetzt, deren Umlaufrichtung im Materialflusssystem fest vorgegeben ist. Dadurch wird gegenläufiger Verkehr auf der Schiene verhindert.

Zur Steuerung der Anlage wird zwischen der Betriebsleitebene, der Pro-

[1] Als Käufermarkt wird eine Marktsituation bezeichnet, in der sich der Käufer in einer verhandlungstaktisch günstigeren Position als der Verkäufer befindet.

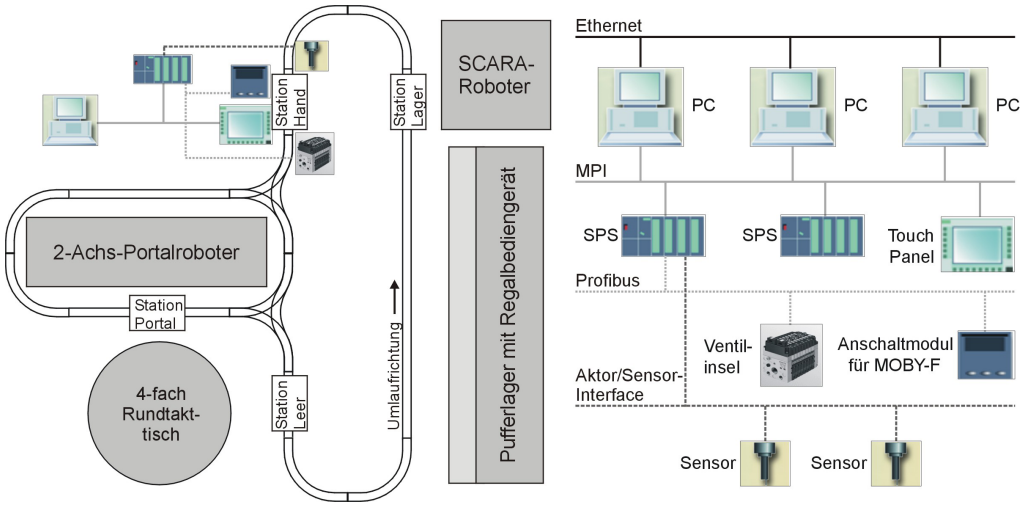


Abbildung 2.1: Schematische Darstellung des Fertigungssystems und der verwendeten Steuerungstechnik

zessleitebene, der Steuerungsebene und der Feldebene unterschieden (vgl. Abbildung 2.1, rechts). Die Betriebs- und Prozessleitebene werden häufig zusammengefasst und einfach nur als Leitebene bezeichnet. Die Leitebene ist für übergeordnete Aufgaben zuständig, die insbesondere die Produktionsplanung betreffen. Hierzu werden herkömmliche PCs eingesetzt. Die Steuerungs- und Feldebene dient zur Steuerung lokaler Komponenten wie zum Beispiel Stationen und Werkzeugmaschinen. Hierzu haben sich Speicherprogrammierbare Steuerungen (SPS) durchgesetzt, die ihre Popularität insbesondere ihrer Robustheit, ihrer hohen Verfügbarkeit und ihren niedrigen Anschaffungskosten verdanken. Die SPSen interagieren mit den Komponenten der Anlage über Sensoren und Aktoren (auch Aktuatoren genannt). Hierzu sind sie entweder über ein Actuator-Sensor-Interface (ASI) oder einen Profibus mit den SPSen verbunden. Die Kommunikation der SPSen mit den PCs der Leitebene findet über ein Multi-Point-Interface (MPI) statt. Zur Programmierung einer SPS werden in der Praxis verschiedene Sprachen eingesetzt, die in der herstellerunabhängigen SPS-Programmiernorm IEC 61131-3 standardisiert sind [IEC03].

Die Flexibilität eines solchen Fertigungssystems ergibt sich daraus, dass die Werkstücke sowohl an den einzelnen Werkzeugmaschinen mit unterschiedlichen Bearbeitungsverfahren gefertigt werden können als auch dadurch, dass die Reihenfolge der Bearbeitungsschritte durch das eingesetzte

Materialflusssystem flexibel gestaltet werden kann. Daher können auf einem Fertigungssystem verschiedene Produkte kostengünstig auch in kleiner Serie gefertigt werden.

Ein weiterer Vorteil eines solchen Fertigungssystems besteht darin, dass in kürzester Zeit weitere Stationen hinzugefügt werden können. Beispielsweise kann durch eine Erweiterung des Materialflusssystems und eine Mehrfachauslegung von Stationen das Produktionsvolumen gesteigert werden. Ebenso kann durch die Integration neuer Werkzeugmaschinen die Variantenvielfalt der Produkte erhöht werden. Durch diese Maßnahmen kann die Produktion an die Bedürfnisse des Marktes schnell angepasst werden.

Dem gegenüber steht allerdings die erhöhte Komplexität bei der Entwicklung der Steuerungssoftware. Diese muss insbesondere bei einer Erweiterung eines solchen Fertigungssystems an die neuen Gegebenheiten angepasst werden. Damit keine langen Ausfallzeiten bei der Produktion entstehen, sollten auch die nötigen Softwareanpassungen möglichst schnell realisiert werden können. Als ein besonderes Problem hat sich dabei allerdings das Fehlen einer durchgängigen Methodik zur Spezifikation der Software für derartige Systeme erwiesen.

2.1.2 Das ISILEIT-Projekt

Im ISILEIT[2]-Projekt wurde zur Verbesserung der zuvor beschriebenen Situation eine durchgängige Methode zur Erstellung von Steuerungssoftware für flexibel automatisierte Fertigungssysteme entwickelt. In diesem Abschnitt stellen wir den entwickelten Ansatz kurz vor. Für eine ausführlichere Darstellung verweisen wir auf [SWGE04] und [NSZ03].

Um eine vollständige Spezifikation der Steuerungssoftware für flexible Fertigungssysteme zu ermöglichen, wurden verschiedene Modellierungstechniken, wie die Specification and Description Language (SDL) [ITU96] und die Unified Modeling Language (UML) [UML05], zu einer durchgängigen Spezifikationssprache integriert. Das Ziel der Integration ist, eine präzise und konsistente Modellierung auf einer hohen Abstraktionsebene zu ermöglichen und daraus ausführbaren Code zu generieren. Auf der Basis dieser Konzepte wurde eine Methode entwickelt, die eine Möglichkeit zur Simulation der spezifizierten Steuerungssoftware bietet und dadurch eine Analyse in

[2] ISILEIT ist ein Akronym für 'Integrative Spezifikation von verteilten Leitsystemen der flexibel automatisierten Fertigung'. Das Projekt wurde im Rahmen des DFG Schwerpunktprogramms Software-Spezifikation – Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen (SPP 1064) durchgeführt.

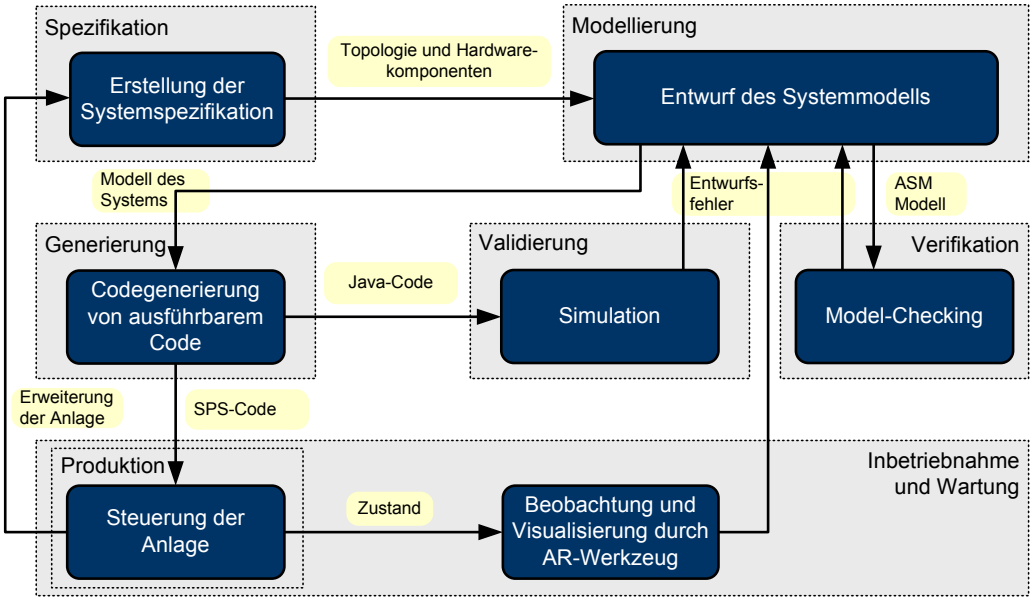


Abbildung 2.2: Überblick zur ISILEIT-Methode

den frühen Entwicklungsphasen ermöglicht. Die entwickelte Methode umfasst die Bereiche *Spezifikation* und *Modellierung*, *Generierung*, *Verifikation* und *Validierung* sowie *Inbetriebnahme und Wartung*. Die Methode ist in Abbildung 2.2 schematisch dargestellt.

Die *Spezifikation* wird in der Regel durch Ingenieure durchgeführt. Die Ingenieure legen die zu verwendenden Hardwarekomponenten und deren Eigenschaften fest. Zur Spezifikation der Hardwarekomponenten werden SDL-Blockdiagramme verwendet. Die Blockdiagramme beschreiben den hierarchischen Aufbau der eingesetzten Hardwarekomponenten sowie notwendige Kommunikationskanäle, die zur Steuerung dieser Komponenten benötigt werden. Zusätzlich legen die Ingenieure die Topologie, also den statischen Aufbau, des Fertigungssystem fest. Die Topologie stellt die Grundlage für eine spätere Simulation des Fertigungssystems dar.

Zur *Modellierung* eines Fertigungssystems wird das SDL-Blockdiagramm in ein UML-Klassendiagramm überführt. Dazu werden Blöcke und Prozesse des Blockdiagramms auf Klassen eines Klassendiagramms abgebildet; die Kommunikationskanäle im Blockdiagramm werden zu Assoziationen zwischen korrespondierenden Klassen im Klassendiagramm. Das Klassendiagramm wird anschließend durch das Hinzufügen von Attributen und Methoden sowie zusätzlichen Klassen und Assoziationen ver-

feinert. Zur Verhaltensmodellierung wird hingegen eine Kombination aus UML-Zustandsdiagrammen, UML-Aktivitätsdiagrammen und UML-Kollaborationsdiagrammen verwendet, deren Semantik im Rahmen des ISILEIT-Projekts formal mit Graphgrammatiken spezifiziert wurde. Aufgrund der durchgeführten Formalisierung entstehen sowohl für die Hardwarekomponenten als auch für die Steuerungssoftware ausführbare Modelle, die zur automatischen Codegenerierung und zur Verifikation und Validierung des Systems verwendet werden können.

Die formale *Verifikation* erfolgt aufgrund der Komplexität des Gesamtsystems (Modell der Hardware und der Steuerungssoftware) nur für sicherheitskritische Teile des Fertigungssystems. Als Grundlage für die Verifikation wurden Abstract State Machines (ASM) und die Abstract State Machine Language (AsmL) in Kombination mit einem auf Model-Checking Techniken basierenden Verifikationsverfahren verwendet [KR04].

Die automatische *Generierung* von ausführbarem Code dient der Validierung und der anschließenden Steuerung der Anlage. Zur Validierung werden die Modelle der Hardwarekomponenten und der Steuerungssoftware mithilfe eines Codegenerators in Java-Code übersetzt. Der zur Steuerung des realen Fertigungssystems benötigte SPS-Code wird durch einen SPS-Codegenerator erzeugt.

Zur Überprüfung des Gesamtsystems wurde die *Validierung durch Simulation* in die entwickelte Methode aufgenommen. Zur Simulation wird der generierte Java-Code kompiliert und ausgeführt. Die Visualisierung dieser Simulation erfolgt durch eine angekoppelte 3D-Software. Dadurch wird eine Überprüfung und Analyse der entwickelten Steuerungssoftware in einem sehr frühen Stadium der Entwicklung ermöglicht, so dass Fehler noch vor der Inbetriebnahme der Anlage beseitigt werden können.

Zur Unterstützung der *Inbetriebnahme und Wartung* wurde eine Augmented Reality (AR) Anwendung entwickelt, die sowohl die Zustände der Steuerungssoftware als auch die Zustände der realen Anlage beobachtet und visualisiert. Hierdurch können Wirkzusammenhänge leichter identifiziert werden, so dass Fehler, die bei der Validierung nicht erkannt wurden, nachvollzogen werden können. Durch diese Unterstützung verkürzt sich die Zeit für die Inbetriebnahme und Wartung der Anlage [Eck07].

Nach der Inbetriebnahme der Anlage wird der generierte und während der Inbetriebnahme getestete SPS-Code zur Steuerung der Produktion eingesetzt. Flexible Fertigungssysteme zeichnen sich dadurch aus, dass sie schnell und einfach an neue Anforderungen angepasst werden können. Bei einer Änderung der Anlage muss allerdings häufig auch die Steuerungssoft-

ware erweitert werden. Daher ist die ISILEIT-Methode auf ein iterativ-inkrementelles Vorgehen bei der Entwicklung der Steuerungssoftware ausgelegt. Dies erklärt die Zyklen in Abbildung 2.2.

2.1.3 Synchronisationsbedarf

Bei der ISILEIT-Methode findet an mehreren Stellen eine automatische Übersetzung statt. Zum Beispiel wird das SDL-Blockdiagramm in ein initiales UML-Klassendiagramm übersetzt, um dann das Verhalten von Klassen, die einen SDL-Prozess repräsentieren, mit einem Zustandsdiagramm zu spezifizieren. Eine weitere Übersetzung erfolgt bei der Codegenerierung. Hier wird das Modell der Steuerungssoftware zur Validierung durch Simulation in Java-Code und zur Steuerung der realen Anlage in SPS-Code übersetzt.

Bei einer Anpassung des Fertigungssystems an neue Anforderungen, die zum Beispiel den Einsatz neuer Hardwarekomponenten und/oder eine Änderung der Topologie der Anlage erfordern, muss in der Regel die Spezifikation des Fertigungssystems geändert und der Entwurf der Steuerungssoftware an die neue Spezifikation angepasst werden. Eine manuelle Anpassung des Entwurfs an die neue Spezifikation kann insbesondere bei großen und komplexen Fertigungssystemen sehr zeitaufwändig sein. Eine erneute, automatische Übersetzung der Spezifikation in einen initialen Entwurf führt dazu, dass die bereits durchgeführten Verfeinerungen im Entwurf überschrieben und dadurch verloren gehen – sie müssen dann erneut modelliert werden. Dies erhöht den benötigten Zeitaufwand und die mit der Anpassung des Fertigungssystems verbundenen Entwicklungskosten für die Steuerungssoftware.

Diese Problematik existiert auch bei der Codegenerierung. Beispielsweise wird der generierte Java-Code zur Visualisierung der Simulation an eine 3D-Software angebunden. Der hierzu benötigte Code muss manuell zu dem generierten Code hinzugefügt werden. Erfolgt nach durchgeführten Änderungen im Entwurfsmodell eine erneute Codegenerierung, so werden die manuell hinzugefügten Codeergänzungen durch den Generierungsvorgang überschrieben. Die notwendigen Ergänzungen zur Anbindung der 3D-Software müssen daher erneut manuell durchgeführt werden.

Um den zusätzlichen Aufwand, der beim Überschreiben manueller Änderungen entsteht, zu vermeiden, wird ein Werkzeug benötigt, das statt einer einfachen Übersetzung eine *Synchronisation* durchführt. Eine solche Synchronisation muss sowohl zwischen zwei Modellen als auch zwischen einem Modell und dem daraus generierten Code möglich sein.

2.1.4 Synchronisationsszenarien

In diesem Abschnitt betrachten wir einige typische Szenarien der Modellsynchronisation an einem Beispiel. Als Beispiel dient uns die Modellsynchronisation zwischen einem SDL-Blockdiagramm und einem UML-Klassendiagramm aus dem ISILEIT-Projekt.[3]

Ein SDL-Blockdiagramm dient der hierarchischen Strukturierung eines Systems. Ein *Block* auf der obersten Ebene stellt das spezifizierte *System* dar. Ein System muss mindestens einen Block enthalten, wobei jeder Block aus weiteren Blöcken bestehen kann. Hierdurch ergibt sich die hierarchische Strukturierung des Systems. Blöcke können *Prozesse* enthalten. Allerdings darf ein Block nie Prozesse und Blöcke gleichzeitig enthalten. Die Kommunikation zwischen den Elementen findet mithilfe von *Signalen* statt, die über *Kanäle* zwischen den Elementen eines Blockdiagramms ausgetauscht werden.[4]

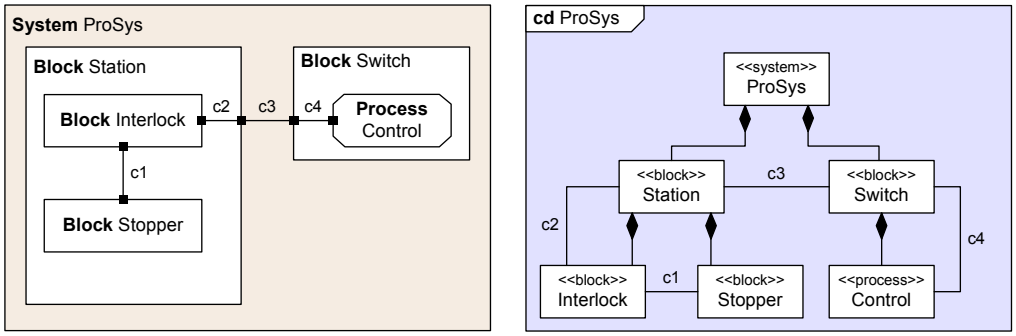
In Abbildung 2.3(a) ist ein SDL-Blockdiagramm dargestellt. Der oberste Block ist das System. In der graphischen Darstellung wird ein System durch das Schlüsselwort `System`, ein Block durch das Schlüsselwort `Block` und ein Prozess durch das Schlüsselwort `Process` gekennzeichnet. Systeme, Blöcke und Prozesse besitzen eindeutige Namen. In unserem Beispiel heißt das System `ProSys` (als Abkürzung für „Production System“). Es enthält die beiden Blöcke `Station` und `Switch`. Der Block `Station` ist weiter unterteilt in die Blöcke `Interlock` und `Stopper`. Der Block `Switch` enthält den Prozess `Control`.

Die Blöcke sind untereinander über die Kanäle `c1, c2` und `c3` verbunden. Zusätzlich besteht eine Verbindung zwischen dem Block `Switch` und dem darin enthaltenen Prozess `Control` über den Kanal `c4`. Kanäle sind nur zwischen Blöcken derselben Ebene erlaubt (vgl. Kanal `c1` und `c3`) oder zwischen einem übergeordneten Block und seinen direkt darin enthaltenen Blöcken oder Prozessen (vgl. Kanäle `c2` und `c4`).

In Abbildung 2.3(b) ist das zum Blockdiagramm korrespondierende Klassendiagramm dargestellt. Es besteht aus Klassen, die mit unterschiedlichen Stereotypen annotiert sind, sowie Assoziationen zwischen diesen Klassen.

[3] Ein Beispiel zur Synchronisation von Modell und Code geben wir in Kapitel 4.

[4] In SDL wird zwischen *verzögernden* und *verzögerungsfreien* Kanälen unterschieden. Außerdem kommunizieren Prozesse über sogenannte *Signalrouten*. Um unser Beispiel jedoch möglichst einfach zu halten, verzichten wir an dieser Stelle auf diese Unterscheidung und sprechen lediglich von Kanälen – auch wenn damit in einigen Fällen Signalrouten gemeint sind. Zusätzlich verzichten wir auf Signale an Kanälen.



(a) SDL-Blockdiagramm (b) UML-Klassendiagramm

Abbildung 2.3: Zwei zueinander korrespondierende Modelle

Eine spezielle Assoziation ist die Komposition. Die Komposition wird durch eine ausgefüllte Raute an einem Ende einer Assoziation dargestellt. Sie wird verwendet, um die hierarchische Struktur eines Blockdiagramms im Klassendiagramm wiederzugeben.

Die Zuordnung der Elemente eines Blockdiagramms zu einem korrespondierenden Klassendiagramm ist informell in Abbildung 2.4 dargestellt. In jeder der Zuordnungen 1–6 sehen wir auf der linken Seite die Elemente eines Blockdiagramms und auf der rechten Seite die dazu korrespondierenden Elemente eines Klassendiagramms. Die gestrichelt dargestellten Elemente geben den Kontext der von der Zuordnung betroffenen Elemente an.

Ein System wird auf eine Klasse im Klassendiagramm abgebildet (Zuordnung 1). In der Zuordnung sind sowohl das System als auch die Klasse mit einem `X` benannt. Dies bedeutet, dass beide Elemente nur dann zueinander korrespondieren, wenn die Namen dieser Elemente identisch sind. Zusätzlich ist die Klasse mit dem Stereotyp ≪system≫ gekennzeichnet. Diese Kennzeichnung ist nötig, um eine Klasse, die ein System repräsentiert, von einer Klasse zu unterscheiden, die einem Block oder Prozess zugeordnet wurde.

Die Abbildung eines Blocks auf eine Klasse im Klassendiagramm wird in den nachfolgenden beiden Zuordnungen dokumentiert (Zuordnungen 2 und 3). Auch hier heißen sowohl der Block als auch die Klasse gleich. Im Gegensatz zur vorherigen Zuordnung wird die Klasse jedoch jetzt mit dem Stereotyp ≪block≫ annotiert. Ist der betrachtete Block in einem System enthalten, so existiert im Klassendiagramm zwischen der Klasse, die dem Block zugeordnet ist, und der Klasse, die das System repräsentiert, eine Kompositionsbeziehung (Zuordnung 2). Ist der Block hingegen in einem

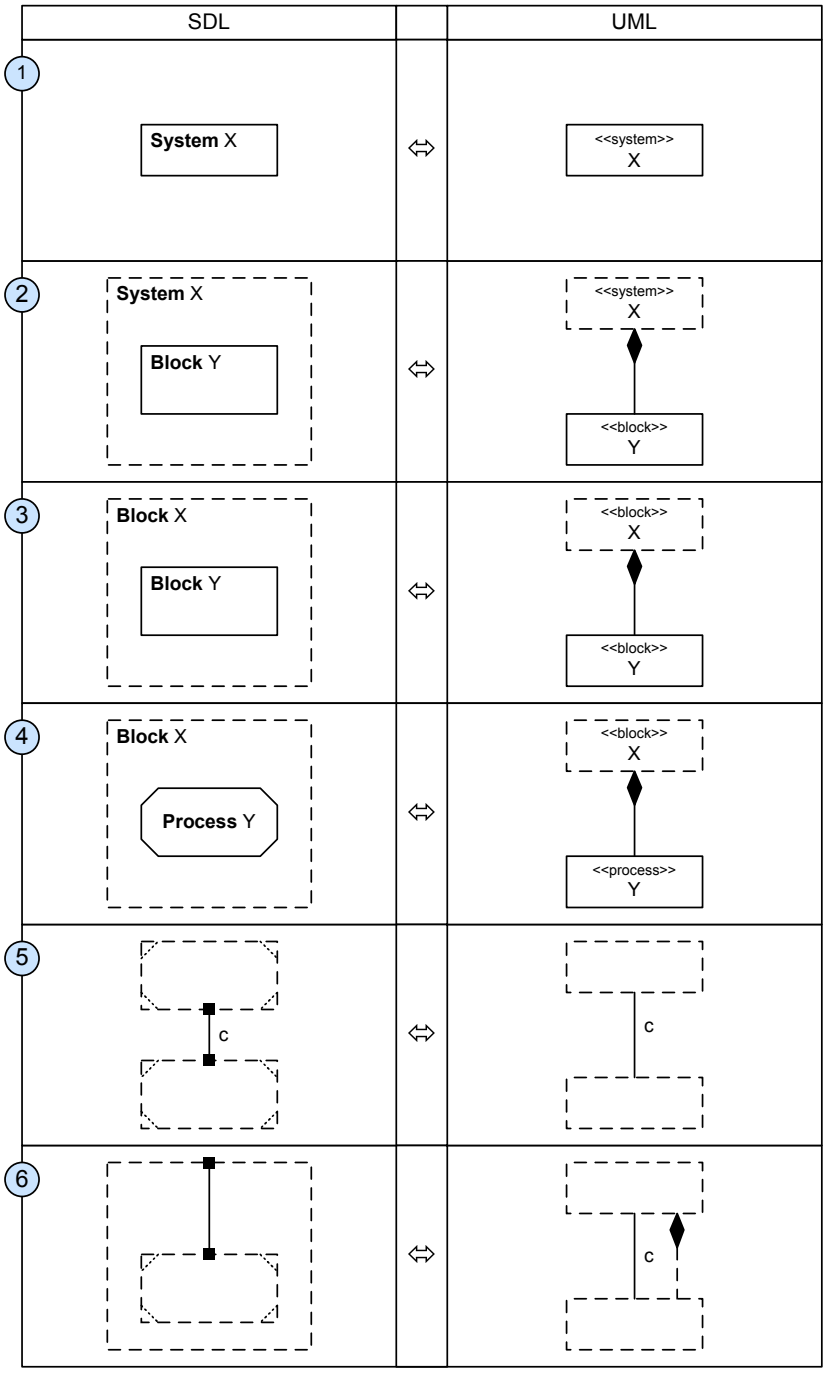


Abbildung 2.4: Informelle Zuordnung von Elementen eines Blockdiagramms zu Elementen eines Klassendiagramms

übergeordneten Block enthalten, so wird die Komposition zu der Klasse hergestellt, die den übergeordneten Block repräsentiert (Zuordnung 3).

Einem Prozess wird ebenfalls eine Klasse im Klassendiagramm zugeordnet (Zuordnung 4). Der Prozess wird hier zu einer Klasse in Beziehung gesetzt, die mit dem Stereotyp ≪process≫ gekennzeichnet ist. Die hierarchische Struktur wird – wie in den vorangegangen Zuordnungen – wieder durch eine Komposition abgebildet. Allerdings kann ein Prozess nur in einem Block enthalten sein, so dass hier keine Fallunterscheidung nötig ist.

Ein Kanal eines Blockdiagramms wird im Klassendiagramm auf eine Assoziation abgebildet (Zuordnungen 5 und 6). Bei den Elementen, die durch den Kanal verbunden werden, kann es sich um ein System sowie Blöcke und Prozesse handeln. Ein Kanal zwischen diesen Elementen kann auf der gleichen Hierarchieebene existieren (vgl. Zuordnung 5) oder zwischen Elementen benachbarter Hierarchieebenen (vgl. Zuordnung 6). Die zueinander korrespondierenden Kanäle und Assoziationen müssen identisch benannt sein.

Die Zuordnungen können verwendet werden, um ein Blockdiagramm in ein korrespondierendes Klassendiagramm zu übersetzen. Nach Änderungen in den Diagrammen können diese Zuordnungen herangezogen werden, um zu überprüfen, ob die beiden Diagramme weiterhin zueinander synchron sind. Zusätzlich kann aus den Zuordnungen abgeleitet werden, wie die Diagramme verändert werden müssen, damit sie wieder synchron zueinander sind. Hierfür schauen wir uns einige Synchronisationsszenarien an.

Initiale Synchronisationsszenarien

In unserem ersten Synchronisationsszenario gehen wir davon aus, dass ein Blockdiagramm bereits existiert und nun mit einem leeren Klassendiagramm synchronisiert wird. Diese initiale Synchronisation kann durch eine *Modelltransformation* erreicht werden. Eine Modelltransformation erhält ein Quellmodell als Eingabe und erstellt auf der Grundlage von definierten Regeln aus diesem Modell ein Zielmodell. In unserem Beispiel sind die Regeln durch die informellen Zuordnungen gegeben. Als Quellmodell dient uns das Blockdiagramm aus Abbildung 2.3(a). Das Ergebnis der initialen Modellsynchronisation durch Modelltransformation ist das uns bereits aus der Abbildung 2.3(b) bekannte Klassendiagramm.

Ein solches Synchronisationsszenario ist auch in der umgekehrten Richtung denkbar. In diesem Fall ist ein Klassendiagramm vorhanden, aber kein dazu korrespondierendes Blockdiagramm. Diese Situation kann zum Beispiel dann eintreten, wenn die Spezifikation des Blockdiagramms abhanden

gekommen und nicht mehr verfügbar ist. Das zum Klassendiagramm korrespondierende Blockdiagramm kann wieder durch eine Modelltransformation erzeugt werden, wobei jetzt die Eingabe ein Klassendiagramm und das Ergebnis der Modelltransformation ein Blockdiagramm ist.

Ein weiteres Synchronisationsszenario ergibt sich, wenn beide Modelle gegeben sind und der Benutzer überprüfen möchte, ob die beiden Modelle zueinander synchron sind. Dazu müssen die zueinander korrespondierenden Modellelemente identifiziert und in Beziehung gesetzt werden. Falls alle Modellelemente zu Modellelementen des jeweils anderen Modells zugeordnet werden konnten, sind die beiden Modelle zueinander synchron. Ansonsten müssen die Modelle geeignet miteinander synchronisiert werden.

Einfache Synchronisationsszenarien

Nach einer initialen Modellsynchronisation können die Modelle von einem Entwickler geändert werden. Der Entwickler kann neue Modellelemente hinzufügen, bestehende Modellelemente verändern oder sie gänzlich aus einem Modell löschen. Eine Modellsynchronisation kann sofort nach jeder dieser Änderungen erfolgen oder erst nach einer gewissen Anzahl von durchgeführten Änderungen. Außerdem kann der Entwickler die Änderungen an beiden Modellen durchführen, ohne dass er zwischendurch die Modelle miteinander synchronisiert.

Abbildung 2.5 zeigt auf der linken Seite die beiden Diagramme aus den Abbildungen 2.3(a) und 2.3(b), nachdem der Entwickler diese Diagramme geändert hat. Zur besseren Übersicht sind die Änderungen farblich unterlegt und durch die Begriffe `created`, `modified` und `deleted` gekennzeichnet. Hinzugefügte Modellelemente sind grün unterlegt und mit dem Begriff `created` annotiert, gelöschte Elemente sind rot unterlegt und mit `deleted` gekennzeichnet.[5] Änderungen an den Modellelementen sind orange unterlegt und zusätzlich durch das Wort `modified` gekennzeichnet.

In unserem Beispiel hat der Entwickler im Blockdiagramm den Namen des Blocks `Switch` in `Robot` geändert. Darüber hinaus hat er einen neuen Block `Storage` angelegt und den Block `Stopper` sowie den dazugehörigen Kanal `c1` gelöscht. Im Klassendiagramm hingegen hat er die beiden Klassen `Storage` und `Loading` erstellt. Die Klasse `Storage` wurde mit dem Stereotyp ≪block≫ gekennzeichnet und über eine Kompositionsbeziehung mit der

[5] Die als gelöscht markierten Modellelemente in Abbildung 2.5 sind nur aus Präsentationsgründen noch vorhanden – sie sind als gelöscht und nicht mehr existent zu betrachten.

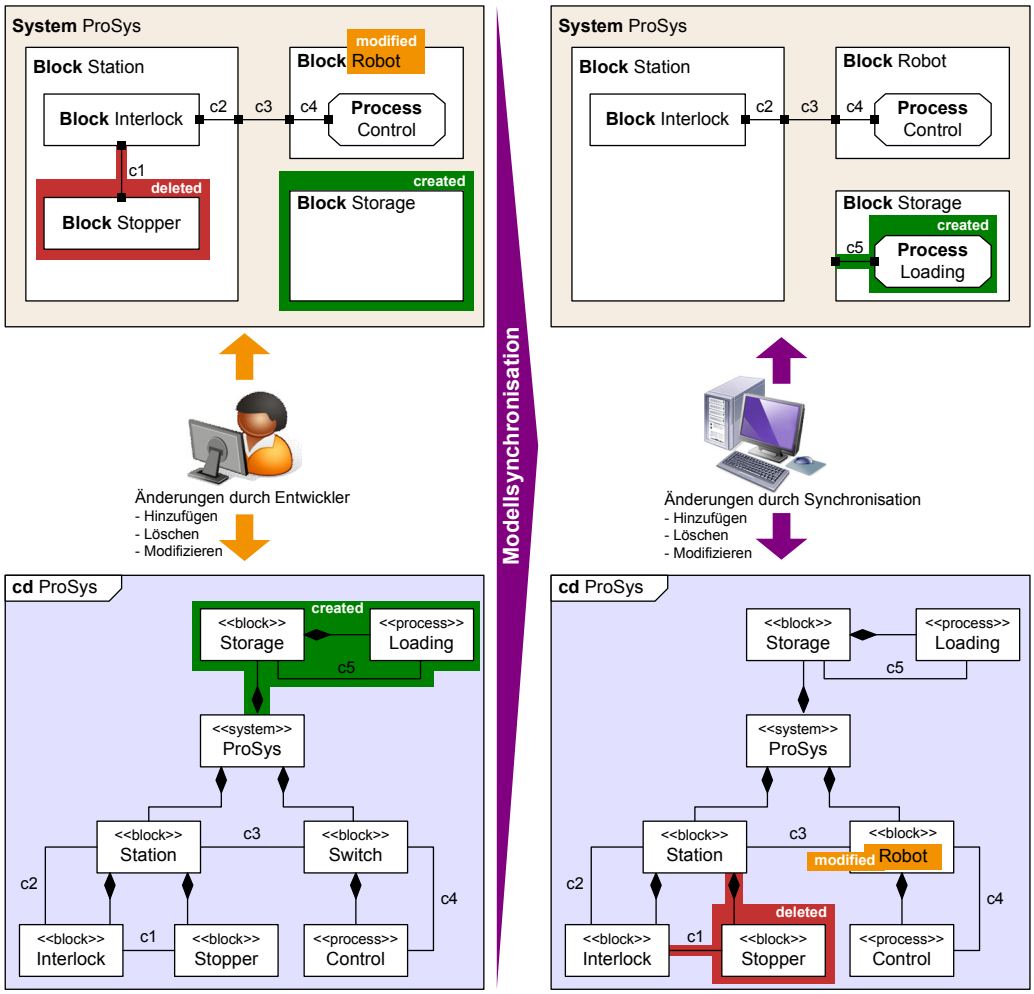


Abbildung 2.5: Diagramme vor (links) und nach (rechts) der Modellsynchronisation

Klasse `ProSys` verbunden. Die Klasse `Loading` wurde hingegen mit dem Stereotypen ≪process≫ versehen und durch eine Kompositionsbeziehung mit der Klasse `Storage` verbunden. Zusätzlich enthält das Klassendiagramm eine neue Assoziation `c5` zwischen den Klassen `Storage` und `Loading`.

Die Änderungen an den beiden Diagrammen führen dazu, dass die Diagramme nicht mehr zueinander synchron sind. Die Namensänderung des Blocks `Switch` in `Robot` wurde nur im Blockdiagramm vorgenommen – die dazu korrespondierende Klasse heißt immer noch `Switch`. Die zum gelöschten Block `Stopper` korrespondierende Klasse `Stopper` ist immer noch im Klassendiagramm vorhanden. Dies gilt auch für die Assoziation `c1`. Zu dem Block `Storage` hingegen wurde eine korrespondierende Klasse `Storage` mit dem Stereotyp ≪block≫ und einer dazugehörigen Komposition erstellt. Diese Änderung ist also konsistent in beiden Diagrammen erfolgt. Allerdings wurde im Klassendiagramm die Klasse `Loading` mit einem Stereotypen ≪process≫ angelegt, zu der keine Entsprechung im Blockdiagramm vorhanden ist. Dies gilt auch für die Assoziation `c5` zwischen den Klassen `Storage` und `Loading`.

Die einfachste Möglichkeit, beide Diagramme miteinander zu synchronisieren besteht darin, alle inkonsistent durchgeführten Änderungen rückgängig zu machen. Dadurch wären die Diagramme bezüglich der Korrespondenzregeln wieder zueinander synchron. Diese Art der Modellsynchronisation entspricht allerdings nicht der Erwartungshaltung eines Entwicklers – insbesondere weil dadurch die von ihm gemachten Änderungen verloren gehen würden. Außerdem würde dies bedeuten, dass einmal erstellte und synchronisierte Modelle nur noch konsistent zueinander erweitert werden könnten, was eine erhebliche Einschränkung des Entwicklers darstellen würde.

Zueinander synchrone Diagramme erhalten wir auch, wenn wir – wie im initialen Synchronisationsszenario – eine Modelltransformation einsetzen. Allerdings können in diesem Fall Änderungen in einem der Diagramme verloren gehen. Transformieren wir zum Beispiel das Blockdiagramm in ein Klassendiagramm, so sind nach der Modelltransformation die Klasse `Loading` und die Assoziation `c5` nicht mehr im Klassendiagramm vorhanden. Übersetzen wir hingegen das Klassendiagramm in ein Blockdiagramm, so wird die Namensänderung des Blocks `Switch` in `Robot` und die Löschung des Blocks `Stopper` nicht berücksichtigt. Sollen Änderungen in beiden Diagrammen berücksichtigt werden, so ist eine Modelltransformation ungeeignet.

Um Änderungen in beiden Diagrammen zu berücksichtigen, wird eine inkrementelle Modellsynchronisation benötigt, die geänderte Modellelemente in beide Richtungen miteinander abgleicht. Dabei werden die Inkonsisten-

zen zwischen den Diagrammen durch die Modellsynchronisation automatisch aufgelöst, und zwar so, dass möglichst keine Änderungen eines Entwicklers rückgängig gemacht werden müssen. Hierzu wird in unserem Beispiel zuerst im Klassendiagramm die Klasse `Stopper`, die dazugehörige Komposition und die Assoziation `c1` entfernt. Anschließend erfolgt eine Umbenennung der Klasse `Switch` in `Robot`. Im Blockdiagramm ergänzen wir den Block `Storage` um den Prozess `Loading` und einen Kanal `c5` zwischen diesen Elementen. Nach diesen Änderungen sind beide Diagramme wieder synchron zueinander. Das Ergebnis der Modellsynchronisation ist auf der rechten Seite der Abbildung 2.5 zu sehen, wobei auch hier die Änderungen wieder farblich unterlegt sind – diesmal handelt es sich jedoch um Änderungen, die im Rahmen der automatischen Modellsynchronisation durchgeführt wurden.

Problematische Synchronisationsszenarien

Im vorangegangenen Abschnitt konnten wir beide Diagramme ohne weitere Probleme miteinander synchronisieren. Die Modellsynchronisation ist aber nicht immer so problemlos möglich. In einigen Fällen bieten sich mehrere Möglichkeiten an, wie die Modelle miteinander synchronisiert werden können. Einige dieser Synchronisationsszenarien stellen wir nun vor.

In unserem Beispiel wird ein SDL-Blockdiagramm in ein initiales UML-Klassendiagramm mit dem Ziel übersetzt, das UML-Klassendiagramm anschließend zu verfeinern. Bei diesen Verfeinerungen kann der Entwickler zu bestehenden Klassen verschiedene Attribute und Methoden hinzufügen, aber auch neue Klassen und Assoziationen zwischen den Klassen im UML-Klassendiagramm erstellen.

Eine problematische Situation entsteht zum Beispiel, wenn in einer zu einem Block korrespondierenden Klasse zuerst einige Attribute hinzugefügt werden und anschließend der Block in einen anderen Block innerhalb des Blockdiagramms verschoben wird. In dieser Situation könnte die Verschiebung so interpretiert werden, dass ein alter Block gelöscht und ein neuer Block innerhalb eines anderen Blocks erstellt wurde. Bei dieser Interpretation könnte eine Modellsynchronisation – wie in dem vorangegangenen Abschnitt beschrieben – so vorgehen, dass sie die zum gelöschten Block korrespondierende Klasse löscht und an der zur neuen Position im Blockdiagramm korrespondierenden Stelle im Klassendiagramm eine neue Klasse erstellt. In diesem Fall gehen allerdings die manuell hinzugefügten Attribute der alten Klasse verloren.

Ein weiteres problematisches Synchronisationsszenario ergibt sich, wenn

der Entwickler im Klassendiagramm eine neue Klasse erzeugt. Falls die Klasse mit einem Stereotyp gekennzeichnet ist und eine Kompositionsbeziehung zu einer anderen Klasse besitzt, die ein System oder einen Block repräsentiert, können wir – wie im Szenario des vorangegangenen Abschnitts dargestellt – die beiden Diagramme miteinander synchronisieren, indem wir ein entsprechendes Element im Blockdiagramm erzeugen. Hat der Entwickler jedoch eine Klasse ohne Stereotyp und Kompositionsbeziehung erstellt, so müsste diese Klasse vor einer Modellsynchronisation um die fehlenden Modellelemente ergänzt werden. Ist im SDL-Blockdiagramm aber kein Block oder Prozess mit identischem Namen vorhanden, so wissen wir nicht, ob diese Klasse mit einem ≪block≫ oder ≪process≫ Stereotyp vervollständigt werden soll. Zusätzlich ergibt sich das Problem, dass nicht klar ist, zu welcher Klasse die Kompositionsbeziehung erstellt werden soll. Diese Entscheidungen könnten durch Standardvorgaben festgelegt werden. Beispielsweise könnte eine solche Klasse immer um einen ≪block≫ Stereotyp ergänzt und eine Kompositionsbeziehung zum System erstellt werden. Allerdings könnte es sich bei einer Klasse ohne Stereotyp und Kompositionsbeziehung ebenso um eine Hilfsklasse handeln, die zwar im Entwurf benötigt wird aber nicht mit dem Blockdiagramm synchronisiert werden soll. In diesem Fall sollte diese Klasse von einer Synchronisation ausgeschlossen werden und die Modellsynchronisation nur partiell, das heißt, nur für einen Teil des Klassendiagramms, durchgeführt werden.

Eine Ergänzung gestaltet sich hingegen einfacher, wenn zu einer neu erstellten Klasse ein bereits korrespondierendes Element im Blockdiagramm mit einem identischen Namen existiert. In diesem Fall wissen wir, um welchen Stereotyp die Klasse erweitert und zu welchem Element die Kompositionsbeziehung hergestellt werden muss. Allerdings ist hierbei zu beachten, dass die neu erstellte Klasse nicht zwangsläufig in der Absicht erstellt wurde, ein Element des Blockdiagramms zu repräsentieren. Hier kann ein einfacher Namenskonflikt zwischen dem Block und der Klasse vorliegen, der zum Beispiel durch eine Umbenennung eines der beiden Elemente aufgelöst werden kann. Auch hier kann es besser sein, die Synchronisation nur partiell durchzuführen.

Einige weitere problematische Synchronisationsszenarien ergeben sich, wenn zum Beispiel Assoziationen zwischen Klassen erstellt werden, die dazu korrespondierenden Kanäle aber über die Hierarchieebenen des Blockdiagramms gehen würden. Sollen die beiden Diagramme immer vollständig zueinander synchron sein, so müsste eine solche Assoziation gelöscht werden. Häufig wird so eine Assoziation im Klassendiagramm benötigt, ohne

dass sie mit dem Blockdiagramm synchronisiert werden soll. Daher ist es auch hier sinnvoll, die Assoziation von einer Synchronisation auszuschließen und die Modellsynchronisation nur partiell durchzuführen.

Zu Konflikten kommt es auch, wenn zusätzliche Kompositionsbeziehungen zwischen Klassen im Klassendiagramm erstellt werden, die nicht im Blockdiagramm abgebildet werden können. Ein Beispiel hierfür ist, wenn eine Klasse, die einen Block repräsentiert, über eine Komposition mit einer Klasse verbunden ist, die einen Prozess darstellt und einer weiteren Klasse, die einen Block repräsentiert. Diese Situation ist im Blockdiagramm nicht zulässig, da ein Block entweder nur Blöcke oder nur Prozesse enthalten darf, aber niemals beide gleichzeitig.

Die Ursache für diese Art von Problemen ist, dass die Modelle im Regelfall nicht bijektiv aufeinander abbildbar sind. So ist in unserem Beispiel eine isomorphe Abbildung zwischen einem Block- und einem Klassendiagramm nicht möglich. Allerdings kann – wie bereits zu Anfang dieses Abschnitts erwähnt wurde – eine vollständige Synchronisation zwischen zwei Modellen immer dadurch erzwungen werden, dass die Modellelemente, die keine Entsprechung in dem jeweils anderen Modell haben, oder die die Ursache für Konflikte darstellen, einfach gelöscht werden. Aus Benutzersicht ist dieser Eingriff jedoch nicht immer vorteilhaft und erwünscht. Hier ist es häufig besser, zueinander in Konflikt stehende Modellelemente und Modellelemente, die nicht zugeordnet werden konnten, dem Benutzer zu präsentieren und ihm die Entscheidung zu überlassen, wie mit diesen Elementen weiter verfahren werden soll. Diese partielle Modellsynchronisation ermöglicht ein „Leben mit Inkonsistenzen“ [Bal91, EC01].

2.2 Begriffe und Definitionen

Diese Arbeit beschäftigt sich mit dem Thema der „Modellsynchronisation“. Obwohl dieser Begriff in der Literatur häufig verwendet wird [GSCK04, LTM+04, IK04a, KC05a, XLH+07], existieren bislang noch keine standardisierten Begriffsdefinitionen, wie sie beispielsweise für viele Begriffe der Softwaretechnik in der Norm [IEE90] zu finden sind. In diesem Abschnitt wird daher zunächst geklärt, was wir in dieser Arbeit unter diesem Begriff verstehen. Bevor wir eine Definition für die Modellsynchronisation angeben, klären wir zunächst die allgemeine Bedeutung sowie die Verwendung des Begriffs „Synchronisation“ im Kontext der Informatik.

2.2.1 Bedeutung der Modellsynchronisation

Das Wort „Modellsynchronisation“ setzt sich aus dem Wort „Modell“ und „Synchronisation“ zusammen. Während wir unter einem Modell eine abstrakte Beschreibung eines Systems verstehen, ist die allgemeine Bedeutung des Wortes „Synchronisation“ nicht sofort ersichtlich.

Synchronisation im Allgemeinen Der Wortstamm des Wortes „Synchronisation“ lautet „synchron“ und ist auf die beiden griechischen Begriffe „sýn“ (zusammen mit, gemeinsam, gleichartig) und „chrónos“ (Zeit) zurückzuführen. Im Fremdwörterbuch der Dudenredaktion [Dud06] wird die Bedeutung des Wortes „synchron“ mit „gleichzeitig“ oder „mit gleicher Geschwindigkeit [ab]laufend“ angegeben.

Synchronisation im Kontext der Informatik Im Duden der Informatik [CS06] wird der Begriff der „Synchronisation“ (oder auch „Synchronisierung“) mit „Abstimmung nebenläufiger Vorgänge aufeinander. [...]“ erklärt. In der Informatik wird der Begriff unter anderem im Zusammenhang mit (verteilten) Betriebssystemen und Datenbanken verwendet. Zu den Synchronisationsproblemen in diesen Bereichen zählen beispielsweise die *Uhrensynchronisation*, die *Prozesssynchronisation* und die *Datensynchronisation* [Tan95, KE96].

Bei der *Uhrensynchronisation* geht es darum, logische und/oder physikalische Uhren untereinander abzugleichen. Insbesondere in Echtzeitsystemen, die beispielsweise im Auto zur Steuerung eines Antiblockiersystems oder eines Airbags eingesetzt werden, spielt Zeit eine besonders große Rolle. Ungenauigkeiten bei den Taktfrequenzen der Uhren können zu Abweichungen der Uhren sowohl von der realen Zeit als auch zu Abweichungen der Uhren untereinander führen und damit die fehlerfreie Funktion eines solchen Systems gefährden. Diese Abweichungen werden durch die Uhrensynchronisation wieder ausgeglichen.

Die *Prozesssynchronisation* wird zur Koordination zeitlicher Abläufe von Prozessen eingesetzt. Die zeitliche Abstimmung nebenläufiger Prozesse dient beispielsweise der Interprozesskommunikation, also dem Austausch von Nachrichten zwischen den Prozessen. Als Prozesssynchronisation wird aber auch die Koordination von Zugriffen auf ein gemeinsam genutztes Betriebsmittel, wie zum Beispiel eine Datei, bezeichnet. Hierbei gilt es, einen gleichzeitigen Zugriff auf das Betriebsmittel zu verhindern, beispielsweise um die Konsistenz einer Datei sicherzustellen. Ein bekanntes Verfahren hierfür ist

der *wechselseitige Ausschluss*, bei dem eine Zugriffsreihenfolge für die konkurrierenden Prozesse festgelegt wird.

Bei der *Datensynchronisation* hingegen geht es darum, Daten, die zum Beispiel durch Replikation[6] auf verschiedenen, voneinander unabhängigen Systemen hinterlegt und dort modifiziert wurden, wieder miteinander abzugleichen [SS05]. Die Datensynchronisation kann auf der Datei-, der Datenbank- oder der Applikationsebene stattfinden. Ein Beispiel für die Datensynchronisation auf der Applikationsebene ist eine Anwendung zur Verwaltung eines Terminkalenders. Hier möchte man häufig die Termine eines stationären Geräts auf einem mobilen Endgerät verfügbar machen, damit der Benutzer auf dem mobilen Gerät einen lokalen Zugriff auf seine Termindaten hat. Werden die Daten auf dem mobilen und/oder stationären Endgerät modifiziert, entstehen unterschiedliche Datenbestände. Bei der Datensynchronisation werden diese Daten zu einem bestimmten Zeitpunkt wieder miteinander abgeglichen, so dass die Daten auf den verschiedenen Systemen wieder identisch sind. Die Datensynchronisation wird daher häufig auch als *Datenabgleich* bezeichnet.

An den drei hier vorgestellten Synchronisationsaufgaben ist ersichtlich, dass mit „Synchronisation“ in den unterschiedlichen Bereichen unterschiedliche Ziele verfolgt werden, die nicht immer mit den oben genannten Definitionen übereinstimmen. Während beispielsweise die Uhrensynchronisation der ursprünglichen Definition von „Synchronisation“ ziemlich genau entspricht und die Uhren tatsächlich „gleichzeitig“ voranschreiten beziehungsweise „mit gleicher Geschwindigkeit laufen“ sollen, hat die Prozesssynchronisation mit „gleichzeitig“ wenig gemeinsam. Hier geht es sogar darum, eine bestimmte Reihenfolge festzulegen um einen „gleichzeitigen“ Zugriff zu verhindern. Hier passt eher die Definition aus [CS06], in der die Synchronisation eingesetzt wird, um eine „Abstimmung nebenläufiger Vorgänge aufeinander“ zu erreichen. Die vorgestellten Definitionen treffen jedoch am wenigsten auf die Datensynchronisation zu. Eine treffendere Definition liefert [Wik07]:

> *Data synchronization is the process of establishing consistency among data on remote sources and the continuous harmonization of the data over time. [...]*

Letztlich sind Modelle auch Daten, so dass die Modellsynchronisation am ehesten mit der Datensynchronisation vergleichbar ist. Daher wird die Modellsynchronisation umgangssprachlich auch als *Modellabgleich* bezeichnet.

[6] Replikation bezeichnet in der Datenverarbeitung die mehrfache Speicherung von Daten an unterschiedlichen Orten.

Allerdings ist der Konsistenzbegriff der Datensynchronisation nicht so einfach auf die Modellsynchronisation übertragbar. Bei der Datensynchronisation sind die Daten konsistent, wenn sie gleich sind. Bei der Modellsynchronisation müssen die Modelle, die miteinander „abgeglichen“ werden, weder vor noch nach der Synchronisation tatsächlich gleich sein.[7] Das liegt daran, dass in den meisten Fällen bereits die zugrundeliegenden Formalismen der zu synchronisierenden Modelle voneinander verschieden sind.[8] Daher untersuchen wir zunächst, was Konsistenz im Zusammenhang mit Modellen bedeutet.

2.2.2 Zusammenhang zwischen Modellkonsistenz und Modellsynchronisation

Der Begriff der *Konsistenz* „[...] wird in verschiedenen Teilgebieten der Informatik unterschiedlich benutzt. Man bezeichnet Aussagen, Formeln, Modelle oder Systeme als konsistent, wenn sie 'in sich stimmig' sind, wenn sie also keinen Unsinn ergeben, keine Widersprüche enthalten, mit der Realität im Einklang stehen, bzw. keine undefinierten Zustände annehmen können. [...]“ [CS06]. In der Logik bedeutet Konsistenz beispielsweise, dass ein zugrundeliegendes axiomatische System wiederspruchsfrei ist. Im Zusammenhang mit Datenbanken werden Daten als konsistent angesehen, wenn zuvor definierte Konsistenzbedingungen (engl. constraints) eingehalten werden.

In der modellbasierten Softwareentwicklung sind Modelle *konsistent*, wenn der Informationsgehalt der Modelle bezüglich des modellierten Softwaresystems keine Widersprüche enthält. Dementsprechend wird ein Widerspruch als *Inkonsistenz* und die Modelle als *inkonsistent* bezeichnet. Wird die Konsistenz verletzt, so liegt ein Problem vor, das häufig auch als Konsistenzproblem bezeichnet wird.

In der modellbasierten Softwareentwicklung existieren verschiedene Arten von Konsistenzproblemen, wie zum Beispiel syntaktische und semantische Konsistenzprobleme, Konsistenzprobleme innerhalb eines einzigen Modells sowie Konsistenzprobleme, die zwischen mehreren Modellen auftreten können [EKHG01]. Für viele dieser Konsistenzprobleme existieren in der

[7] Allerdings ist eine Modellsynchronisation zur Herstellung identischer Modelle auch denkbar, zum Beispiel im Kontext von Versions- und Konfigurationssystemen.

[8] Dies ist nicht zwangsläufig so. Ein Beispiel, in dem der zugrunde liegende Formalismus für beide zu synchronisierenden Modelle gleich ist und nur die Modelle sich unterscheiden, findet sich in der Diplomarbeit von Patrick Könemann [Kön07].

Literatur verschiedene Ansätze zur Überprüfung und Erhaltung der Konsistenz. Die meisten dieser Ansätze betrachten aber nur ein Konsistenzproblem und bieten nur hierfür eine Lösung an, d.h., es existiert weder eine formale und allgemein gültige Definition der Modellkonsistenz, noch existiert ein Ansatz, der alle Konsistenzprobleme löst [Küs04a]. Auch in dieser Arbeit werden wir weder eine Definition noch einen allgemeingültigen Ansatz zur Überprüfung und Gewährleistung der Modellkonsistenz angeben. In dieser Arbeit gehen wir vielmehr von der Annahme aus, dass die Modelle für sich betrachtet konsistent sind und fokussieren lediglich auf die Konsistenz zwischen mehreren Modellen.

Eine notwendige Bedingung zur Entstehung von Inkonsistenzen zwischen Modellen ist, dass die Modelle sich in ihrem Informationsgehalt überlappen. Die Überlappungen setzen die Modellelemente zueinander in Beziehung. Im Folgenden nennen wir diese Beziehung *Korrespondenzbeziehung* und eine Regel, die beschreibt, welche Modellelemente und unter welchen Bedingungen diese Modellelemente zueinander korrespondieren, bezeichnen wir als *Korrespondenzregel*. Eine Korrespondenzregeln beschreibt, welche Modellelemente und unter welchen Bedingungen diese Modellelemente zueinander konsistent sind, d. h., wir betrachten zwei Modelle bezüglich der spezifizierten Korrespondenzregeln als zueinander konsistent, wenn die Korrespondenzbeziehungen eingehalten werden.

Während der Bearbeitung der Modelle können die Korrespondenzbeziehungen verletzt und die Modelle zueinander inkonsistent werden. In einigen Fällen lässt sich eine verletzte Korrespondenzbeziehung automatisch wiederherstellen, so dass die damit verbundene Inkonsistenz zwischen den Modellen behoben wird. Eine solche automatische Wiederherstellung der Konsistenz zwischen den Modellen nennen wir Modellsynchronisation.

2.2.3 Definition und Aufgabe der Modellsynchronisation

Auf Grundlage der zuvor durchgeführten Betrachtungen können wir die Modellsynchronisation wie folgt definieren:

> **Definition 2.1** *Die **Modellsynchronisation** beschäftigt sich mit der Erkennung und der Aufrechterhaltung von Korrespondenzbeziehungen zwischen Modellen. Die Grundlage der Modellsynchronisation bilden Korrespondenzregeln, mit denen eine Korrespondenzbeziehung formal beschrieben wird.*

> *Die Modellsynchronisation heißt **inkrementell**, wenn nur die von einer Änderung tatsächlich betroffenen Modellelemente betrachtet werden müssen, um sie miteinander zu synchronisieren.*
>
> *Die Modellsynchronisation nennen wir **partiell**, wenn nicht das gesamte Modell sondern nur ein Teil eines Modells synchronisiert wird, d. h., wenn Teile eines Modells von vornherein von der Synchronisation ausgeschlossen werden.*

Das Ziel der Modellsynchronisation besteht darin, zueinander in Beziehung stehende Modelle während der Entwicklung miteinander abzugleichen, so dass sie bezüglich der spezifizierten Korrespondenzregeln zueinander konsistent bleiben. Aufgrund der Tatsache, dass heutige Softwareentwicklungsprozesse in den meisten Fällen ein inkrementell-iteratives Vorgehensmodell aufweisen, ist es nur natürlich, dass auch Modelle oft geändert werden. Änderungen an den Modellen können aber dazu führen, dass die Korrespondenzbeziehungen zwischen den Modellen verletzt werden. Die Aufgabe der Modellsynchronisation ist daher die *Erkennung* und die *Aufrechterhaltung* der zugrunde liegenden Korrespondenzbeziehungen.

Bei der *Erkennung* der Korrespondenzbeziehungen geht es in erster Linie darum, zueinander korrespondierende Modellelemente auf der Grundlage von Korrespondenzregeln zu identifizieren und die durch die Korrespondenzregeln spezifizierten Bedingungen zu überprüfen. Nachdem zueinander in Beziehung stehende Modellelemente identifiziert worden sind, müssen diese Korrespondenzbeziehungen aufrecht erhalten werden. Ziel der *Aufrechterhaltung* ist es, erkannte Inkonsistenzen bezüglich der Korrespondenzregeln durch geeignete Maßnahmen zu beseitigen, so dass die Modelle bezüglich der definierten Korrespondenzregeln wieder zueinander konsistent sind.

Die Modellsynchronisation kann manuell oder automatisiert durchgeführt werden. Bei großen Modellen ist eine manuelle Synchronisation der Modelle sehr zeitaufwändig. Daher ist eine automatische Modellsynchronisation durch geeignete Werkzeuge vorzuziehen. Die automatische Modellsynchronisation kann batch-artig, das heißt, vollständig in einem Schritt, oder inkrementell durchgeführt werden. Eine vollständige Modellsynchronisation in einem Schritt kann beispielsweise durch Modelltransformationstechniken realisiert werden. Wie im vorangegangenen Abschnitt dargelegt wurde, dürfen dabei die Modelle nicht gleichzeitig geändert werden. Außerdem müssen die Modelle bijektiv aufeinander abbildbar sein. Dies ist jedoch selten gegeben. Daher sollten zur Modellsynchronisation Techniken eingesetzt werden, die Modelle sowohl inkrementell als auch partiell synchronisieren können.

Die hier angegebene Definition macht keine Aussage darüber, wie die Korrespondenzregeln formuliert werden oder wie die Modellsynchronisation technisch umgesetzt wird. Diese sehr allgemeine Form der Definition wurde bewusst so gewählt, um die Auswahl der Spezifikationstechniken für Korrespondenzregeln und der verwendeten Algorithmen nicht unnötig einzuschränken.

2.3 Kriterien der Modellsynchronisation

In diesem Abschnitt stellen wir einige allgemeine Kriterien vor, die zur Klassifikation und Einordnung verschiedener Synchronisationsansätze verwendet werden können. Das Ziel, das wir damit verfolgen, ist einerseits dem Leser einen Überblick über charakteristische Merkmale einer Modellsynchronisation zu verschaffen. Andererseits verwenden wir diese Kriterien, um unseren eigenen Ansatz im nächsten Abschnitt zu klassifizieren.

Ähnliche Kriterien sind zur Klassifikation von Ansätzen zur Modelltransformation aufgestellt [CH03, MG05] und später aktualisiert worden [CH06, MG06]. Einige dieser Kriterien wurden bereits in anderen Beiträgen zur Modellsynchronisation aufgegriffen [IK04b, KC05a, AC07].

2.3.1 Synchronisationsaufgabe und -umgebung

Eine Modellsynchronisation kann aufgrund struktureller Eigenschaften, die durch eine spezielle Synchronisationsaufgabe und Synchronisationsumgebung festgelegt sind, charakterisiert werden. Hierzu zählen die *Anzahl und Topologie der Modelle*, die *Synchronisationsrichtung*, die *Kardinalität von Korrespondenzbeziehungen*, die zugrundeliegende *Modellrepräsentation und Technologie* sowie die *Ebene der Synchronisation*.

Anzahl und Topologie der Modelle Eine erste Unterscheidung der Modellsynchronisation kann anhand der Anzahl der zu synchronisierenden Modelle vorgenommen werden. Hier können wir unterscheiden, ob eine Synchronisation zwischen genau zwei Modellen stattfindet oder ob weitere Modelle an der Synchronisation beteiligt sind. Werden mehrere Modelle miteinander synchronisiert, so kann die Topologie, das heißt, die Anordnung der Modelle, zur Charakterisierung herangezogen werden. In der Abbildung 2.6 sind drei Beispiele für mögliche Synchronisationsbeziehungen zwischen Modellen dargestellt. Das erste Beispiel zeigt eine Modellsynchronisation zwischen genau

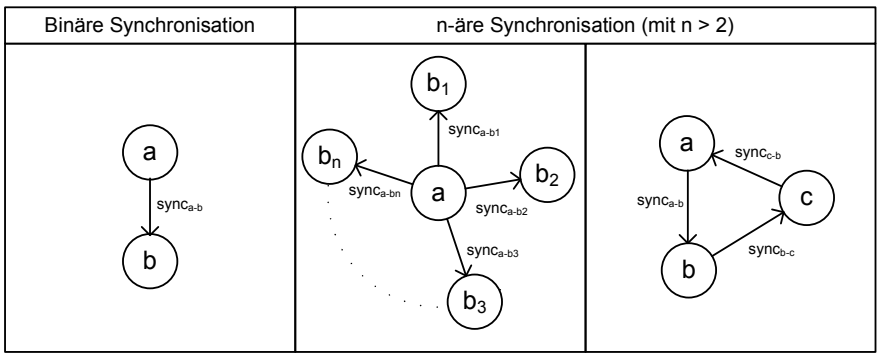


Abbildung 2.6: Beispiele für verschiedene Topologien

zwei Modellen. Im zweiten Beispiel sind die Modelle sternförmig um ein Modell angeordnet, das seine Änderungen an alle anderen Modelle weitergibt und somit diese Modelle zu sich selbst synchronisiert. Das dritte Beispiel zeigt eine zyklische Anordnung der Modelle, in der jedes Modell seine Änderungen an sein nachfolgendes Modell weitergibt.

Natürlich sind weitere Topologien denkbar. Hier kann allgemein unterschieden werden, ob ein Ansatz zur Modellsynchronisation nur *binäre* oder auch *n-äre* Modellsynchronisationen (mit $n > 2$) unterstützt. Falls ein Ansatz eine n-äre Synchronisation unterstützt, ist weiterhin zu unterscheiden, ob er *mit Zyklen* umgehen kann oder nur für eine Modellsynchronisation *ohne Zyklen* geeignet ist.[9]

Synchronisationsrichtung Ein weiteres Unterscheidungskriterium ist die Richtung der Synchronisation. Grundsätzlich können wir zwischen einer *unidirektionalen* und einer *bidirektionalen* Modellsynchronisation unterscheiden. Bei der unidirektionalen Synchronisation werden Änderungen in einem Quellmodell beobachtet und an ein Zielmodell weitergeleitet. Dabei wird während der Synchronisation das Quellmodell als nicht änderbar angesehen und Änderungen nur im Zielmodell zugelassen. Bei einer bidirektionalen Modellsynchronisation werden im Rahmen der Synchronisation Änderungen in beiden Modellen berücksichtigt und in beide Richtungen propagiert.

[9]Bei einer zyklischen Anordnung der Modelle kann es vorkommen, dass eine Modellsynchronisation wiederholt ausgeführt werden muss, um die Modelle miteinander zu synchronisieren. Dabei kann es passieren, dass ein solcher Zyklus bzw. Kreislauf nicht terminiert, sofern der zugrundeliegende Ansatz keine speziellen Maßnahmen zur Erkennung und Auflösung solcher Zyklen bereitstellt.

Kardinalität von Korrespondenzbeziehungen Eine Modellsynchronisation findet zwischen zueinander korrespondierenden Elementen der Modelle statt. Bei diesen Korrespondenzbeziehungen kann es sich um *1-zu-1*, *1-zu-n*, oder *n-zu-m Beziehungen* handeln. Ein Beispiel für 1-zu-1 Beziehungen findet sich in vielen UML-Modellierungswerkzeugen, die eine Modellsynchronisation zwischen einem UML-Klassendiagramm und dazugehörigen Java-Code anbieten. Ein Beispiel für eine 1-zu-n Modellsynchronisation wird in [KC05a] vorgestellt, während zum Beispiel in [KW07] auch n-zu-m Beziehungen verwendet werden.

Modellrepräsentation und Technologie Eine weitere Charakterisierung kann auf Grundlage der Modellrepräsentation und der Technologie vorgenommen werden. Zunächst können wir unterscheiden, ob die zu synchronisierenden Modelle durch einen *gemeinsamen Formalismus* beschrieben sind oder eine Synchronisation zwischen Modellen stattfindet, denen *unterschiedliche Formalismen* zugrunde liegen.

Zusätzlich können sich Modellsynchronisationsaufgaben darin unterscheiden, dass nur Modelle synchronisiert werden, denen eine bestimmte Struktur zugrunde liegt. Zum Beispiel ist es deutlich einfacher, baum-artige, hierarchische Datenstrukturen als graph-basierte Datenstrukturen zu synchronisieren.

Schließlich ist noch zu unterscheiden, ob eine Modellsynchronisation nur mit Modellen umgehen kann, die auf *einer einzigen und fest vorgegebenen Technologie* basieren, oder auch eine Synchronisation zwischen Modellen *verschiedener Technologien* möglich ist. Die Überbrückung verschiedener Technologien erlaubt die Synchronisation zwischen Modellen, die mit unterschiedlichen Werkzeugen erzeugt und bearbeitet werden.

Ebene der Synchronisation Ein grundsätzliches Kriterium ist die Synchronisationsebene. Hier kann zwischen einer *horizontalen* und einer *vertikalen* Modellsynchronisation unterschieden werden.[10] Mit einer horizontalen Modellsynchronisation bezeichnet man eine Synchronisation zwischen Modellen, die sich auf derselben Abstraktionsebene befinden. Die vertikale Modellsynchronisation bezeichnet eine Synchronisation zwischen Modellen unterschiedlicher Abstraktionsebenen. Dieses Kriterium steht in einem engen Zusam-

[10] Die Bezeichnung „vertikale und horizontale Modellsynchronisation“ ist angelehnt an die Begriffe der vertikalen und horizontalen Konsistenz aus [EKHG01] sowie der vertikalen und horizontalen Modelltransformation aus [MG06].

menhang zu den Kriterien der Modellrepräsentation und Technologie, da bei der horizontalen Modellsynchronisation meistens zwischen unterschiedlichen Formalismen synchronisiert wird und bei der vertikalen Modellsynchronisation zwischen Modellen, die auf einem gemeinsamen Formalismus basieren, sich aber auf unterschiedlichen Abstraktionsebenen befinden.

2.3.2 Synchronisationsregeln

Einige Kriterien betreffen die Synchronisationsregeln. Hierzu zählen die *Parametrisierung der Synchronisation durch Regeln*, die *Spezifikation der Regeln* und die *Richtung der Regeln*.

Parametrisierung der Synchronisation Ein wichtiges Unterscheidungsmerkmal der Synchronisation bilden die Regeln der Synchronisation. Die Regeln können *fest vorgegeben* oder *frei spezifizierbar* sein. Bei fest vorgegebenen Regeln ist eine Anpassung der Synchronisation durch einen Benutzer nicht möglich. Sind die Regeln hingegen frei spezifizierbar, so ist die Modellsynchronisation parametrisierbar und kann an spezielle Anforderungen eines Benutzers angepasst werden.

Spezifikation der Regeln Sind die Regeln einer Modellsynchronisation frei spezifizierbar, so können zur Unterscheidung der Synchronisationsansätze Eigenschaften der zugrunde liegenden Spezifikation als weitere Kriterien herangezogen werden.

Zunächst können wir zwischen einer *operationalen* und einer *deklarativen* Spezifikation der Regeln unterscheiden. Bei einer operationalen Spezifikation wird beschrieben, *wie* eine Modellsynchronisation erreicht wird. Im Gegensatz dazu beschreibt eine *deklarative* Spezifikation nur, *was* miteinander synchronisiert werden soll.

Eine weitere Unterscheidung kann anhand der Notation getroffen werden. Bei der Notation der Regeln kann es sich um eine *textuelle* oder um eine *graphische* Notation handeln. Hybride Ansätze, die sich aus graphischen und textuellen Bestandteilen zusammensetzen, sind ebenfalls denkbar.

Schließlich betrifft ein weiteres Kriterium die Korrespondenzbeziehungen zwischen den Modellelementen. Bei einer Regelspezifikation kann die Festlegung von Korrespondenzbeziehungen entweder *implizit* oder *explizit* erfolgen. Bei einer impliziten Festlegung der Korrespondenzbeziehungen müssen die Korrespondenzbeziehungen nicht weiter spezifiziert werden. Dadurch

entfällt zusätzlicher Aufwand für den Entwickler der Regeln. Bei einer expliziten Spezifikation der Korrespondenzbeziehungen ist der Aufwand zwar höher, allerdings können dadurch die Korrespondenzbeziehungen genauer festgelegt werden.

Richtung der Regeln Ein weiteres Kriterium betrifft die Richtung der spezifizierten Synchronisationsregeln. Hierbei kann es ausreichend sein, *eine einzige Regel für alle Richtungen* zu spezifizieren. Alternativ dazu kann aber auch verlangt werden, jeweils *eine eigene Regel für jede Synchronisationsrichtung* zu erstellen. Dementsprechend unterschiedlich kann der Spezifikationsaufwand ausfallen. Zusätzlich ist bei einer separaten Regelspezifikation für jede Richtung die Gefahr groß, dass die Korrespondenzbeziehungen zueinander inkonsistent spezifiziert werden. Hier entsteht also eine potentielle Fehlerquelle, die bei der Spezifikation einer einzigen Regel für alle Richtung nicht gegeben ist.

2.3.3 Synchronisationsverfahren

Weitere Kriterien der Modellsynchronisation beziehen sich auf das zugrundeliegende Synchronisationsverfahren. Hierzu zählen der *Grad der Automatisierung*, die eingesetzte *Synchronisationsstrategie*, der *Zeitpunkt und Häufigkeit der Ausführung*, der *Umgang mit Änderungen und Konflikten* sowie das *Verfahren zur bidirektionalen Synchronisation.*

Grad der Automatisierung Ein grundsätzliches Kriterium der Modellsynchronisation ist der Grad der Automatisierung. Hierbei ist zu unterscheiden, ob eine Modellsynchronisation vollständig automatisch ausgeführt werden kann oder Benutzerinteraktionen möglich sind. Damit eng verbunden ist auch das Verfahren zur Konfliktresolution (siehe Umgang mit Änderungen und Konflikten).

Synchronisationsstrategie Eine entscheidende Rolle bei der Modellsynchronisation spielt die Synchronisationsstrategie. Hierzu zählen die *Identifikation von Änderungen*, *die Skalierbarkeit* und *die Synchronisationsmodi.*

Bei der Identifikation von Änderungen können wir zwischen einer *ereignisorientierten* und einer *zustands-orientierten* Modellsynchronisation unterscheiden. Bei einem ereignis-orientierten Verfahren wird die Modellsynchronisation erreicht, indem Ereignisse, wie zum Beispiel Hinzufügen, Löschen

oder Ändern, im Quellmodell aufgezeichnet, in korrespondierende Ereignisse des Zielmodells transformiert und dort ausgeführt werden. Bei einem zustands-orientierten Verfahren wird der Zustand der Modelle vor und nach durchgeführten Änderungen analysiert und die Modellsynchronisation auf Grundlage der Analyseergebnisse durchgeführt.

Die Skalierbarkeit eines Synchronisationsverfahrens ist wichtig, um auch große Modelle schnell oder zumindest mit einem vertretbaren zeitlichen Aufwand synchronisieren zu können. Eine Modellsynchronisation kann entweder *batch-artig* oder *inkrementell* erfolgen. Eine batch-artige Modellsynchronisation kann beispielsweise durch eine Modelltransformation realisiert werden. Dabei wird das Zielmodell verworfen und ein neues Zielmodell erstellt. Nach der Modelltransformation sind beide Modelle wieder zueinander synchron. Im Gegensatz dazu werden bei der inkrementellen Modellsynchronisation nur Modellelemente betrachtet, die seit der letzten Modellsynchronisation geändert wurden.

Die Modi der Modellsynchronisation haben einen Einfluss auf die möglichen Anwendungsfälle aus der Sicht eines Benutzers. Hierbei können wir zwischen einem *Push-* und einem *Pull-Modus* unterscheiden. Im Push-Modus kann eine Modellsynchronisation durchgeführt werden, indem ein Modell seine Änderungen an die anderen Modelle propagiert. Im Pull-Modus löst ein Modell eine Modellsynchronisation aus, um andere Modelle explizit auf Änderungen zu überprüfen und diese dann zu übernehmen.

Zeitpunkt und Häufigkeit der Ausführung Ein weiteres Unterscheidungskriterium ist der Zeitpunkt und die Häufigkeit der Ausführung einer Modellsynchronisation. Grundsätzlich kann hier zwischen einer *automatischen Ausführung* und einer *Ausführung auf Anforderung* unterschieden werden.

Bei einer automatischen Ausführung existieren mehrere Varianten, wann die Synchronisation ausgelöst wird. Eine Möglichkeit besteht beispielsweise darin, die Modellsynchronisation *nach jeder Änderung* in einem Modell durchzuführen. Eine andere Möglichkeit ist, die Synchronisation *in Zeitintervallen* auszulösen. Die Zeitintervalle können hierbei *fest vorgegeben* oder *variabel* sein. Eine variable Synchronisationsauslösung könnte beispielsweise immer dann erfolgen, wenn der Rechner gerade nicht ausgelastet ist. Hier sind auch weitere Varianten oder Kombinationen denkbar, wie zum Beispiel die Auslösung *nach einer bestimmten (Anzahl von) Änderung(en)* oder *nach bestimmten Ereignissen*, wie zum Beispiel nach dem Speichern eines Modells.

Die automatische Ausführung einer Modellsynchronisation ist nicht immer

vorteilhaft. Insbesondere dann, wenn man inkonsistente Zustände in einem Modell zulassen möchte, ist es oft besser, auf eine automatische Ausführung der Modellsynchronisation zu verzichten und diese nur auf Anforderung durch den Benutzer durchzuführen.

Umgang mit Änderungen und Konflikten Ein Unterscheidungskriterium betrifft den Umgang mit manuellen Änderungen in den beteiligten Modellen. Hierbei kann eine Modellsynchronisation Änderungen *überschreiben* oder *erhalten*. Für Modellelemente, die automatisch im Rahmen einer Modellsynchronisation erstellt worden sind, ist das Überschreiben der Modellelemente nicht so problematisch. Sind die Modellelemente von einem Benutzer explizit erstellt worden, so sollte eine Modellsynchronisation diese Modellelemente so weit es geht erhalten. Dabei kann es von Vorteil sein, auf eine *vollständige* Synchronisation zu verzichten und die Modellsynchronisation *partiell*, das heißt, nur für ganz bestimmte Teile eines Modells, durchzuführen.

Wenn sich die zu synchronisierenden Modelle widersprechen und zur Synchronisation der Modelle mehrere Alternativen vorhanden sind, entstehen Konflikte. Bei der Behandlung der Konflikte spielt der Grad der Automatisierung eine entscheidende Rolle. Aus der Sicht eines Benutzers sollte eine Modellsynchronisation zumindest in allen zweifelsfreien Fällen automatisch erfolgen. Treten hingegen Konflikte während einer Synchronisation auf, so kann die Konfliktresolution *automatisch*, *semi-automatisch* oder *manuell* durchgeführt werden.

Eine vollautomatische Modellsynchronisation kann zum Beispiel aus einer der Alternativen wählen und die Synchronisation durchführen. Die Auswahl kann dabei nicht-deterministisch erfolgen oder im Vorfeld – zum Beispiel durch die Vergabe von Prioritäten – gesteuert werden.

Bei einer semi-automatischen Konfliktresolution könnten mögliche Alternativen einem Benutzer vorgeschlagen werden, der daraus die aus seiner Sicht beste Alternative auswählt. Die eigentliche Durchführung der Konfliktauflösung erfolgt aber automatisch.

Im Gegensatz dazu wird bei der manuellen Konfliktresolution nur die Stelle markiert, an der ein Konflikt aufgetreten ist. Die Behebung dieses Konflikts hingegen wird dem Benutzer überlassen.

Verfahren zur bidirektionalen Synchronisation Anhand des eingesetzten Synchronisationsverfahrens zur bidirektionalen Modellsynchronisation kann eine weitere Unterscheidung getroffen werden. Die bidirektionale Modell-

synchronisation kann durch *zwei unidirektionale Modellsynchronisationen* in jeweils entgegengesetzter Synchronisationsrichtung realisiert werden.[11] Eine Alternative hierzu ist eine bidirektionale Synchronisation, die *in einem einzigen Durchlauf* die beteiligten Modelle in beide Richtungen abgleicht.

2.4 Methodischer Ansatz

In diesem Abschnitt stellen wir den methodischen Ansatz der vorliegenden Arbeit vor. Hierzu erläutern wir zunächst die Ausgangslage der Arbeit. Anschließend präsentieren wir unseren Ansatz und die in dieser Arbeit entwickelte Methode zur Erstellung von Werkzeugen zur Modellsynchronisation. Anschließend ordnen wir unseren Ansatz gemäß der im vorangegangenen Abschnitt entwickelten Kriterien ein.

2.4.1 Ausgangslage und Anforderungen

Am Fachgebiet für Softwaretechnik an der Universität Paderborn wurde die modellbasierte Softwareentwicklungsumgebung Fujaba[12] implementiert [FNTZ98]. Seitdem wird Fujaba erfolgreich in verschiedenen modellgetriebenen Softwareentwicklungsprojekten eingesetzt. Die formale Grundlage von Fujaba bildet ein Graphersetzungssystem [Zün01], auf dessen Basis ausführbarer Code aus Diagrammen der Unified Modeling Language (UML) [UML05] automatisch erzeugt werden kann. Zusätzlich erlaubt Fujaba ein sogenanntes „Round-Trip Engineering“ zwischen den Diagrammen und dem generierten Code, so dass sowohl eine Bearbeitung der Modelle als auch eine Bearbeitung des generierten Codes möglich ist.

Eine Besonderheit der Entwicklungsumgebung bildet ein Erweiterungsmechanismus, über den sogenannte Plug-ins dynamisch zu Fujaba hinzugefügt werden können [BGN+04]. Dieser Erweiterungsmechanismus erlaubt es unter anderem, verschiedene Modellierungssprachen nachträglich in Fujaba zu integrieren und auf diese Art und Weise domänenspezifische Sprachen (engl. Domain Specific Languages , kurz DSL) für unterschiedliche Anwendungsdomänen zu realisieren.

[11] Im Rahmen der Datensynchronisation wird dieses Verfahren als *Zwei-Wege-Synchronisation* bezeichnet. Eine unidirektionale Datensynchronisation hingegen wird *Ein-Wege-Synchronisation* genannt.

[12] Fujaba ist ein Akronym für „From UML to Java and Back Again“

Das ursprüngliche Ziel dieser Arbeit bestand darin, einen Ansatz zur Modelltransformation in FUJABA zu integrieren, um beispielsweise eine automatische Übersetzung zwischen Blockdiagrammen und Klassendiagrammen zu ermöglichen. Bereits während der Umsetzung hat sich jedoch gezeigt, dass eine einfache Modelltransformation nur bedingt hilfreich ist. Schnell entstand der Wunsch, die Konsistenz zwischen den Modellen auch nach durchgeführten Änderungen an den Modellen und einer bereits erfolgten Modelltransformation möglichst automatisch sicherzustellen. Die Modelle sollten automatisch miteinander synchronisiert, also zueinander konsistent gehalten werden, um eine unabhängige Bearbeitung der Modelle zu ermöglichen. Aufgrund der Tatsache, dass auch Code als ein Modell des implementierten Systems aufgefasst werden kann, lag es nahe, einen einzigen und durchgängigen Ansatz zu erarbeiten, mit dem sowohl Modelle untereinander als auch ein Modell mit dem daraus generierten Code synchronisiert werden kann.

Natürlich ist es prinzipiell möglich eine Modellsynchronisation zwischen Modellen – oder wie im Fall des in FUJABA implementierten „Round-Trip Engineering“ eine Synchronisation zwischen Modell und Code – von Hand zu programmieren und fest in ein Werkzeug einzubauen. Allerdings hat die manuelle Implementierung solcher Synchronisationswerkzeuge zwei wesentliche Nachteile. Zunächst sind die Korrespondenzregeln fest in das Werkzeug codiert. Dadurch kann eine bereits umgesetzte Modellsynchronisation nur mit einem hohen Aufwand um neue Korrespondenzregeln erweitert oder bereits vorhandene Korrespondenzregeln an neue Bedürfnisse angepasst werden. Insbesondere bei einer Synchronisation von Modell und Code sind Anpassungen der Synchronisation an firmen-, domänen- oder projektspezifische Vorgaben nicht unüblich. Der zweite wesentliche Nachteil einer manuellen Implementierung ist durch die Komplexität der Aufgabe an und für sich gegeben. Dazu tragen häufig die Größe und die Komplexität der zugrundeliegenden Metamodelle sowie die Korrespondenzbeziehungen bei. Zusammen mit den benötigten Kenntnissen zu technischen und technologischen Details muss man feststellen, dass die Entwicklung eines Werkzeugs zur Modellsynchronisation an und für sich kompliziert und die Programmierung von Hand sehr zeitaufwändig ist.

Die Zielsetzung dieser Arbeit besteht daher darin, eine Methode zu erarbeiten und entsprechende Werkzeuge zu entwickeln, die einen Softwareentwickler bei der Erstellung von Werkzeugen zur Modellsynchronisation unterstützen. Die Unterstützung durch automatisierte Werkzeuge soll den Softwareentwickler von der Komplexität einer manuellen Entwicklung der Werkzeuge befreien, die zur Entwicklung benötigte Zeit reduzieren und da-

mit die Entwicklungskosten senken. Dabei sollte der methodische Ansatz die folgenden Anforderungen erfüllen:

Funktionalität Neben einer bidirektionalen Modellsynchronisation soll der Ansatz auch zur Modelltransformation geeignet sein. Aufgrund der Annahme, dass Modelle unabhängig voneinander bearbeitet werden können, ohne dass sie sofort und ständig miteinander synchronisiert werden, muss es mit dem Ansatz zusätzlich möglich sein, Korrespondenzbeziehungen auch im Nachhinein zu überprüfen.

Anpassbarkeit Modelle können in verschiedenen Anwendungsdomänen oder sogar in verschiedenen Projekten einer Anwendungsdomäne sehr unterschiedlich genutzt werden. Insbesondere die Abbildung eines Modells auf andere Modelle kann – je nach späterem Verwendungszweck – sehr unterschiedlich ausfallen. Daher ist es notwendig, dass die Korrespondenzbeziehungen zwischen den Modellen an verschiedene Bedürfnisse leicht anpassbar sind. Dazu dürfen die Korrespondenzregeln nicht fest im Werkzeug codiert sein. Stattdessen muss das Werkzeug zur Modellsynchronisation durch Korrespondenzregeln parametrisierbar sein.

Skalierbarkeit Die modellbasierte Softwareentwicklung wird hauptsächlich eingesetzt, um große und komplexe Softwaresysteme zu entwickeln. Der Hauptnutzen der automatischen Modellsynchronisation liegt damit bei der Synchronisation großer Modelle. Daher muss die automatische Modellsynchronisation sehr gut skalieren und auch die Synchronisation großer Modelle mit vertretbarem Aufwand ermöglichen.

Interoperabilität Die Modellsynchronisation soll nicht nur auf Fujaba beschränkt bleiben. Häufig ist es notwendig Modelle zu transformieren oder zu synchronisieren, die mit anderen Werkzeugen bearbeitet werden. Bei einer Erweiterung bereits existierender Modellierungswerkzeuge kann in der Regel kein Einfluss auf die verwendeten Technologien genommen werden. Die vorhandenen Möglichkeiten und Schnittstellen erfüllen nur selten alle Anforderungen an die gewünschten Eigenschaften der zu entwickelnden Modellsynchronisation, so dass hier häufig Einschränkungen und Kompromisse gemacht werden müssen. Unser Ansatz soll auch solche Umstände berücksichtigen und trotz eventuell vorhandener Einschränkungen leicht in andere Werkzeuge integrierbar sein.

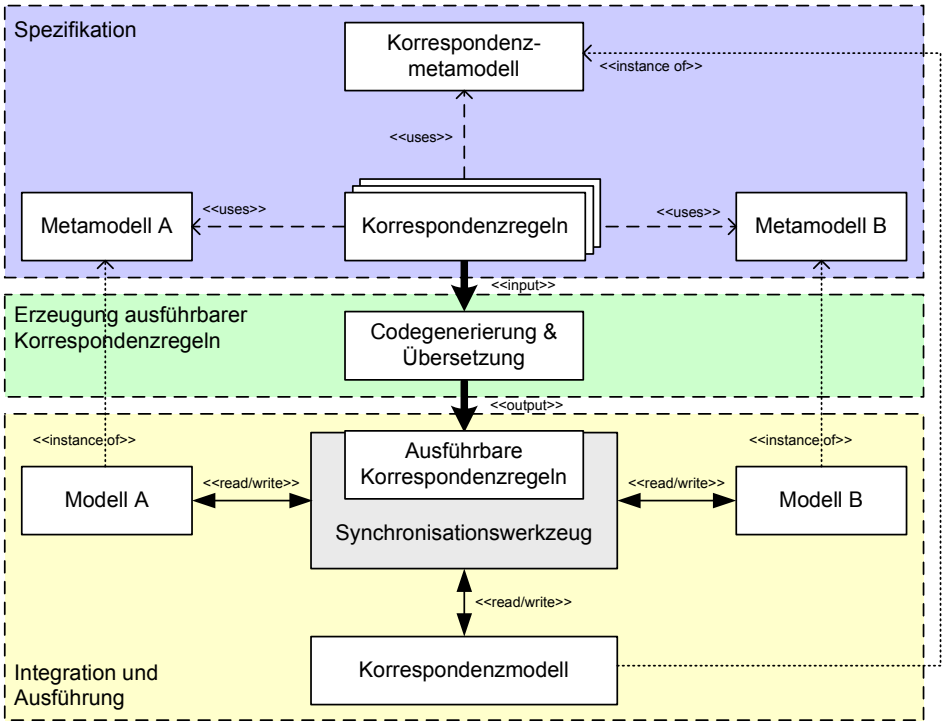


Abbildung 2.7: Überblick zur Methode

Bei den hier genannten Anforderungen handelt es sich um ganz wesentliche Anforderungen, die der Ansatz erfüllen sollte. Wie wir im weiteren Verlauf dieser Arbeit noch sehen werden, besitzt der in dieser Arbeit verfolgte Ansatz darüber hinaus weitere Merkmale, die im Rahmen einer Modellsynchronisation besonders positiv auffallen.

2.4.2 Überblick über die Methode

Nachdem wir im vorangegangenen Abschnitt die Ausgangslage der vorliegenden Arbeit erläutert haben, geben wir in diesem Abschnitt einen Überblick über unseren Ansatz und die erarbeitete Methode, mit der Werkzeuge zur Modellsynchronisation weitestgehend automatisiert entwickelt werden können. Die Methode ist in Abbildung 2.7 dargestellt. Im Folgenden erläutern wir die einzelnen Schritte der Methode.

Spezifikation

Zur Spezifikation der Korrespondenzregeln verwenden wir den Ansatz der Tripel-Graph-Grammatiken (TGGs) [Sch94]. Bei dieser formalen und deklarativen Spezifikationstechnik handelt es sich um einen Ansatz, der bereits erfolgreich zur Spezifikation einer Modelltransformation und Modellintegration eingesetzt wurde. Auf dieser Grundlage können Korrespondenzbeziehungen überprüft, hergestellt und aufrecht erhalten werden. Wie in dieser Arbeit noch gezeigt wird, eignen sich die deklarativ spezifizierten Korrespondenzbeziehungen damit auch zur Realisierung von bidirektional und inkrementell arbeitenden Modellsynchronisationswerkzeugen. Durch die graphische Notation dieser Spezifikationstechnik können die Korrespondenzregeln verständlich beschrieben und später leicht an geänderte Anforderungen angepasst werden. Eine detaillierte Beschreibung dieser Spezifikationstechnik gibt Kapitel 3.

Der hier verwendete Ansatz kann auch zur Codegenerierung und der anschließenden Synchronisation eines Modells mit dem daraus generierten Code verwendet werden. Allerdings kann aufgrund der Größe und Komplexität der abstrakten Syntax einer textuellen Programmiersprache die Spezifikation der benötigten Korrespondenzregeln sehr aufwändig werden. Daher haben wir unsere Methode um eine weitere Möglichkeit zur Spezifikation von Korrespondenzregeln erweitert. Bei diesem Ansatz gibt der Entwickler zueinander korrespondierende Beispiele vor, aus denen dann TGG-Regeln weitestgehend automatisch synthetisiert werden. Bei den Beispielen handelt es sich um zueinander korrespondierende Modelle, die in ihrer konkreten Syntax angegeben werden. Ebenso kann hier aber auch ein Modell und dazu korrespondierender Code verwendet werden. Dieser Ansatz vereinfacht die Spezifikation der benötigten Korrespondenzregeln signifikant – auch wenn in einigen Fällen eine Nachbearbeitung von Hand noch nötig ist. Diesen Ansatz sowie seine Einschränkungen erläutern wir in Kapitel 4.

Bevor die Korrespondenzregeln durch einen Entwickler spezifiziert werden können, müssen zunächst die Metamodelle der beteiligten Modellierungssprachen definiert werden. Zusätzlich muss ein Metamodell für ein sogenanntes Korrespondenzmodell spezifiziert werden. Das Korrespondenzmodell verwaltet explizit die in einer Korrespondenzbeziehung stehenden Modellelemente der zu synchronisierenden Modelle. In unserem Ansatz verwenden wir zur Beschreibung der Metamodelle einfache UML-Klassendiagramme. Die Spezifikation der Metamodelle und der Korrespondenzregeln erfolgt in der Entwicklungsumgebung Fujaba, die in Kapitel 7 näher vorgestellt wird.

Generierung

Um die spezifizierten TGG-Regeln im Rahmen einer Modellsynchronisation nutzen zu können, werden die TGG-Regeln in ausführbaren Code übersetzt. Alternativ hierzu können TGG-Regeln auch interpretativ ausgeführt werden. In dieser Arbeit wird jedoch ein generativer Ansatz verfolgt. Einen TGG-Interpreter haben wir in [KRW04, KW07] vorgestellt.

Bei dem generativen Ansatz dieser Arbeit werden die TGG-Regeln zunächst in operationale Regeln übersetzt. Hierbei wird für jede Richtung der Synchronisation eine eigene Regel erzeugt. Zusätzlich generieren wir eine operationale Regel, die der Überprüfung von Korrespondenzbeziehungen dient und lediglich korrespondierende Modellelemente zueinander zuordnet. Nach der automatischen Übersetzung der TGG-Regeln in operationale Regeln enthalten die operationalen Regeln die regelspezifischen Synchronisationsoperationen.

Bei den in dieser Arbeit verwendeten operationalen Regeln handelt es sich um sogenannte Storydiagramme [FNTZ98]. Storydiagramme sind eine erweiterte Form von UML-Aktivitätsdiagrammen, die zur Beschreibung eines Kontrollflusses zwischen einzelnen Aktivitäten einer Methode verwendet werden. In die Aktivitäten eines Storydiagramms können Graphgrammatikregeln eingebettet werden. Ebenso können die Aktivitäten aber auch Code enthalten. Storydiagramme werden wir noch in Kapitel 5 genauer kennen lernen. Nach der Übersetzung in Storydiagramme nutzen wir den in Fujaba integrierten Codegenerator, um aus den Storydiagrammen ausführbaren Java-Code zu generieren.

Integration und Ausführung

Nachdem der Java-Code erstellt und kompiliert worden ist, kann er genutzt werden, um eine Modellsynchronisation gemäß der spezifizierten Korrespondenzregeln durchzuführen. Zu diesem Zweck existiert eine Softwarebibliothek, die die benötigten Algorithmen zur inkrementellen Modellsynchronisation bereit stellt. Diese Bibliothek ist bereits in Fujaba integriert. Sie kann mit Hilfe einer separaten Benutzerschnittstelle genutzt werden, um verschiedene Modellsynchronisationsaufgaben durchzuführen. Sofern es sich bei den zu synchronisierenden Modellen um Modelle der Fujaba-Entwicklungsumgebung handelt, sind keine weiteren Schritte zur Integration notwendig. Dies gilt insbesondere auch für Modellierungssprachen, die über den Erweiterungsmechanismus nachträglich zu Fujaba hinzugefügt wurden.

Die in dieser Arbeit vorgestellte Methode kann auch verwendet werden, um eine Modellsynchronisation in andere Modellierungswerkzeuge zu integrieren. Bei einer Erweiterung bereits existierender Modellierungswerkzeuge kann in der Regel kein Einfluss auf die verwendeten Technologien genommen werden. Daher müssen oft unterschiedliche, werkzeugspezifische Technologien überbrückt werden. Leider erfüllen die vorhandenen Möglichkeiten und Schnittstellen häufig nicht alle Anforderungen an die gewünschten Eigenschaften der zu entwickelnden Modellsynchronisation, so dass hier in den meisten Fällen Einschränkungen und Kompromisse gemacht werden müssen.

Bei unserem Ansatz werden solche Umstände berücksichtigt und auch in diesen schwierigen Fällen eine Modellsynchronisation ermöglicht – auch wenn dann nicht alle Vorteile des Ansatzes ausgeschöpft werden können. Die in dieser Arbeit vorgestellte Methode ist somit nicht auf FUJABA beschränkt. Vielmehr wird der in FUJABA umgesetzte Ansatz genutzt, um Synchronisationswerkzeuge für andere Entwicklungsumgebungen mit FUJABA modellbasiert zu entwickeln.

Hierzu sind im Wesentlichen zwei Schritte notwendig. Zunächst muss die Softwarebibliothek mit den Algorithmen in das Modellierungswerkzeug integriert sowie eine werkzeugspezifische Benutzerschnittstelle erstellt werden. Anschließend müssen für die spezifizierten Metamodelle entsprechende Modelladapter implementiert werden. Diese sind notwendig, um auf die Modelle im Werkzeug zugreifen zu können.

In einigen Fällen sind die werkzeugspezifischen Metamodelle nicht dokumentiert. Sie müssen dann zum Beispiel aus der API[13]-Dokumentation ermittelt werden. Daher sind häufig die Erstellung der Adapter und die Spezifikation der Metamodelle eng miteinander verbunden. Auch hier kann FUJABA, zum Beispiel durch das integrierte Reverse Engineering, behilflich sein. Auf die Erstellung von Modelladaptern gehen wir in Kapitel 7 noch genauer ein. Einige Besonderheiten für Adapter, die im Rahmen der Synchronisation von Modell und Code benötigt werden, erläutern wir in Kapitel 4. Die in der Softwarebibliothek implementierten Algorithmen zur Modellsynchronisation hingegen werden in Kapitel 5 vorgestellt.

2.4.3 Einordung

In diesem Abschnitt ordnen wir unseren Ansatz anhand der zuvor vorgestellten Kriterien ein. Die Einordnung ist in Tabelle 2.1 zusammengefasst.

[13] Application Programming Interface

Synchronisationsaufgabe und -umgebung	
Anzahl und Topologie der Modelle	zwei Modelle
	n-äre Synchronisation
	ohne Zyklen
Synchronisationsrichtung	bidirektional
Kardinalität von Korrespondenzbeziehungen	n-zu-m
Modellrepräsentation und Technologie	unterschiedliche Formalismen
	graph-basierte Strukturen
	verschiedene Technologien
Ebene der Synchronisation	vertikal und horizontal
Synchronisationsregeln	
Parametrisierung der Synchronisation	frei spezifizierbar
Spezifikation der Regeln	deklarativ
	graphisch
	explizit
Richtung der Regeln	eine Regel für beide Richtungen
Synchronisationsverfahren	
Grad der Automatisierung	automatisch
Synchronisationsstrategie	zustands-orientiert
	batch-artig und inkrementell
	Push- und Pull-Modus
Zeitpunkt und Häufigkeit der Ausführung	Ausführung auf Anforderung (automatische Ausführung jedoch möglich)
Umgang mit Änderungen und Konflikten	erhaltend automatische Konfliktresolution
Verfahren zur bidirektionalen Synchronisation	zwei unidirektionale Modellsynchronisationen

Tabelle 2.1: Einordnung anhand der Kriterien aus Abschnitt 2.3

Der Ansatz dieser Arbeit ist für Synchronisationsaufgaben geeignet, bei denen zwei Modelle miteinander synchronisiert werden sollen. Allerdings können einzelne Synchronisationsaufgaben zu n-ären Synchronisationstopologien kombiniert werden. In diesem Fall ist unbedingt darauf zu achten, dass die Synchronisationsregeln sich nicht widersprechen, da bei der Synchronisation auftretende Zyklen nicht automatisch erkannt werden, was dazu führen kann, dass die Synchronisation nicht mehr terminiert.

Mit unserem Ansatz kann eine bidirektionale Modellsynchronisation durchgeführt werden. Dies ist jedoch nicht zwingend notwendig. Ebenso kann die Synchronisation in nur eine Richtung ausgeführt werden. Zusätzlich erlaubt der Ansatz m-zu-n Korrespondenzbeziehungen zwischen den Modellelementen. Der Ansatz ist weder auf einen speziellen Formalismus spezialisiert noch wird zwischen horizontalen und vertikalen Synchronisationsaufgaben unterschieden. Der in dieser Arbeit verwendete Ansatz unterstützt graph-basierte Strukturen und kann damit auch mit baum-artig strukturierten Modellen umgehen. Allerdings ist er nicht auf baum-artige Modelle optimiert. Über Adapter werden unterschiedliche Technologien unterstützt.

Die Synchronisationsregeln werden deklarativ und explizit in einer graphischen Notation spezifiziert. Mit diesen Regeln kann eine Synchronisation parametrisiert werden. Dabei ist es für eine bidirektionale Modellsynchronisation ausreichend, eine Korrespondenzbeziehung mit nur einer einzigen Korrespondenzregel zu beschreiben – eine separate Regel für jede Richtung ist nicht notwendig.

Das zugrundeliegende Synchronisationsverfahren arbeitet automatisch. Es arbeitet zustands-orientiert und ist somit nicht auf Änderungsereignisse angewiesen. Die Modellsynchronisation kann batch-artig, das heißt, in einem Schritt, ausgeführt werden. Der Ansatz führt bei einer batch-artigen Ausführung notwendige Änderungen im Zielmodell bereits inkrementell durch. Werden außerdem Änderungen im Ausgangsmodell in Form von Ereignissen gemeldet, so kann die Synchronisation direkt auf den geänderten Modellelementen starten und vollständig inkrementell ablaufen. Die benötigte Zeit für eine Modellsynchronisation wird dadurch stark reduziert, so dass eine Synchronisation großer Modelle mit vertretbarem Aufwand möglich ist.

Die Synchronisation kann automatisch nach jeder Änderung ausgeführt werden. In einigen Fällen kann es sinnvoll sein, erst nach einer gewissen Anzahl von Änderungen oder nach bestimmten Ereignissen eine Synchronisation zu starten. Eine automatische Ausführung kann allerdings zu Problemen führen, wenn bei der Bearbeitung temporär Inkonsistenzen in den Modellen

erlaubt sind. Aufgrund der Tatsache, dass keine allgemeingültige, optimale Strategie für den Zeitpunkt und die Häufigkeit einer Synchronisation existiert und die Strategie von den eingesetzten Modellierungswerkzeugen und dem Synchronisationsszenario abhängt, werden beide Ausführungsvarianten in FUJABA unterstützt, wobei die Synchronisation dabei sowohl im Push- als auch im Pull-Modus stattfinden kann.

Modellelemente, die manuell vom Benutzer geändert werden und nicht an einer Korrespondenzbeziehung teilnehmen, werden nicht überschrieben, sondern bleiben erhalten. Hingegen werden Änderungen an Modellelementen, die an einer Korrespondenzbeziehung beteiligt sind und in Konflikt zu anderen Änderungen stehen, automatisch aufgelöst. Hierbei gibt die Richtung der Synchronisation vor, welche Änderungen übernommen und welche Änderungen überschrieben werden. Bei einer bidirektionalen Synchronisation, die in unserem Ansatz durch zwei einzelne, unidirektionale Synchronisationen erreicht wird, ist hierfür die Richtung maßgeblich, mit der gestartet wird. In dem Fall, dass zwei Regeln zueinander in Konflikt stehen, das heißt, beide ausgeführt werden können, wird ein Konflikt dadurch aufgelöst, dass die in der Konfigurationsdatei festgelegte Reihenfolge der Regeln beachtet wird und die dort zuerst genannte Regel ausgeführt wird.

2.5 Zusammenfassung

In diesem Kapitel wurde ein Überblick zur Modellsynchronisation gegeben. Hierzu haben wir in Abschnitt 2.1 eine Modellsynchronisation und die damit verbundenen Probleme an einem Beispiel betrachtet. Das dort vorgestellte Beispiel ist aus Präsentationsgründen recht übersichtlich gestaltet, so dass eine manuelle Modellsynchronisation ohne Werkzeugunterstützung in diesem konkreten Fall durchaus denkbar und vertretbar ist. Allerdings steigt mit der Größe der Modelle auch der Bedarf für eine werkzeuggestützte, automatische Modellsynchronisation.

In Abschnitt 2.2 haben wir uns mit dem Begriff der „Modellsynchronisation“ beschäftigt. Wir haben dort den Zusammenhang zu anderen Bereichen der Informatik aufgezeigt, die sich ebenfalls mit Formen der „Synchronisation“ beschäftigen. Anschließend haben wir definiert, was wir in dieser Arbeit unter einer „Modellsynchronisation“ verstehen und die Aufgabe einer „Modellsynchronisation“ näher erläutert.

Der Abschnitt 2.3 hingegen war Kriterien der Modellsynchronisation gewidmet. Mit den dort aufgestellten Kriterien wurden drei Ziele verfolgt. Das

erste Ziel bestand darin, dem Leser unterschiedliche Möglichkeiten zur Modellsynchronisation und deren Eigenschaften aufzuzeigen. Das zweite Ziel bestand darin, eine Grundlage zu schaffen, um verschiedene Ansätze zur Modellsynchronisation besser miteinander vergleichen zu können. Das dritte Ziel war, eine Einordnung unseres Ansatzes anhand der aufgestellten Kriterien zu ermöglichen und damit dem Leser einen besseren Überblick über die Leistungsfähigkeit des in dieser Arbeit vorgestellten Ansatzes zu vermitteln.

Den Ansatz dieser Arbeit haben wir in Abschnitt 2.4 vorgestellt. Der Ansatz setzt sich aus einer formalen Technik zur Spezifikation von Korrespondenzregeln und einer Methode mit dazugehörigen Werkzeugen zusammen, mit denen Modellsynchronisationswerkzeuge weitestgehend automatisch entwickelt werden können. Im letzten Teil dieses Abschnitts haben wir den Ansatz anhand der zuvor aufgestellten Kriterien eingeordnet.

Das Ziel dieses Kapitels bestand darin, dem Leser einen ersten Eindruck und Überblick zur Modellsynchronisation ganz allgemein und zu dem in dieser Arbeit verfolgten Ansatz zu vermitteln. Die nachfolgenden Kapitel stellen den Ansatz im Detail vor.

Kapitel 3

Spezifikation von Korrespondenzregeln

In diesem Kapitel stellen wir die Spezifikation von Korrespondenzregeln vor. Hierzu setzen wir Tripel-Graph-Grammatiken ein. Diese formale, deklarative und graphische Spezifikationstechnik bildet die Basis der Modellsynchronisation. Bevor wir uns allerdings mit dieser Spezifikationstechnik beschäftigen, gehen wir zunächst auf einige Grundlagen ein, die zum Verständnis benötigt werden.

3.1 Grundlagen

In diesem Abschnitt befassen wir uns zuerst mit Modellen und Metamodellen. Anschließend stellen wir das Prinzip von Graphgrammatiken und die damit verbundene Graphersetzung vor.

3.1.1 Modelle und Metamodelle

Die wichtigsten Artefakte der modellbasierten Softwareentwicklung sind Modelle. Ein Modell basiert auf einem *Formalismus*, der die *Syntax* und die *Semantik* der dem Modell zugrunde liegenden *Modellierungssprache* definiert. Die Syntax einer Modellierungssprache setzt sich aus der *konkreten Syntax* und der *abstrakten Syntax* zusammen. Die konkrete Syntax legt die Notation der Sprache fest. Die abstrakte Syntax beschreibt die Struktur der Sprache. Die Semantik hingegen definiert die Bedeutung der Sprache [HR00].

Sowohl die konkrete Syntax als auch die Semantik der meisten Modellierungssprachen werden häufig nur informell durch verbale Beschreibungen und Beispiele angegeben - obwohl geeignete Mittel zur Verfügung stehen und beide formal definiert werden könnten [HR00]. Im Gegensatz dazu wird die

abstrakte Syntax einer Modellierungssprache in den meisten Fällen formal angegeben. Der Grund hierfür ist, dass die Formalisierung der abstrakten Syntax einer Sprache eine notwendige Voraussetzung für eine automatisierte Verarbeitung dieser Sprache durch Softwarewerkzeuge darstellt.

Die abstrakte Syntax einer textuellen Sprache wird im Regelfall durch kontextfreie Grammatiken definiert. Zur Formalisierung einer visuellen Sprache, also einer Sprache mit einer graphischen konkreten Syntax, haben sich in der modellbasierten Softwareentwicklung hingegen Modelle durchgesetzt. Ein Modell, das die Elemente und die Struktur einer Sprache definiert, wird als *Metamodell* (von „meta“, griech. „über“) bezeichnet. Ein Metamodell definiert sowohl die Elemente der Sprache als auch ihre strukturellen Beziehungen zueinander.[1] Die Instanzen eines Metamodells stellen syntaktisch gültige Modelle der Sprache dar.

Beispiel

In dieser Arbeit verwenden wir als durchgängiges Beispiel die in Abschnitt 2.1 beschriebene Modellsynchronisation zwischen einem Block- und einem Klassendiagramm. Dort haben wir bereits beide Diagrammarten in ihrer graphischen Notation – also der konkreten Syntax – kennen gelernt. Um die Korrespondenzbeziehungen zwischen beiden Modellen mithilfe von Korrespondenzregeln zu spezifizieren, müssen die Metamodelle der beiden Diagrammarten vorliegen. An dieser Stelle betrachten wir daher die dazugehörigen Metamodelle, mit denen die abstrakte Syntax der beiden Diagramme definiert ist, sowie die Darstellung der abstrakten Syntax am Beispiel eines Blockdiagramms. Auf der Grundlage der beiden Metamodelle und der Darstellung der Modelle in abstrakten Syntax werden wir in Abschnitt 3.2 die Spezifikation von Korrespondenzregeln erläutern.

Metamodell für Klassendiagramme In Abbildung 3.1 ist ein Ausschnitt des in dieser Arbeit verwendeten Metamodells für UML-Klassendiagramme dargestellt. Zur Darstellung von Metamodellen verwenden wir eine eingeschränkte Form von UML-Klassendiagrammen, die auch in der Meta Object Facility (MOF) enthalten ist – einem Standard der Object Management Group (OMG) zur Metamodellierung.

[1] Die Angabe eines Metamodells alleine reicht in der Regel aber nicht aus. Häufig müssen zusätzliche Einschränkungen definiert werden. In der UML wird dazu die Object Constraint Language (OCL) verwendet, auf die wir an dieser Stelle jedoch nicht weiter eingehen.

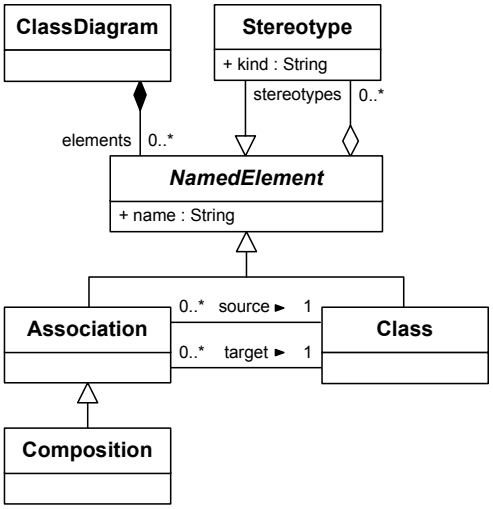


Abbildung 3.1: Metamodell für Klassendiagramme

Ein Klassendiagramm (`ClassDiagram`) besteht aus einer Menge von Modellelementen (`NamedElement`), die sich aus Klassen (`Class`), Assoziationen (`Association`) und Stereotypen (`Stereotype`) zusammensetzt. Ein Modellelement kann mit unterschiedlichen Stereotypen annotiert werden. Die Art eines Stereotyps wird dabei durch das Attribut `kind:String` in der Metaklasse `Stereotype` festgelegt. Eine Assoziation verbindet zwei Klassen miteinander. Eine spezielle Assoziation ist die Komposition (`Composition`).

Das hier gezeigte Metamodell stellt nur einen Ausschnitt aus dem originalen Metamodell für UML-Klassendiagramme dar. Beispielsweise sind keine Modellelemente für Attribute und Methoden in unserem Ausschnitt dargestellt. Ebenso wurde auf ein Metamodellelement für Generalisierungen verzichtet. Um die später zu spezifizierenden Korrespondenzregeln in unserem Beispiel möglichst übersichtlich zu halten, ist das hier gezeigte Metamodell zudem stark vereinfacht dargestellt. So verfügen zum Beispiel im originalen UML-Metamodell Assoziationen über explizite Assoziationsenden. Diese Assoziationsenden werden dort unter anderem verwendet, um zwischen einer gewöhnlichen Assoziation, einer Aggregation und einer Komposition zu unterscheiden. In unserem vereinfachten Metamodell verzichten wir auf die Assoziationsenden und verwenden lediglich eine Komposition, die durch eine von `Association` abgeleitete Metaklasse `Composition` repräsentiert wird. Anhang A enthält das vollständige Metamodell, das im Rahmen unserer Evaluation verwendet wurde.

In unserem Beispiel synchronisieren wir Klassendiagramme mit Blockdiagrammen. Daher benötigen wir noch ein Metamodell für Blockdiagramme. Aufgrund der Tatsache, dass wir in unserem Beispiel bei den Blockdiagrammen auf Signale und eine Unterscheidung der verschiedenen Verbindungsarten verzichten, haben wir auch dieses Metamodell vereinfacht dargestellt.

Metamodell für Blockdiagramme Das Metamodell für Blockdiagramme ist in der Abbildung 3.2 zu sehen. Sämtliche Modellelemente eines Blockdiagramms basieren auf der abstrakten Metaklasse `Element`, die ein Attribut `name:String` enthält. Somit können alle Modellelemente benannt werden.

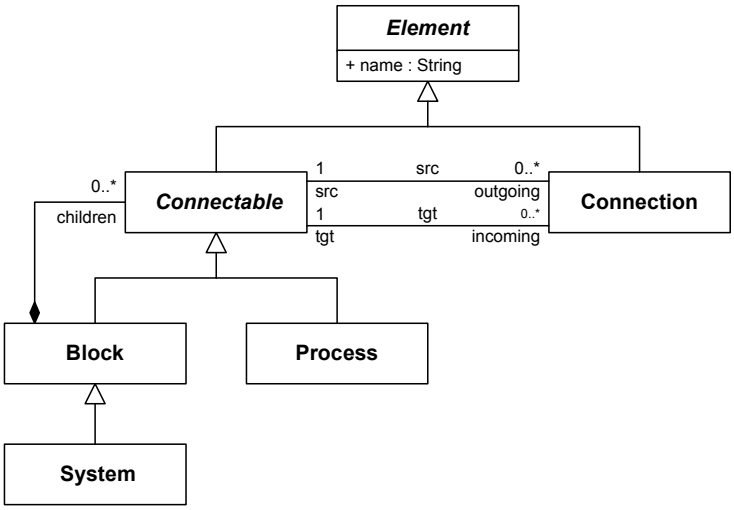


Abbildung 3.2: Metamodell für Blockdiagramme

Die hierarchische Struktur eines Blockdiagramms wird über die Kompositionsbeziehung `children` zwischen den Metaklassen `Block` und `Connectable` hergestellt. Der oberste Block in der Hierarchie eines Blockdiagramms stellt das modellierte System dar. Dieser spezielle Block wird durch die Metaklasse `System` repräsentiert. Modellelemente eines Blockdiagramms, die miteinander verbunden werden können, erben von der abstrakten Metaklasse `Connectable`. Hierzu zählen die Metaklassen `Block`, `Process` und die von `Block` abgeleitete Metaklasse `System`. Eine Verbindung zwischen diesen Elementen wird durch die Metaklasse `Connection` mit ihren Assoziationsbeziehungen (`src` und `tgt`) zur Metaklasse `Connectable` realisiert.

Blockdiagramm in abstrakter Syntax In Abbildung 2.3 (siehe Seite 23) haben wir ein SDL-Blockdiagramm in der graphischen konkreten Syntax kennen gelernt. Abbildung 3.3 hingegen zeigt das SDL-Blockdiagramm als UML-Objektdiagramm. Ein UML-Objektdiagramm besteht aus Objekten und Beziehungen zwischen diesen Objekten. Ein Objekt wird als Rechteck dargestellt und enthält im oberen Teil einen Bezeichner. Der Bezeichner setzt sich aus einem optionalen Objektnamen und dem darauf folgenden, durch einen Doppelpunkt getrennten, Klassennamen zusammen. Der untere Teil enthält einen optionalen Bereich, in dem Attribute dargestellt werden können. Eine Beziehung zwischen zwei Objekten wird durch eine Linie dargestellt, die optional durch einen Namen annotiert werden kann. In der UML wird eine Beziehung zwischen Objekten *Link* genannt.

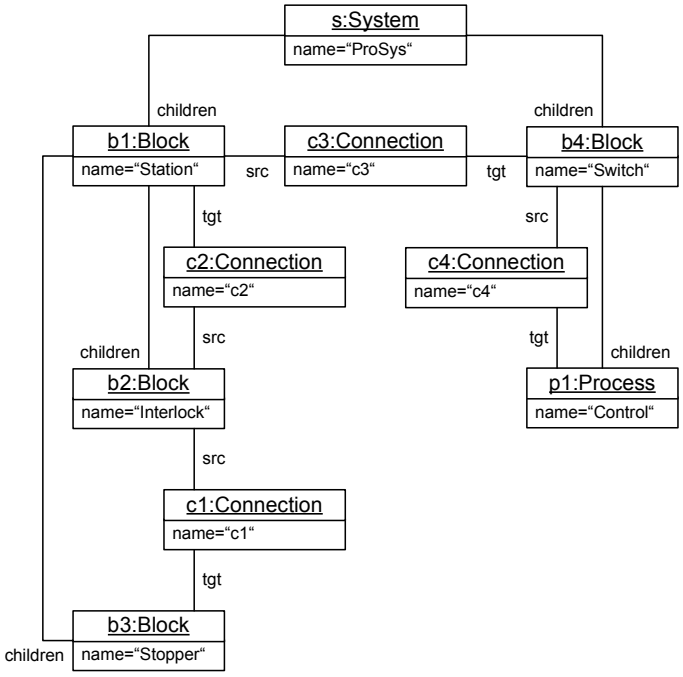


Abbildung 3.3: Blockdiagramm in abstrakter Syntax

Die in Abbildung 3.3 dargestellten Objekte und Links sind Instanzen der im Metamodell spezifizierten Klassen und Assoziationsbeziehungen. Beispielsweise stellen die Objekte `b1` und `b4` Instanzen der Klasse `Block` dar und der Link zwischen `b1` und `c3` eine Instanz der Assoziationsbeziehung `src`, die zwischen den Klassen `Connectable` und `Connection` in dem in Abbildung 3.2 spezifizierten Metamodell definiert wurde.

Diese Art der Darstellung eines Modells nennt man ein Modell in abstrakter Syntax. Diese Darstellung werden wir verwenden, wenn wir Korrespondenzbeziehungen zwischen Modellen mit Korrespondenzregeln spezifizieren.

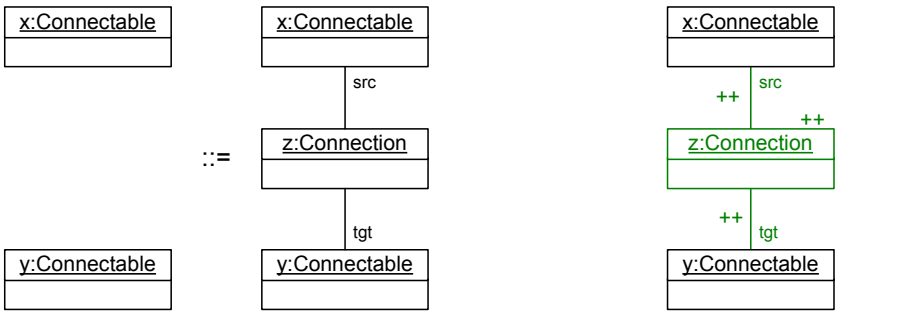
Aufgrund der Darstellung der abstrakten Syntax in Form eines UML-Objektdiagramms und der Tatsache, dass UML-Objektdiagramme als getypte und attributierte Graphen interpretiert werden können [Zün01], liegt es nahe, Graphgrammatiken und die Technik der Graphersetzung zur Spezifikation und Ausführung von Änderungsoperationen zu nutzen. Im nun folgenden Unterabschnitt gehen wir daher kurz auf Graphgrammatiken und die damit verbundene Graphersetzung ein. Für eine detaillierte Beschreibung verweisen wir auf [Roz97].

3.1.2 Graphgrammatiken

Analog zu klassischen Chomsky-Grammatiken, die – vereinfacht ausgedrückt – aus einer Menge von Produktionsregeln bestehen, besteht eine Graphgrammatik aus einer Menge von Graphgrammatikregeln. Beide Grammatiken definieren die durch sie erzeugbaren Wörter einer Sprache. Während jedoch bei einer klassischen Grammatik ein Wort durch eine Zeichenkette repräsentiert wird, besteht ein Wort einer Graphgrammatik aus einem Graphen. Das bedeutet, dass eine Graphgrammatik die durch sie erzeugbaren Graphen beschreibt.

Grundsätzlich besteht eine Graphgrammatikregel aus einer *linken* und einer *rechten* Regelseite, deren Elemente durch Knoten und Kanten repräsentiert werden, also Graphen sind. In Abbildung 3.4 sehen wir ein Beispiel für eine Graphgrammatikregel in zwei unterschiedlichen Notationen: Abbildung 3.4(a) zeigt die Graphgrammatikregel in der traditionellen Darstellung mit linker und rechter Regelseite; in der Abbildung 3.4(b) ist dieselbe Regel in einer kompakteren Darstellung zu sehen, die aus nur einem einzigen Graphen besteht. In beiden Fällen verwenden wir zur Darstellung der Graphen UML-Objektdiagramme.

In der traditionellen Darstellung in Abbildung 3.4(a) sind die linke Regelseite und die rechte Regelseite durch ein Zuweisungszeichen („`::=`“) voneinander getrennt. Das erste UML-Objektdiagramm repräsentiert die linke Regelseite, die zwei Objekte enthält: ein Objekt `x` und ein Objekt `y`. Beide Objekte sind vom Typ `Connectable` und tauchen auch auf der rechten Regelseite unserer Graphgrammatikregel auf. Durch die Verwendung derselben Namen `x` und `y` sowohl für Objekte der linken als auch der rechten Regelseite wird ausgedrückt, dass es sich hierbei um dieselben Objekte handelt.



(a) Traditionelle Notation (b) Kompakte Notation

Abbildung 3.4: Graphgrammatikregel in unterschiedlichen Notationen

Die rechte Regelseite enthält allerdings ein zusätzliches Objekt `z` vom Typ `Connection`, welches zwei Links (`src` und `tgt`) zu den Objekten `x` und `y` besitzt. Dieses Objekt repräsentiert eine Verbindung zwischen zwei Elementen eines Blockdiagramms.

In Abbildung 3.4(b) ist die kompaktere Regeldarstellung zu sehen. Dabei repräsentieren die Objekte und Links, die nicht mit `++` annotiert sind, alle Elemente, die sowohl auf der linken als auch auf der rechten Regelseite vorkommen. Die mit `++` annotierten Objekte und Links hingegen repräsentieren Elemente, die nur auf der rechten Regelseite vorkommen. Diese Objekte und Links sind zusätzlich grün dargestellt. In dieser Arbeit werden wir nur noch diese Kurzschreibweise für Graphgrammatikregeln verwenden.

Die Semantik einer Graphgrammatikregel entspricht der Semantik klassischer Grammatiken in formalen Sprachen. Die Anwendung einer Graphgrammatikregel ändert ein Objektdiagramm ähnlich wie eine Textgrammatikregel eine Zeichenfolge ändert. Im Gegensatz zu einer Produktionsregel einer klassischen Grammatik wird eine Graphgrammatikregel jedoch auf einen Graphen angewendet. Hierzu wird zuerst das Muster der linken Regelseite im Graphen gesucht und anschließend das im Graphen gefundene Muster durch das Muster der rechten Regelseite ersetzt. Tatsächlich werden bei der Anwendung einer Graphgrammatikregel allerdings die Elemente, die sowohl auf der linken als auch auf der rechten Regelseite vorkommen, nicht ersetzt, sondern beibehalten und nur neue Elemente dem Graphen hinzugefügt.

Bei der kompakten Darstellung einer Graphgrammatikregel zeigt sich ein besonderer Vorteil dieser Kurzschreibweise. Die Kennzeichnung mit `++` betont in der kompakten Darstellung einer Graphgrammatikregel die Bedeu-

tung dieser Elemente: Diese Elemente werden zum UML-Objektdiagramm hinzugefügt, sobald eine Zuordnung der Elemente der linke Regelseite im UML-Objektdiagramm erfolgreich durchgeführt werden konnte.

In Abbildung 3.5 ist ein Beispiel für eine Anwendung einer Graphgrammatikregel zu sehen. Dabei wenden wir die Graphgrammatikregel aus Abbildung 3.4 auf das UML-Objektdiagramm aus Abbildung 3.3 an. Um die Graphgrammatikregel an einer bestimmten Stelle im UML-Objektdiagramm anwenden zu können, müssen zuerst die Objekte und Links der linken Regelseite auf Objekte und Links im UML-Objektdiagramm abgebildet werden. Dieser Vorgang wird auch *Binden* genannt.

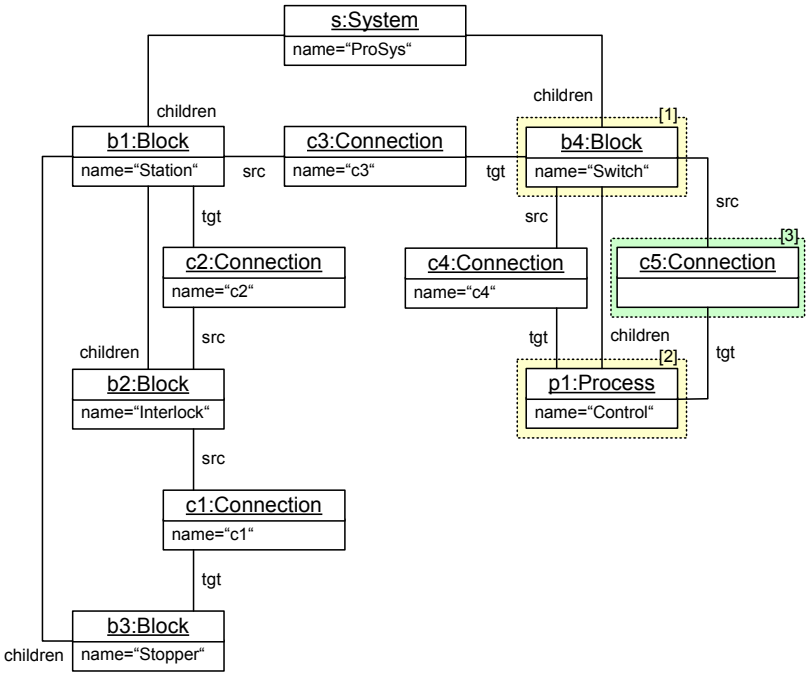


Abbildung 3.5: Blockdiagramm nach der Regelanwendung

In unserem Beispiel binden wir den Knoten `x` der Graphgrammatikregel an das Objekt `b4` (siehe gelb schattierter Bereich [1]) und den Knoten `y` an das Objekt `p1` (siehe gelb schattierter Bereich [2]). Diese Zuordnung ist möglich, da sowohl `Block` als auch `Process` von der Metaklasse `Connectable` abgeleitet sind. Durch diese Vererbung kann ein Objekt vom Typ `Connectable` sowohl an Objekte des Typs `Block` als auch an Objekte des Typs `Process` gebunden werden.

Wurde eine gültige Zuordnung gefunden, so kann die Graphgrammatikregel ausgeführt werden. Dies bedeutet, dass nun alle Objekte und Links erzeugt werden, die zwar in der rechten Regelseite, aber nicht in der linken Regelseite vorkommen. Dabei wird die zuvor gefundene Zuordnung, das heißt, die bereits gebundenen Objekte und Links, beibehalten und nur Objekte und Links erzeugt, die sich nur auf der rechten Regelseite befinden. In unserem Beispiel wird daher ein Kanal `c5` mit den dazugehörigen Links `src` und `tgt` zwischen den Objekten `b4` und `p1` erzeugt, wie in Abbildung 3.5 zu sehen ist (siehe grün schattierter Bereich [3]). In unserem Beispiel haben wir nur eine mögliche Regelanwendung gezeigt. Natürlich können wir die Graphgrammatikregel immer wieder ausführen - auch wenn das nicht immer einen Sinn ergibt.

Zusätzlich zu den hier dargestellten Konzepten verfügen Graphgrammatikregeln über weitere Konzepte, wie zum Beispiel negative Anwendungsbedingungen, Attributbedingungen und -zuweisungen sowie Löschoperationen. Diese Konzepte benötigen wir im Moment nicht. Wir werden uns daher mit ihnen später an geeigneter Stelle beschäftigen.

3.2 Tripel-Graph-Grammatiken

Tripel-Graph-Grammtiken (TGGs) sind bereits 1994 von Andy Schürr [Sch94] als Erweiterung von T. W. Pratts Pair-Grammatiken [Pra71] eingeführt und formal definiert worden. In diesem Abschnitt verzichten wir daher auf eine formale Darstellung und stellen nur die Idee sowie die zugrundeliegenden Prinzipien der Tripel-Graph-Grammatiken vor.

3.2.1 Syntax und Semantik

Eine Tripel-Graph-Grammatik besteht – genau wie eine Graphgrammatik – aus einer Menge von Graphgrammatikregeln. Die Graphgrammatikregeln einer Tripel-Graph-Grammatik werden TGG-Regeln genannt. Sie stellen die zur Modellsynchronisation benötigten Korrespondenzregeln dar und ermöglichen uns – wie noch später gezeigt wird – eine Modellsynchronisation durchzuführen. In diesem Abschnitt beschäftigen wir uns mit der Syntax und Semantik solcher Korrespondenzregeln.

Syntax

Die Spezifikation von Korrespondenzregeln zeigen wir an unserem bereits aus Abschnitt 2.1 bekannten Beispiel. Die zuvor nur informell beschriebene Zuordnung der Elemente (siehe Abbildung 2.4, Seite 24) wird in diesem Abschnitt mit TGG-Regeln formal definiert. Die Spezifikation einer TGG-Regel erfolgt im Gegensatz zur informellen Zuordnung jedoch nicht in der konkreten sondern in der abstrakten Syntax der Modelle.

Regel 1 (Zuordnung 2 und 3) In Abbildung 3.6 ist eine TGG-Regel dargestellt. Auf den ersten Blick entspricht die TGG-Regel einer gewöhnlichen Graphgrammatikregel. Sie besitzt eine linke sowie eine rechte Regelseite. Der Unterschied zu einer Graphgrammatikregel ist die Aufteilung der TGG-Regel in drei unterschiedliche Bereiche, die verschiedenen *Domänen* zugeordnet sind. Im linken Bereich der TGG-Regel befindet sich die Domäne der Blockdiagramme. Die Elemente der Klassendiagrammdomäne befinden sich im rechten Bereich der TGG-Regel. Der mittlere Bereich hingegen enthält Objekte, die eine explizite Korrespondenzbeziehung zwischen den Elementen eines Block- und eines Klassendiagramms definieren. Diese Objekte werden *Korrespondenzobjekte* genannt. Dementsprechend wird der mittlere Bereich als *Korrespondenzdomäne* bezeichnet. Jeder der drei Bereiche kann als eigenständige Graphgrammtikregel aufgefasst werden – daher auch der Name „Tripel-Graph-Grammatik“.[2]

Die in Abbildung 3.6 gezeigte TGG-Regel *Block2Class* definiert eine Korrespondenzbeziehung zwischen einem Block eines Blockdiagramms und den Elementen eines Klassendiagramms. Sie entspricht der informellen Zuordnung 3 aus Abbildung 2.4 (vergleiche Seite 24). Dabei stellen die Elemente der linken Regelseite den Kontext der Regel dar. Das bedeutet, dass die Regel nur dann ausgeführt wird, wenn diese Elemente gebunden werden können. Die mit ++ annotierten Elemente sind die zu erzeugenden Elemente. Das neue Korrespondenzobjekt zwischen den Elementen, das ebenfalls mit ++ gekennzeichnet ist, repräsentiert dabei explizit die Korrespondenzbeziehung zwischen den Elementen der einzelnen Modelle.

Für die in Abbildung 2.4 dargestellte Zuordnung 2 ist keine eigene TGG-Regel nötig, da die vorgestellte TGG-Regel *Block2Class* die Zuordnung eines in einem System enthaltenen Blocks bereits abdeckt. Dies liegt daran, dass

[2] Die vertikal eingezeichneten Linien gehören nicht zur Syntax einer TGG-Regel. Sie dienen lediglich zur besseren Darstellung der drei unterschiedlichen Domänen.

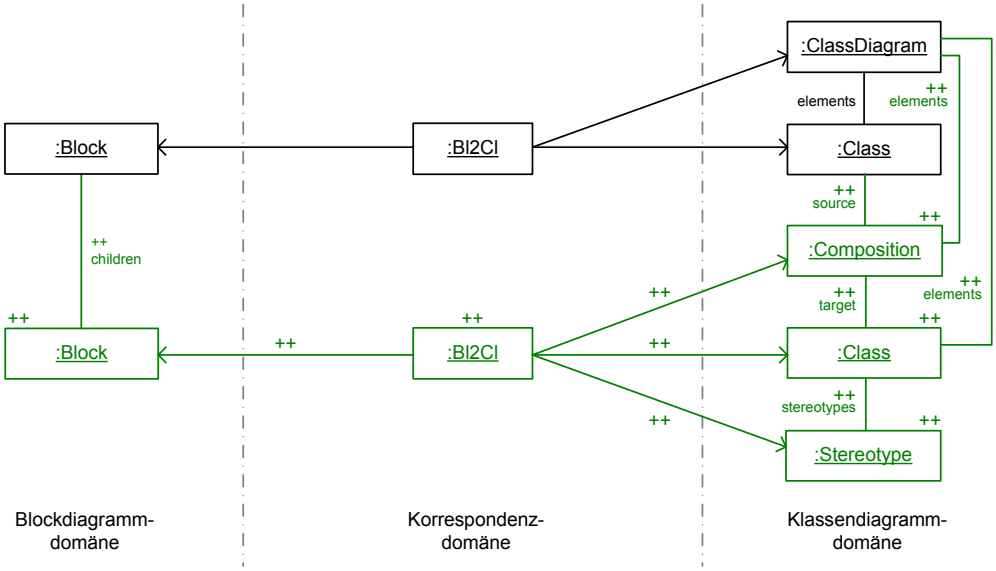


Abbildung 3.6: TGG-Regel *Block2Class*

im Metamodell für Blockdiagramme ein System von einem Block erbt und damit auch ein Block ist (vergleiche Abbildung 3.2, Seite 58).

Regel 2 (Zuordnung 4) Die in Abbildung 2.4 dargestellte Zuordnung 4 eines in einem Block enthaltenen Prozesses wird in der TGG-Regel *Process2Class* ausgedrückt. Diese TGG-Regel ist in Abbildung 3.7 zu sehen. Die TGG-Regel ist sehr ähnlich zu der zuvor beschriebenen TGG-Regel *Block2Class* aufgebaut. Der Unterschied ist hier, dass statt eines Blocks nun ein Prozess zu einer Klasse in Beziehung gesetzt wird.

Regel 3 (Zuordnung 5 und 6) Die TGG-Regel für die Zuordnung von Kanälen eines Blockdiagramms zu Assoziationen in einem Klassendiagramm (vgl. Abbildung 2.4, Zuordnungen 5 und 6) ist hingegen ein wenig anders. Sie ist in Abbildung 3.8 zu sehen. Die TGG-Regel drückt aus, dass ein Kanal zwischen Elementen des Blockdiagramms zu einer Assoziation zwischen zugehörigen Klassen des Klassendiagramms korrespondiert. Dabei ist es unerheblich, ob der Kanal zwischen einem System und einem Block, zwischen zwei Blöcken oder Prozessen, oder zwischen einem Block und einem Prozess besteht, da in dieser TGG-Regel sowohl das Quell- als auch das

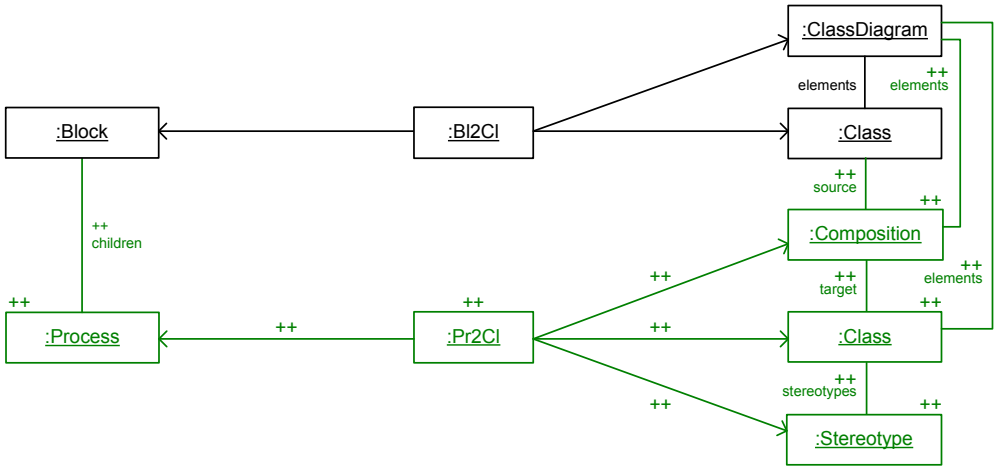


Abbildung 3.7: TGG-Regel *Process2Class*

Zielelement der Verbindung als `Connectable`-Objekt spezifiziert wurde. Da sowohl `System`, `Block` als auch `Process` von `Connectable` erben (vergleiche Abbildung 3.2, Seite 58), ist diese Regel auf Kanäle zwischen allen Elementen anwendbar, die direkt oder indirekt von `Connectable` abgeleitet sind. Statt also explizit verschiedene TGG-Regeln für die einzelnen Varianten anzugeben, reicht an dieser Stelle eine einzige TGG-Regel aus.[3]

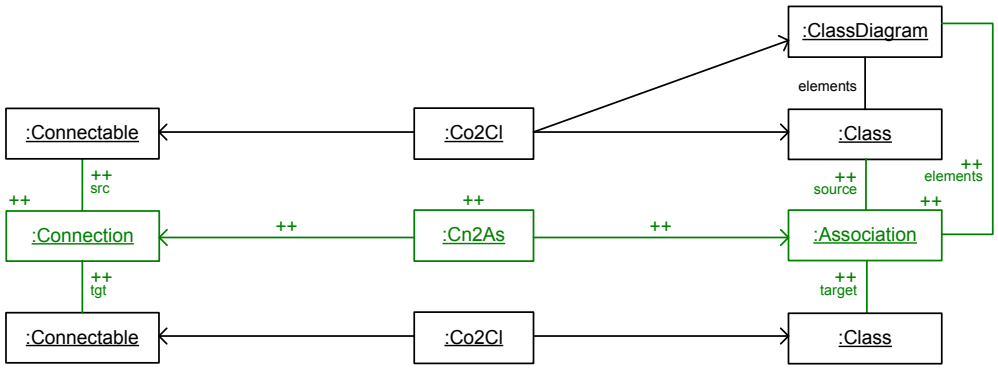


Abbildung 3.8: TGG-Regel *Connection2Association*

[3]Hierbei wurde allerdings vereinfachend angenommen, dass Kanäle über die Grenzen einer Hierarchiebene erlaubt sind.

Axiom (Zuordnung 1) Die initiale Zuordnung 1 eines Blockdiagramms zu einem Klassendiagramm erfolgt durch das in Abbildung 3.9 dargestellte Axiom. Das Axiom setzt ein System zu einem Klassendiagramm in Beziehung, welches eine mit einem entsprechenden Stereotyp gekennzeichnete Klasse enthält. Dies ist die Startsituation, auf die unsere TGG-Regeln angewendet werden können.

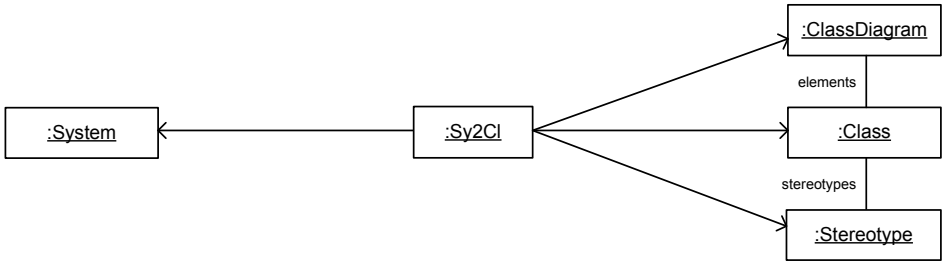


Abbildung 3.9: TGG-Axiom *System2Class*

Korrespondenzmetamodell

Die Grundlage zur Spezifikation der TGG-Regeln bilden die in den Abbildung 3.1 und 3.2 dargestellten Metamodelle. Zusätzlich muss jedoch noch ein Korrespondenzmetamodell spezifiziert werden, welches die Typen der Korrespondenzobjekte definiert. Abbildung 3.10 zeigt das in unserem Beispiel verwendete Korrespondenzmetamodell.

Das Metamodell definiert verschiedene Klassen und Assoziationen, mit denen die Elemente der zwei Modelle zueinander in Beziehung gesetzt werden können. Beispielsweise setzt eine Instanz der Klasse `Cn2As` ein Objekt vom Typ `Connection` und `Association` zueinander in Beziehung. Hierzu besitzt `Cn2As` eine Assoziation zur Klasse `Connection` und eine Assoziation zur Klasse `Association`. Für eine gültige Zuordnung müssen beide Assoziationsbeziehungen gesetzt sein. Daher ist die Kardinalität in beiden Fällen jeweils mit `1` angegeben.

Die Klasse `Co2Cl` stellt eine Beziehung zwischen einer Instanz der Klasse `Connectable` und Instanzen der Klassen `Stereotype`, `Class` und `Composition` her. Aufgrund der Tatsache, dass `Process` und `Block` direkt oder – wie im Fall von `System` – indirekt von `Connectable` abgeleitet sind, erben die Klassen `Pr2Cl` und `Bl2Cl` von der Klasse `Co2Cl` und `Sy2Cl` von der Klasse `Bl2Cl`.

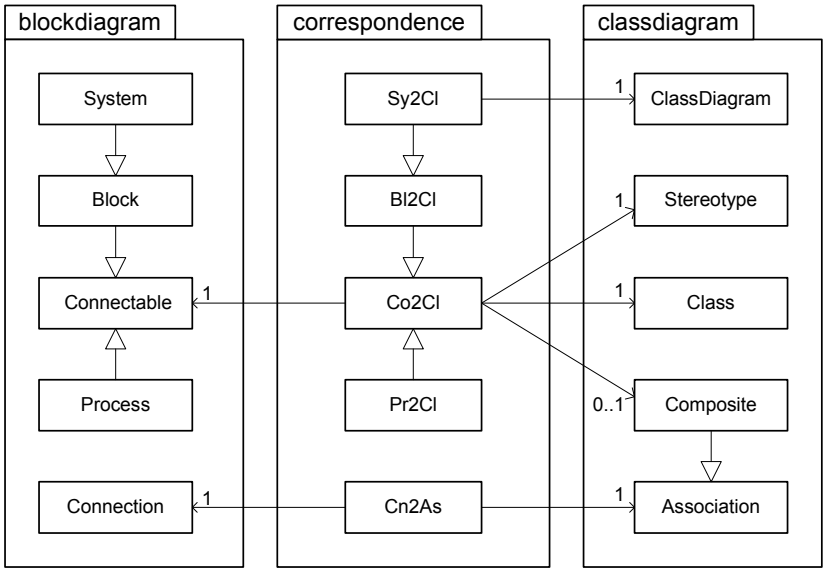


Abbildung 3.10: Metamodell für die Korrespondenzobjekte

Mit Hilfe von Vererbungsbeziehungen können TGG-Regeln häufig allgemeiner formuliert werden (vgl. Abbildung 3.8). Außerdem können zu den Vererbungsbeziehungen zusätzliche Einschränkungen definiert werden, um auf diese Art und Weise eine allgemein gültige Wiederverwendung von TGG-Regeln zu ermöglichen [KKS07]. In der hier vorliegenden Arbeit werden solche Einschränkungen jedoch nicht weiter berücksichtigt. In dieser Arbeit wird die Vererbung zwischen den Klassen im Korrespondenzmetamodell hauptsächlich dazu genutzt, um Assoziationen wiederzuverwenden. Die beiden Klassen `Pr2Cl` und `Bl2Cl` nutzen beispielsweise die durch die Klasse `Co2Cl` vorgegebenen Assoziationen. Sie benötigen keine weiteren Assoziationen, um die Beziehung zwischen den beteiligten Modellelementen herzustellen. Die Klasse `Sy2Cl` hingegen wird verwendet, um eine Beziehung zwischen Instanzen vom Typ `System` und dazu korrespondierenden Instanzen vom Typ `Class` und `ClassDiagram` herzustellen. Hierzu wird eine zusätzliche Assoziation zur Klasse `ClassDiagram` benötigt. Allerdings benötigt `Sy2Cl` keine Assoziation zur Klasse `Composition`, was die Kardinalität `0..1` der Assoziation zwischen `Co2Cl` und `Composite` erklärt.

Semantik

Die Semantik einer TGG-Regel stimmt mit der Semantik einer Graphgrammatikregel überein. Der Unterschied besteht lediglich darin, dass die TGG-Regel – zusammen mit weiteren TGG-Regeln – beschreibt, wie ein Blockdiagramm sowie ein dazu korrespondierendes Klassendiagramm gleichzeitig, also simultan und konsistent zueinander, erzeugt werden können. Die zusätzlich eingeführten Korrespondenzobjekte definieren gültige Korrespondenzbeziehungen zwischen den Elementen der unterschiedlichen Modelltypen, in unserem Beispiel also zwischen einem Block- und einem Klassendiagramm. Im Folgenden stellen wir die simultane Erzeugung beider Modelle an einem Beispiel vor.

Beispiel für die Regelanwendung Die Anwendung der TGG-Regeln ist in den Abbildungen 3.11–3.13 verdeutlicht. Wir beginnen mit der TGG-Regel *Block2Class* aus Abbildung 3.6 und wenden diese TGG-Regel auf das Axiom *System2Class* aus Abbildung 3.9 an. Die Abbildung 3.11 zeigt das Ergebnis dieser Regelanwendung. Die TGG-Regel *Block2Class* konnte auf die durch das Axiom angegebene Startsituation angewendet werden, da das Objekt `System` von `Block` erbt und damit auch vom Typ `Block` ist. Aufgrund der Vererbungsbeziehungen im Korrespondenzmetamodell können auch die Korrespondenzobjekte gebunden werden, so dass die TGG-Regel ausgeführt werden kann. Durch die Ausführung entsteht ein neuer Block mit einer dazu korrespondierenden Klasse im Klassendiagramm, die über eine Komposition und einen Stereotyp verfügt.

Die TGG-Regel *Block2Class* wird noch zweimal ausgeführt, nun jedoch auf dem neu erzeugten Block und der zugehörigen Klasse. Das Ergebnis der zweimaligen Regelanwendung ist in Abbildung 3.12 zu sehen. Hierbei wurden zwei neue Blöcke erzeugt, die mit dem in Abbildung 3.11 erstellten Block verbunden sind. Dementsprechend wurden im Klassendiagramm zwei dazu korrespondierende Klassen mit Stereotypen und Kompositionen erzeugt.

Zuletzt wenden wir die TGG-Regel *Connection2Association* an. Diese Regel erzeugt einen Kanal zwischen zwei Blöcken sowie eine dazu korrespondierende Assoziation im Klassendiagramm. Das Ergebnis dieser Regelanwendung ist in Abbildung 3.13 dargestellt. Das durch die Anwendung der TGG-Regeln entstandene UML-Objektdiagramm repräsentiert einen Ausschnitt aus dem Block- und Klassendiagramm der Abbildung 2.3 sowie einem UML-Objektdiagramm, welches die Korrespondenzen zwischen den Diagrammen darstellt. Die erzeugten Diagramme entsprechen dabei der Struktur, die

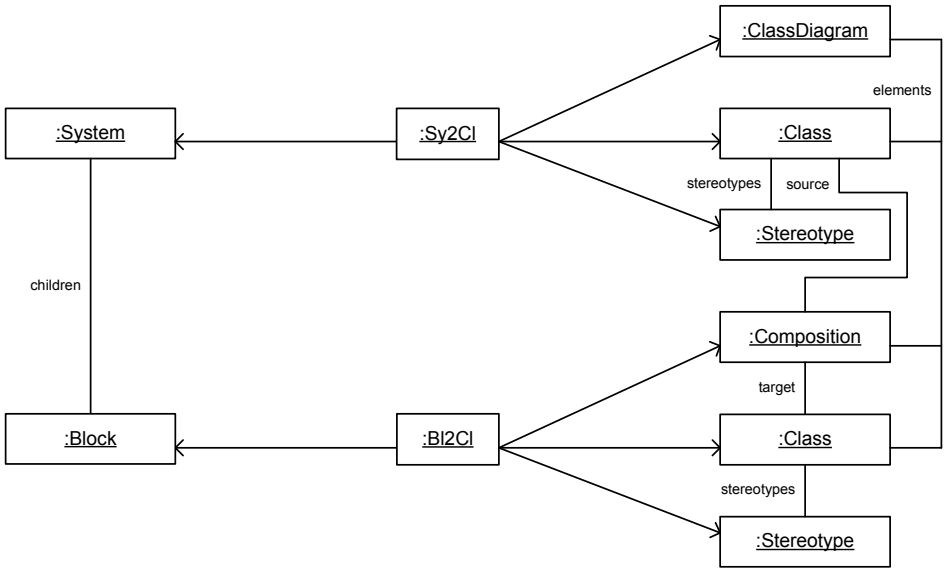


Abbildung 3.11: Anwendung der Regel *Block2Class* auf das Axiom

durch die Blöcke `ProSys`, `Station`, `Interlock` und `Stopper` vorgegeben wird.

3.2.2 Erweiterungen

Im vorherigen Abschnitt wurde das zugrunde liegende Prinzip der TGGs erläutert. Auf dieser Grundlage können beide Modelle simultan mit den Regeln einer TGG aufgebaut und die Elemente der Modelle zueinander in Beziehung gesetzt werden. Für einen Einsatz in der Praxis fehlen jedoch noch einige wichtige Konzepte. So müssen häufig Attributwerte von einzelnen Objekten abgefragt und gesetzt werden können. Ebenso muss es möglich sein, zusätzliche Bedingungen, wie zum Beispiel die Abwesenheit bestimmter Objekte[4], zu überprüfen. Die hierzu notwendigen Erweiterungen werden in diesem Abschnitt vorgestellt.

Attributbedingungen

In UML-Objektdiagrammen verfügen Objekte über Attribute, die mit einem konkreten Wert belegt sein können. Wenn wir die Beziehungen zwischen den

[4]Solche Bedingungen werden *Negative Anwendungsbedingungen* genannt.

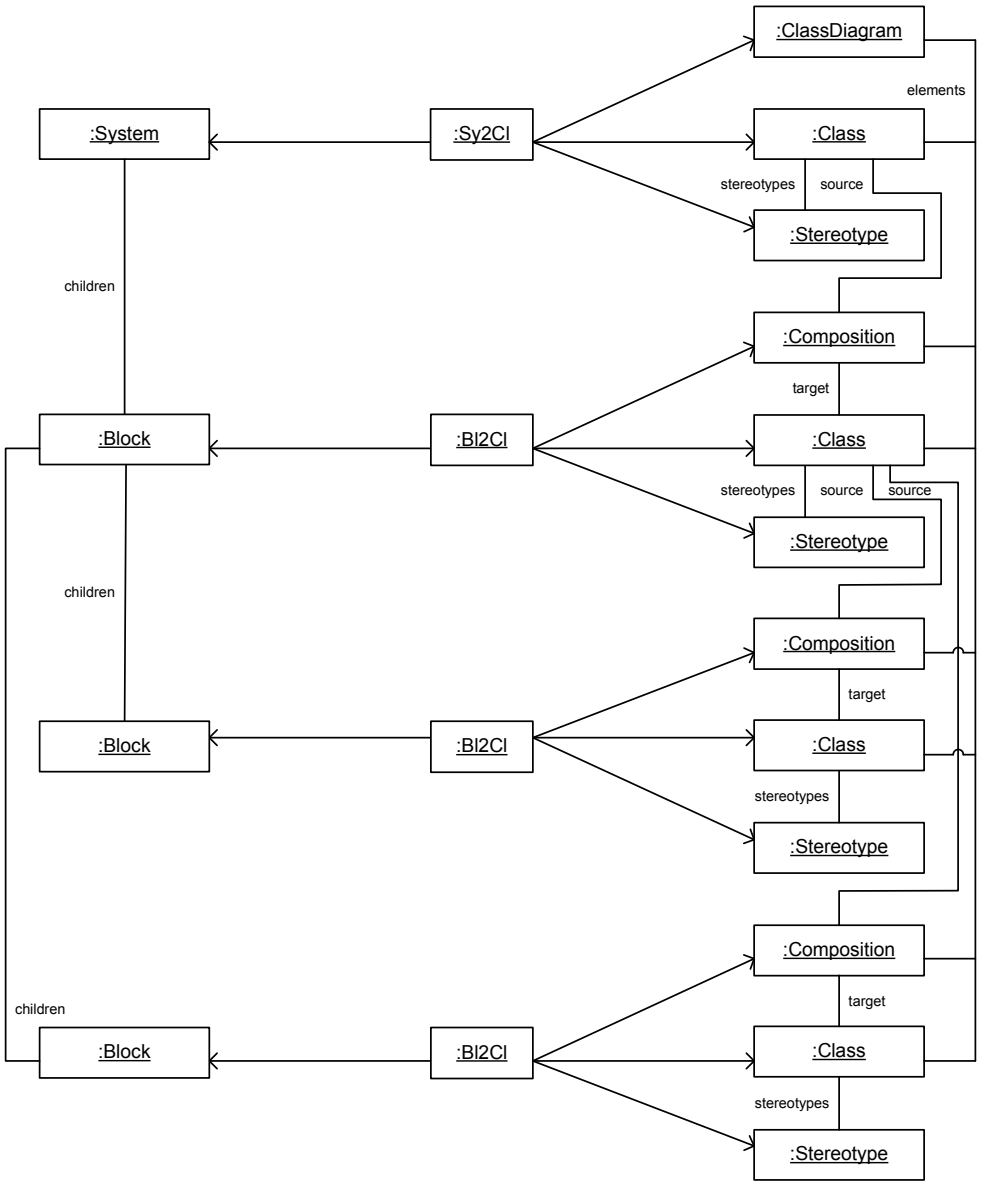


Abbildung 3.12: Zweimalige Anwendung der Regel *Block2Class*

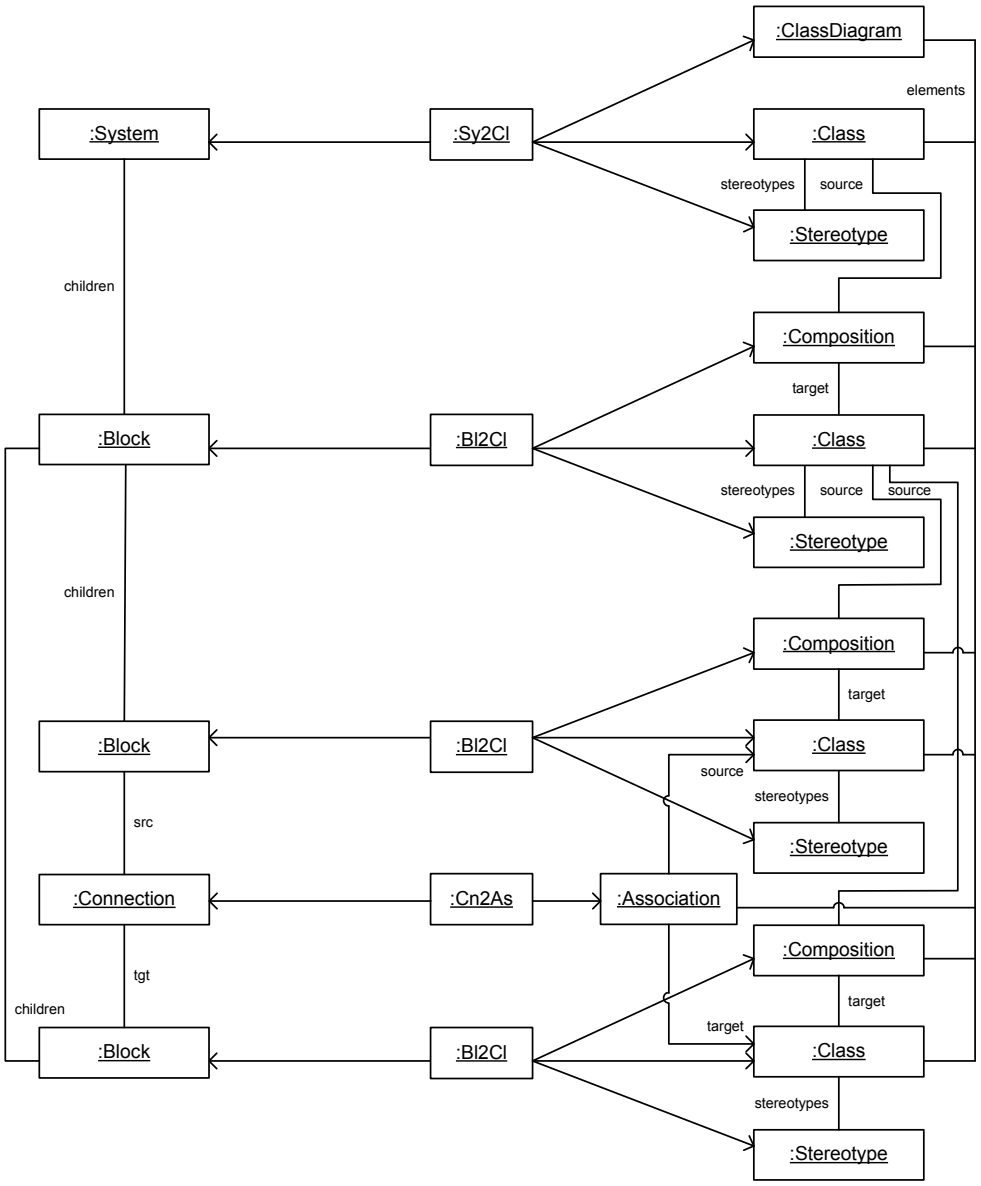


Abbildung 3.13: Anwendung der *Connection2Association* Regel

Elementen der unterschiedlichen Modelle beschreiben, möchten wir häufig Bedingungen an diese Objekte stellen, die über deren Attributwerte ausgedrückt werden. In unserem Beispiel sollen ein Block und eine Klasse nur dann zueinander in Beziehung gesetzt werden, wenn sie einen identischen Namen haben. Daher müssen wir in der Lage sein, die Attributwerte dieser Objekte abzufragen und – insbesondere im Rahmen der noch später vorzustellenden Modellsynchronisation – auch zu verändern.

Alte Notation Eine Möglichkeit ist, eine Attributbedingung direkt in dem davon betroffenen Objekt anzugeben. Falls zur Formulierung der Bedingung die Attributwerte unterschiedlicher Objekte verwendet werden müssen, kann die Bedingung ausserhalb eines konkreten Objektes direkt in der TGG-Regel spezifiziert werden. In Abbildung 3.14 ist die um Attributbedingungen erweiterte TGG-Regel aus Abbildung 3.6 dargestellt. Um innerhalb der Bedingungen die Attribute bestimmter Objekte referenzieren zu können, erhalten die betroffenen Objekte einen eindeutigen Bezeichner, in diesem Fall `bl` und `cl`. Im Gegensatz dazu muss im Fall des Stereotyps kein Objektname vergeben werden, da die Zuweisung direkt in dem betroffenen Objekt stattgefunden hat. Diese Art der Spezifikation von Attributbedingungen ist hauptsächlich durch existierende Graphtransformationswerkzeuge, wie zum Beispiel Progres [SWZ99] oder Fujaba [Fuj], motiviert.

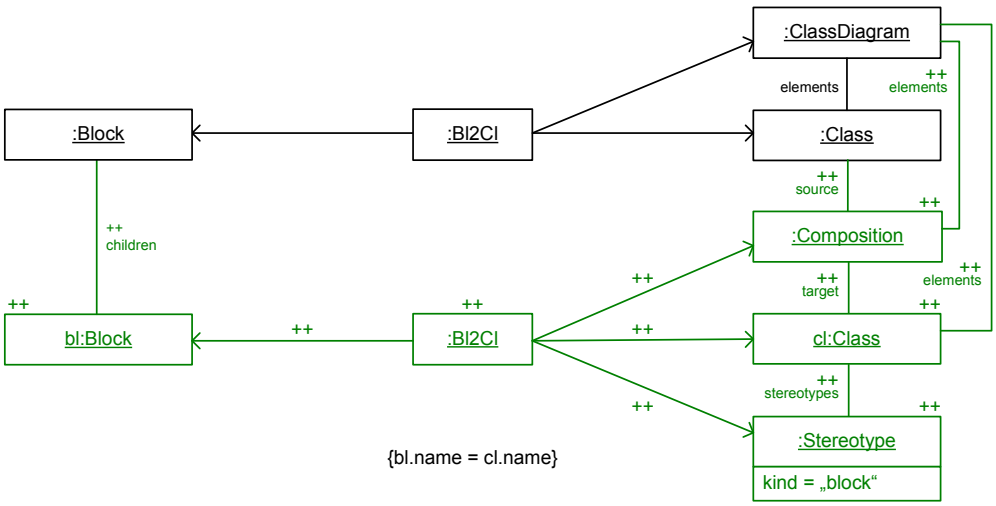


Abbildung 3.14: Alte Notation für Attributwerte

Die Idee hierbei ist, dass Attributzuweisungen, wie sie beispielsweise bei der Erzeugung neuer Objekte benötigt werden, prinzipiell aus den spezifizierten Bedingungen abgeleitet werden können. Beispielsweise kann aus der Bedingung `kind.name = „block“` die Zuweisung `kind.name := „block“` abgeleitet werden. Aus der Bedingung `bl.name = cl.name` kann für das Objekt `bl` die Attributzuweisung `name := cl.name` und für das Objekt `cl` die Attributzuweisung `name := bl.name` abgeleitet werden.

Für komplexere Bedingungen kann der Aufwand zur automatischen Berechung solcher Attributzuweisungen allerdings sehr hoch werden. Bereits bei einer Bedingung wie zum Beispiel `cl.name = bl.name.concat('Block')` ist nicht sofort ersichtlich, wie daraus automatisch eine Zuweisung an `bl.name` abgeleitet werden kann.[5] Daher müssen in den meisten TGG-Implementierungen sowohl Attributbedingungen als auch Attributzuweisungen direkt in den TGG-Regeln spezifiziert werden.

Dieser Ansatz funktioniert, insofern man die TGGs zur Modelltransformation, Modellintegration oder Modellsynchronisation einsetzt und darauf achtet, dass die spezifizierten Attributzuweisungen nicht im Widerspruch zu den spezifizierten Attributbedingungen stehen. Die eigentliche Semantik der TGGs, das heißt, die simultane Erzeugung beider Modelle, wird damit allerdings nicht unterstützt. Dies liegt daran, dass solche Bedingungen zwar Aussagen über die Beziehung der Attribute zueinander machen, aber keine Aussagen darüber, mit welchen Werten diese Attribute belegt werden sollen, falls alle Objekte neu erzeugt werden. Das ist auch nicht weiter verwunderlich, da ein solches Szenario in der Praxis bisher keine Anwendung gefunden hat. Allerdings sollte dies zumindest konzeptionell durch eine TGG unterstützt werden.[6] Aus diesem Grund wird hier ein neues Konzept für Attribute eingeführt. Es beseitigt diesen Nachteil und ist dennoch kompatibel zu den bereits existierenden TGG-Implementierungen.

Neue Notation Die Kernidee besteht aus der Einführung von Attributen in den Korrespondenzobjekten. Die Einführung eines Korrespondenzattributes geschieht allerdings nur dann, wenn Attributwerte zwischen Objekten in Beziehung gesetzt werden. Bei der Erzeugung beider Modelle mit einer TGG können diese Attribute mit konkreten Werten belegt und daraus

[5] Die Zuweisung hierfür könnte beispielsweise `bl.name := cl.name.substring(0, cl.name.lastIndexOf('Block'))` lauten. Die Operation `lastIndexOf(s:String)` existiert in OCL allerdings nicht und müsste daher zunächst definiert werden. Eine automatische Ableitung solcher Zuweisungen wird daher derzeit nicht unterstützt.

[6] vgl. auch die Arbeit von Alexander Königs [Kön08].

die Attributwerte für die Modellelemente abgeleitet werden. Hierzu wird die Ableitungsvorschrift in Form einer Bedingung für jede Domäne einzeln angegeben. Die neue Notation ist in Abbildung 3.15 angegeben. Die abgerundeten Rechtecke repräsentieren die Bedingungen. Jede Bedingung kann auch als Zuweisung an das Objektattribut interpretiert werden, deren Wert entweder direkt angegeben ist – wie im Fall der Bedingung für den Stereotyp – oder aus dem Wert des Korrespondenzattributs berechnet wird.

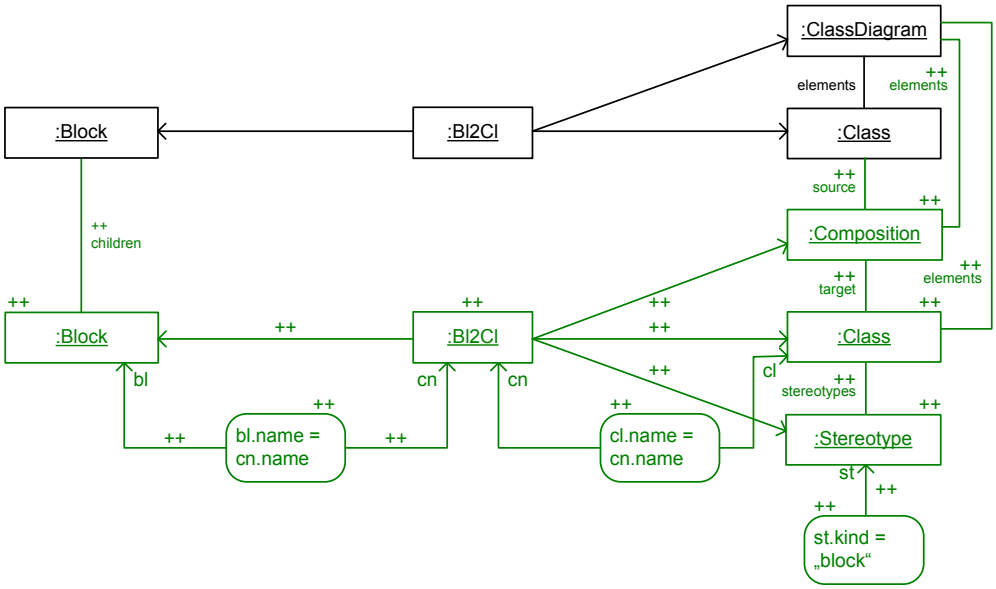


Abbildung 3.15: Neue Notation für Attributbedingungen

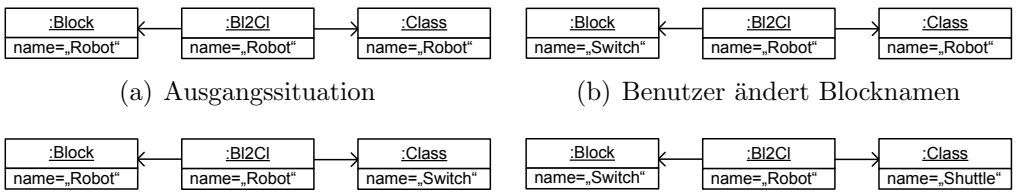
Für die Bedingungen existieren grundsätzlich keine Einschränkungen. Für eine inkrementelle Modellsynchronisation müssen allerdings die in den Bedingungen referenzierten Objekte lokal erreichbar sein, d. h., Pfadausdrücke zu Objekten, die nicht explizit in der TGG-Regel enthalten sind, dürfen in diesem Fall nicht verwendet werden. Darüber hinaus ist es – wie schon bei der alten Notation – vorteilhaft, wenn aus einer Bedingungen die Zuweisung an das Attribut des Korrespondenzobjekts automatisch ableitbar ist. Für die Fälle, in denen dies nicht möglich ist, sollte eine technische Realisierung die Möglichkeit bereit stellen, mit der solche Zuweisungen manuell vom Benutzer angegeben werden können.

Die Einführung von Attributen in Korrespondenzobjekten haben wir schon in früheren Arbeiten dargestellt [KW07]. Ähnliche Überlegungen findet man auch in der Arbeit von Alexander Königs [Kön08], wo zusätzliche Metain-

formationen, wie zum Beispiel die Benutzerkennung oder der Zeitpunkt der Erstellung eines Korrespondenzobjekts, in attributierten Korrespondenzobjekten gespeichert werden. Im Unterschied zu unserem Vorschlag werden dort die Bedingungen direkt in den Objekten spezifiziert. Darüber hinaus können die Werte für die Attribute über Parameter, die an die TGG-Regeln übergeben werden, vorgegeben werden. Eine solche Paramterisierung ist in unserem Ansatz ebenfalls denkbar. Allerdings war sie für die hier vorgestellte Modellsynchronisation nicht notwendig, so dass sie in dieser Arbeit nicht umgesetzt wurde.

Vorteile der neuen Notation Ein Vorteil der neuen Notation ist, dass damit die simultane Erzeugung beider Modelle realisiert werden kann. Eine Möglichkeit ist beispielsweise, im Korrespondenzmetamodell intiale Attributwerte für die Korrespondenzobjekte zu definieren. Bei der Erzeugung eines Korrespondenzobjekts werden dessen Attribute mit den definierten Intialwerten belegt, so dass sich die Attributwerte der übrigen Objekte aus den Attributwerten der Korrespondenzobjekte berechnen lassen könnten. Eine andere Möglichkeit ist, die Attribute der Korrespondenzobjekte über Parameter zu belegen, die einer TGG-Regel übergeben werden.

Ein weiterer Vorteil der Notation besteht darin, dass uns der Attributwert im Korrespondenzobjekt Aufschluss darüber gibt, in welchem Modell eine Attributwertänderung stattgefunden hat. Erlauben wir, dass beide Modelle geändert werden, ohne dass zwischendurch eine Modellsynchronisation stattfindet, so kann das Attribut außerdem zur Erkennung eines Konfliktes herangezogen werden.



(a) Ausgangssituation

(b) Benutzer ändert Blocknamen

(c) Benutzer ändert Klassennamen

(d) Benutzer ändert Block- und Klassennamen

Abbildung 3.16: Erkennung von Änderungen und Konflikten

Zur Erläuterung sind in der Abbildung 3.16 verschiedene Situationen dargestellt, bei denen ein Block zu einer Klasse in Beziehung steht. Die Bedingung unserer Regel fordert, dass ein Block und die dazu korrespondierende

Klasse den gleichen Namen haben. In der Abbildung 3.16(a) wird diese Bedingung erfüllt. Sie stellt die Ausgangssituation dar. Ändert der Benutzer den Namen des Blocks (vgl. Abbildung 3.16(b)), so stimmt der Attributwert des Korrespondenzobjekts mit dem Namen der Klasse überein. Hat der Benutzer hingegen den Namen der Klasse geändert (vgl. Abbildung 3.16(c)), so sind der Attributwert des Korrespondenzobjekts und der Name des Blocks identisch. Nach einem entsprechenden Vergleich kann nun der Blockname an die Klasse oder der Klassenname an den Block propagiert werden. Ohne ein solches Korrespondenzattribut ist diese Information nicht direkt verfügbar und muss über andere Wege, wie zum Beispiel durch eine Aufzeichnung von Änderungsereignissen, ermittelt werden. Erlauben wir hingegen, dass sowohl der Name des Blocks als auch der Name der Klasse geändert werden dürfen, ohne dass zwischendurch eine Modellsynchronisation stattfindet, so kann das Korrespondenzattribut zur Erkennung dieses Konfliktes herangezogen werden (vgl. Abbildung 3.16(d)).

Negative Anwendungsbedingungen

In dem vorangegangenen Abschnitt haben wir Attributbedingungen verwendet, um Korrespondenzbeziehungen zwischen Objekten einzuschränken. In diesem Abschnitt wird das Konzept von Bedingungen weiter verallgemeinert. Dazu motivieren wir zuerst eine weitere Art von Bedingungen – die sogenannten *Negativen Anwendungsbedingungen* – an einem Beispiel und diskutieren die damit verbundenen Probleme im Kontext der TGGs. Am Ende stellen wir unseren Ansatz vor, der sowohl mit Attributbedingungen als auch mit Negativen Anwendungsbedingungen einheitlich umgeht.

Motivation Zur Motivation betrachten wir die TGG-Regel aus Abbildung 3.7, in der ein Prozess zu einem übergeordneten Block hinzugefügt und in Beziehung zu einer Klasse im Klassendiagramm gesetzt wird. Nun ist es aber so, dass ein Prozess nur dann zu einem Block hinzugefügt werden darf, wenn dieser Block keine Blöcke seinerseits enthält (vergleiche Beschreibung auf Seite 22). Damit eine TGG nur korrekte Diagramme erzeugt, müssen solche Bedingungen in den TGG-Regeln berücksichtigt werden.

Abbildung 3.17 zeigt die erweiterte TGG-Regel, die genau diese Zusatzbedingung als OCL-Ausdruck enthält: in der Menge `children` darf kein Objekt vom Typ `Block` enthalten sein. Obwohl die Bedeutung dieser Bedingung offensichtlich und eindeutig zu sein scheint, können – je nach Zeitpuntk der Überprüfung der Bedingung – unterschiedliche Ergebnisse entstehen. Bei-

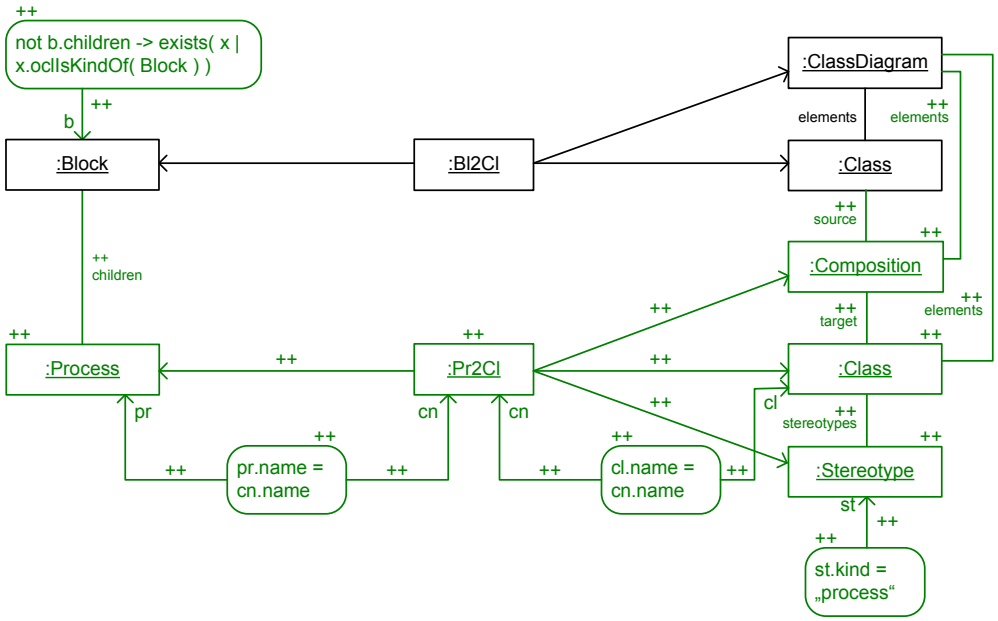


Abbildung 3.17: Erweiterte TGG-Regel *Process2Class*

spielsweise können wir, wie in Abbildung 3.12 gezeigt, zuerst drei Blöcke erzeugen. Wollen wir nun mit Hilfe der erweiterten TGG-Regel aus Abbildung 3.17 einen Prozess zu dem Block hinzufügen, der bereits die anderen zwei Blöcke enthält, so wird dies durch die Bedingung erfolgreich verhindert. Erzeugen wir hingegen zuerst nur einen Block (vgl. Abbildung 3.11) und wenden dann die TGG-Regel aus Abbildung 3.17 an, so ist die dort spezifizierte Bedingung erfüllt. Daher wird ein neuer Prozess erzeugt. Anschließend kann jedoch die TGG-Regel *Block2Class* angewendet werden, die in demselben Block einen neuen Block hinzufügt. Damit wird die Bedingung der erweiterten TGG-Regel *Process2Class* verletzt und letztendlich erhalten wir einen Block, der sowohl einen Prozess als auch einen Block enthält. Dieses Beispiel zeigt, dass der Überprüfungszeitpunkt der Bedingungen einen entscheidenden Einfluss auf das Ergebnis hat.

Das Ergebnis der Regelanwendung sollte unabhängig vom Zeitpunkt der Überprüfung von Bedingungen sein. Aus diesem Grund führen wir hier ein einheitliches und durchgängiges Konzept für alle Bedingungen ein und präzisieren damit die Semantik der Korrespondenzregeln. Die Idee hinter dem neuen Konzept ist einfach: Während der Anwendung von TGG-Regeln werden die Bedingungen in Form von Annotationen zu den erzeugten Modellen

hinzugefügt.[7] Nachdem das gesamte Modell konstruiert wurde, werden die hinzugefügten Bedingungen evaluiert. Die Korrespondenzbeziehungen zwischen zwei Modellen sind nur dann gültig, wenn alle Bedingungen auch nach der Konstruktionsphase erfüllt sind.

Durch diese Definition wird eine a-posteriori Semantik für Bedingungen festgelegt, sodass die Bedingungen einer Korrespondenzregel als Invarianten aufgefasst werden können. In der Praxis kann natürlich der Überprüfungszeitpunkt anders gewählt werden. Beispielsweise können Bedingungen bereits während der Anwendung von TGG-Regeln herangezogen werden, um aus der Menge aller TGG-Regeln die anzuwendende TGG-Regel zu bestimmen oder zwischen mehreren anwendbaren Regelalternativen zu wählen. Wichtig hierbei ist aber, dass die oben beschriebene Definition erfüllt wird, das heißt, dass die Bedingungen auch nach der Anwendung aller TGG-Regeln erfüllt sind. Wie dies sichergestellt wird bleibt jeder TGG-Implementierung überlassen.

Kurzschreibweise für Negative Anwendungsbedingungen Als graphische Notation negativer Anwendungsbedingungen wird in einigen Ansätzen ein durchgestrichenes Objekt verwendet. Aufgrund der Tatsache, dass es hierbei zu Mehrdeutigkeiten kommen kann (wenn z. B. nur das Objekt aber nicht der dazu inzidente Link durchgestrichen dargestellt wird), erlauben wir in unserer graphischen Notation lediglich das Durchstreichen von Links. In Abbildung 3.18 sind drei negative Anwendungsbedingungen dargestellt. In der linken Bildhälfte sind die Anwendungsbedingungen in ihrer graphischen Kurzschreibweise zu sehen. In der rechten Bildhälfte ist die Semantik der graphischen Kurzschreibweise in Form eines OCL-Ausdrucks definiert.

Die erste negative Anwendungsbedingung fordert, dass kein Link zwischen zwei ganz konkreten Objekten existieren darf. Dies wird durch den durchgestrichenen Link ausgedrückt. Zusätzlich ist der Link rot gefärbt. Dies wird auch durch die Bedingung auf der rechten Seite der Abbildung ausgedrückt. Die zweite negative Anwendungsbedingung unterscheidet sich zu der ersten Anwendungsbedingung dadurch, dass nun eines der Objekte durch ein rot gestricheltes Rechteck dargestellt wird. Dies bedeutet, dass es keinen Link zu irgendeinem Objekt dieses Typs geben darf. Die dritte Anwendungsbedingung schließlich fordert, dass zu gar keinem anderen Objekt ein solcher Link existieren darf. In der Notation wird dafür der Typ des Objektes weggelassen.

[7] Die Markierung einer Bedingung mit ++ betont dieses Konzept.

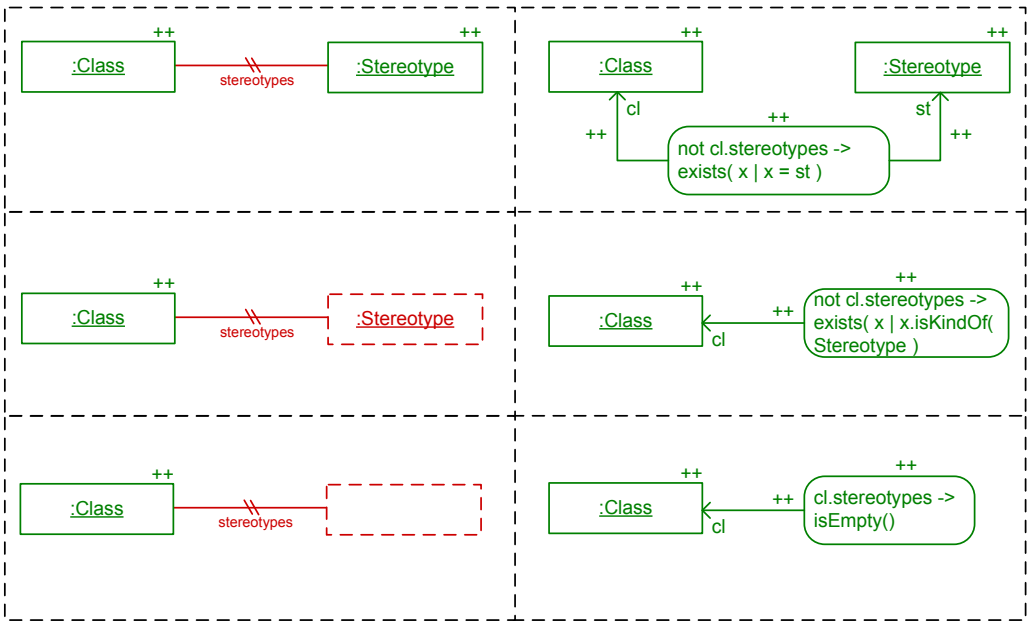


Abbildung 3.18: Negative Anwendungsbedingungen und ihre Übersetzung

Wiederverwendung von Objekten

Bisher verlangen TGGs, dass alle Elemente der zueinander in Beziehung gesetzten Modelle erzeugt werden. In einigen Fällen existieren allerdings Modellelemente, die während der TGG-Regelanwendung weder generiert noch verändert werden sollen, aber trotzdem in Beziehung zu den generierten Modellelementen stehen und von diesen referenziert werden. Um dieses Problem zu lösen, erweitern wir an dieser Stelle die TGGs um ein weiteres Konzept: die Wiederverwendung von Objekten.

Beispiel Bisher wurde in unserem Beispiel zu jeder neu erzeugten Klasse auch ein neuer Stereotyp erzeugt und mit dieser Klasse verbunden. Nun ist es aber so, dass von jeder Stereotypart immer nur ein Stereotypobjekt existieren soll. Ein solches Stereotypobjekt kann aber mehrfach referenziert werden – es kann also wiederverwendet werden. In Abbildung 3.19 ist ein erweitertes Metamodell für Klassendiagramme zu sehen, dass diesen Umstand berücksichtigt. Ein Stereotyp kann weiterhin von `NamedElement` über die Assoziation `stereotypes` referenziert werden. Zusätzlich existiert jedoch die Klasse `Project`, über die verschiedene Stereotypen verwaltet werden.

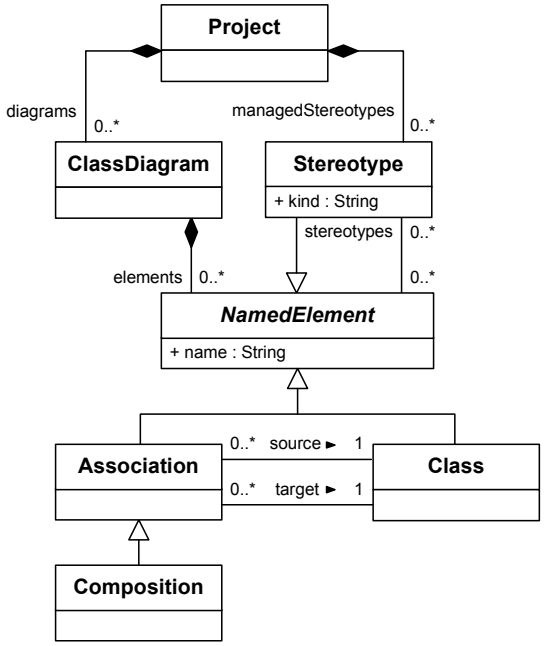


Abbildung 3.19: Erweitertes Metamodell für Klassendiagramme

Die TGG-Regeln, in denen ein Stereotypobjekt verwendet wird, müssen an dieses neue Metamodell angepasst werden. Diese Anpassungen schauen wir uns am Beispiel der TGG-Regel *Block2Class* genauer an. Die modifizierte TGG-Regel ist in Abbildung 3.20 dargestellt. Prinzipiell entspricht die neue TGG-Regel der alten TGG-Regel. Der einzige Unterschied besteht darin, dass nun zusätzlich – ausgehend von `ClassDiagram` – ein Objekt vom Typ `Project` referenziert wird und der neu erstellte Stereotyp mit diesem Objekt verbunden wird. Trotz dieser Veränderung wird weiterhin bei jeder Anwendung der Regel ein jeweils neuer Stereotyp «`block`» erzeugt – auch wenn bereits ein Stereotypobjekt dieser Art im Projekt existiert.

Lösungsvariante 1: Fallunterscheidung Eine naheliegende Idee zur Lösung unseres Problems besteht darin, statt einer TGG-Regel zwei TGG-Regeln zu spezifizieren, die mit Hilfe von Negativen Anwendungsbedingungen eine Fallunterscheidung vornehmen. In Abbildung 3.21 sind die beiden TGG-Regeln zu sehen. Im Vergleich zur TGG-Regel aus Abbildung 3.20 enthält die erste TGG-Regel sowohl den Stereotypen als auch den Link `managedStereotypes` sowohl auf der linken als auch auf der rechten Re-

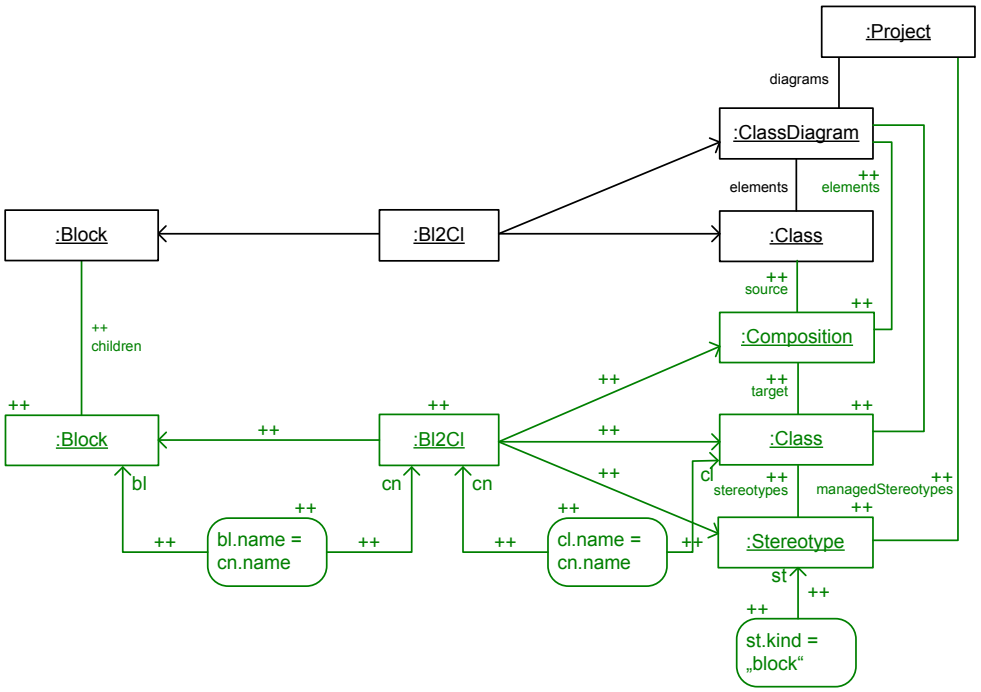


Abbildung 3.20: Erweiterte TGG-Regel *Block2Class*

gelseite, das heißt, die Markierung des Stereotyps und des Links mit ++ ist entfallen. Damit setzt diese TGG-Regel die Existenz eines entsprechenden Stereotypobjekts voraus. Es wird lediglich eine Referenz zwischen der neu erzeugten Klasse und dem Stereotyp erstellt. Die zweite Regel ist identisch zu der TGG-Regel aus Abbildung 3.20, enthält aber noch eine zusätzliche Bedingung, die überprüft, dass noch kein Stereotypobjekt «`block`» in dem Projekt enthalten ist. Damit wird die zweite TGG-Regel nur ausgeführt, wenn ein Block zum ersten Mal erzeugt wurde. Da bei der Erzeugung weiterer Blöcke dann ein Stereotyp «`block`» bereits vorhanden ist, greift die Bedingung dieser Regel nicht mehr. Stattdessen ist dann aber die erste TGG-Regel anwendbar.[8]

Mit der in dieser Arbeit eingeführten Semantik für Bedingungen ist diese Idee jedoch nicht realisierbar. Dies liegt daran, dass die Bedingung der zweiten TGG-Regel nach unserer Definition auch noch nach der Regelanwendung gelten muss. Die Bedingung dieser TGG-Regel wird aber durch die Erzeugung des Stereotyps in der TGG-Regel selbst immer falsifiziert, das heißt, die TGG-Regel steht zu der in ihr enthaltenen Bedingung im Widerspruch.

Ein Ausweg aus diesem Dilemma ist möglich, wenn man eine weitere Form von Bedingungen einführt. Die bisher eingeführten Bedingungen müssen aufgrund unserer Definition insbesondere auch nach der Regelanwendung gelten. Um das beschriebene Szenario zu ermöglichen, könnten aber Bedingungen eingeführt werden, die nur *vor der Regelanwendung* gelten müssen. Zur Unterscheidung könnten diese Bedingungen ohne die ++ Markierungen notiert werden, was die Semantik dieser Bedingungen betonen würde, da nun die Bedingungen nicht als Annotationen während der Regelanwendung hinzugefügt werden würden. Würde man diese weitere Form von Bedingungen einführen, würde dies aber die Semantik verkomplizieren. Daher sehen wir von dieser Möglichkeit ab und führen stattdessen das Konzept der *wiederverwendbaren Objekte* ein.

Lösungsvariante 2: Wiederverwendbare Objekte Wiederverwendbare Objekte werden graphisch als graue Objekte dargestellt. Zur besseren Unterscheidung sind sie zusätzlich mit `[]` markiert. In Abbildung 3.22 ist die TGG-Regel mit dem als wiederverwendbares Objekt markiertem Stereotyp

[8] Negative Anwendungsbedingungen werden häufig dazu eingesetzt, um die Anwendung von TGG-Regeln einzuschränken. In der Dissertation von Alexander Königs [Kön08] werden die damit verbundenen Probleme im Zusammenhang mit Modelltransformationen an einem Beispiel vorgestellt. Es wird gezeigt, dass bislang keine Lösung existiert, die mit negativen Anwendungsbedingungen zufriedenstellend umgehen kann.

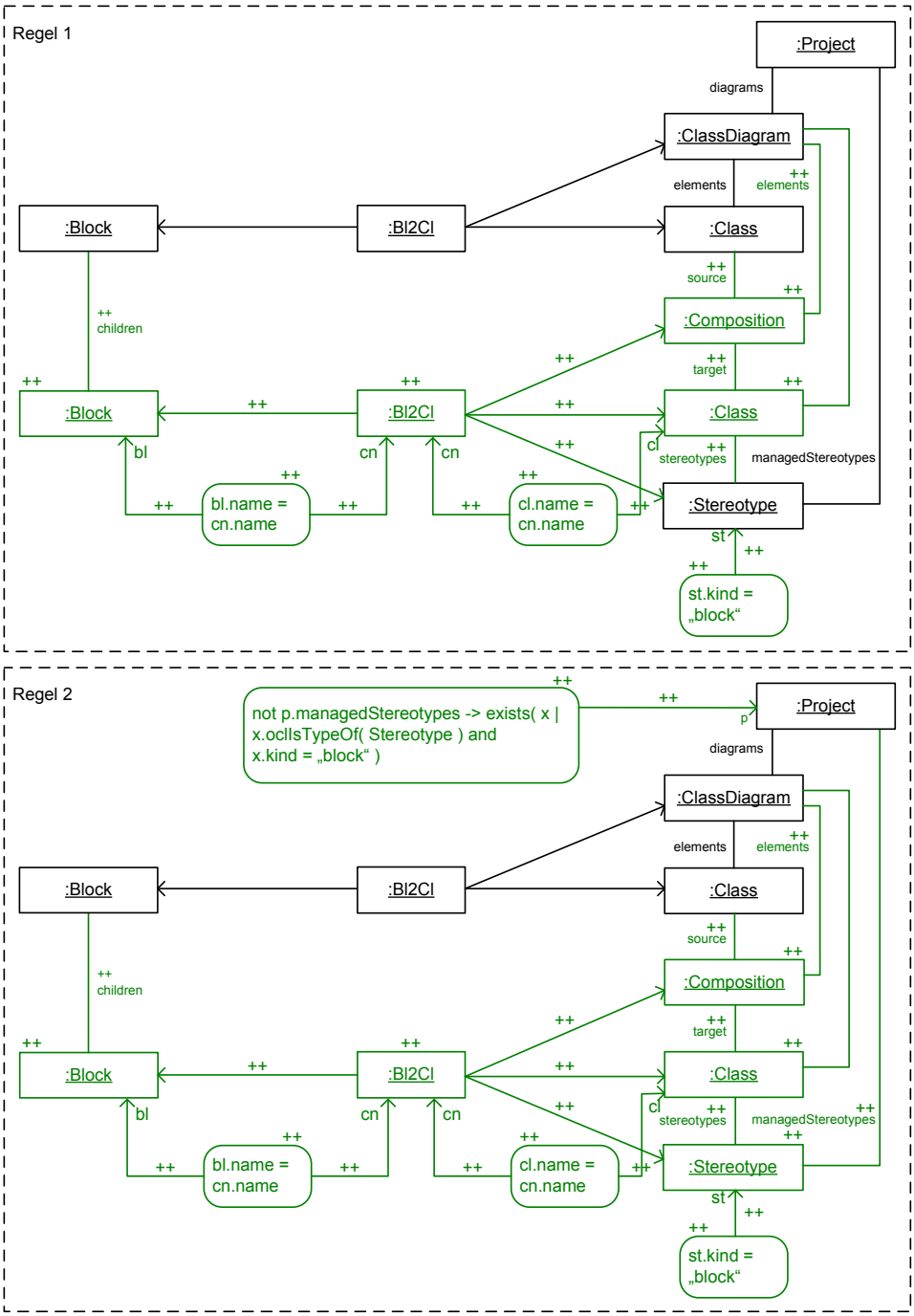


Abbildung 3.21: Fallunterscheidung mit zwei TGG-Regeln zur Wiederverwendung von Stereotypen

und den dazugehörigen Links dargestellt. Die Semantik der wiederverwendbaren Objekte ist so festgelegt, dass es ausdrücklich erlaubt ist, ein bereits vorhandenes Objekt wiederzuverwenden, sofern es den Eigenschaften (Typ und evtl. geforderte Attributwerte) des spezifizierten Objektes entspricht. Existiert ein solches Objekt nicht, so wird es neu erzeugt. Die graue Farbe unterstreicht die Semantik eines solchen Objekts, da das Objekt entweder als schwarzes Objekt (in diesem Fall ist das Objekt sowohl auf der linken als auch auf der rechten Regelseite enthalten, d. h., es wird wiederverwendet) oder als grünes Objekt (und mit ++ Annotationen versehen) interpretiert werden kann (in diesem Fall ist das Objekt nur auf der rechten Seite der Regel vorhanden, d. h., es wird erzeugt).

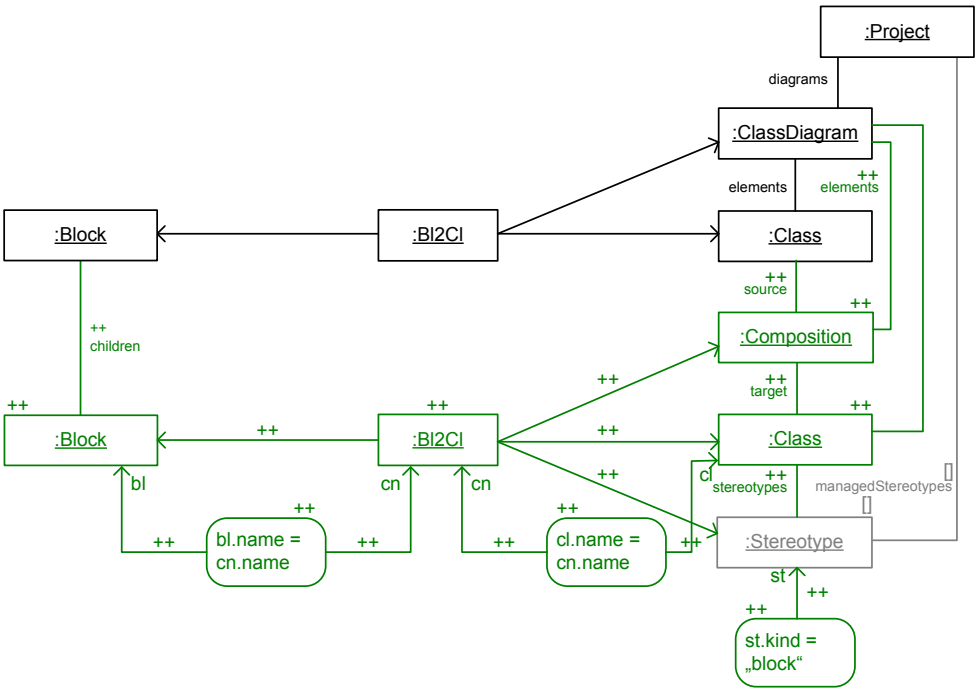


Abbildung 3.22: TGG-Regel mit wiederverwendbarem Stereotyp

Im Gegensatz zu dem ersten Lösungsvorschlag hilft das Konzept der wiederverwendbaren Objekte die Anzahl der TGG-Regeln gering zu halten. Darüber hinaus müssen keine zusätzlichen Bedingungen spezifiziert werden. Dies verringert die Komplexität der Regeln. Ein zusätzlicher Vorteil gegenüber dem ersten Lösungsvorschlag besteht darin, dass die in dieser Arbeit eingeführte Semantik für Bedingungen beibehalten werden kann. Zusam-

men mit dem Konzept der *Bedingungen* bilden wiederverwendbare Objekte eine praxisrelevante Erweiterung, die uns erlaubt, sich auf die wesentlichen Aspekte einer TGG zu konzentrieren.[9]

3.3 Anwendungsszenarien

Im vorangegangenen Abschnitt haben wir die Semantik der TGGs kennengelernt. Diese ist so definiert, dass zwei zueinander in Beziehung gesetzte Modelle simultan durch die Regeln der TGG aufgebaut werden. Dadurch erhalten wir stets zueinander korrespondierende Modelle, bei denen die Korrespondenzbeziehungen zwischen den Modellelementen durch ein Korrespondenzmodell explizit verwaltet werden. Die Semantik einer TGG ist vergleichbar zur Semantik einer klassischen Grammatik, die zur Erzeugung von Wörtern einer Sprache verwendet wird. Im Fall einer TGG sind die Wörter der Sprache allerdings zueinander korrespondierende Modelle.

In der Praxis werden Grammatiken aber nur selten dazu genutzt, um Wörter einer Sprache zu erzeugen. Vielmehr werden Grammatiken dazu verwendet, vorhandene Wörter darauf zu überprüfen, ob sie in der durch die Grammatik definierten Sprache enthalten sind und das Wort in die Struktur – in den meisten Fällen einen Syntaxbaum bzw. Ableitungsbaum – zu parsen[10]. Ähnlich dazu wird auch eine TGG selten zur Konstruktion zweier zueinander korrespondierender Modelle eingesetzt. In den folgenden Unterabschnitten betrachten wir einige typische Anwendungsszenarien für TGGs, zu denen auch die Modellsynchronisation gehört.

3.3.1 Modelltransformation

Ein sehr offensichtliches Anwendungsszenario für TGGs ist die Transformation eines Modells in ein anderes Modell. Diese Anwendung wird als *Modelltransformation* bezeichnet. In dem folgenden Beispiel wird ein Blockdiagramm in ein Klassendiagramm transformiert. Aufgrund der Tatsache, dass in den spezifizierten TGG-Regeln die Domäne der Blockdiagramme auf der

[9] Mit Hilfe der wiederverwendbaren Objekte können zu dem in [Kön08] diskutierten Beispiel TGG-Regeln angegeben werden, mit denen das Modell sowohl erzeugt als auch transformiert werden kann. Die TGG-Regeln kommen dabei gänzlich ohne Negative Anwendungsbedingungen aus.

[10] Genau genommen wird eine Grammatik dazu verwendet, um einen Parser automatisch zu generieren.

linken und die Domäne der Klassendiagramme auf der rechten Seite notiert wurden, was mit unserer natürlichen Leserichtung übereinstimmt, wird diese Transformationsrichtung häufig als *Vorwärtstransformation* bezeichnet. Dabei ist das Blockdiagramm das Quellmodell und das Klassendiagramm das Zielmodell dieser Transformation.

Um diese Transformation durchzuführen, starten wir mit einem Blockdiagramm, das in Abbildung 3.23 in Form eines UML-Objektdiagramms gegeben ist und um das TGG-Axiom erweitert wurde. Auf dieses Objektdiagramm wenden wir nun die TGG-Regeln an, um die fehlenden Objekte im Klassendiagramm und Korrespondenzmodell zu erzeugen.

Abbildung 3.24 zeigt die Situation nach der Anwendung der TGG-Regel für Blöcke (*Block2Class*, vergleiche Abbildung 3.6, Seite 65) auf unsere Ausgangssituation in Abbildung 3.23. In Abbildung 3.25 sehen wir das Ergebnis der zweifachen Regelanwendung auf die in Abbildung 3.24 dargestellte Situation. Wurden alle möglichen Anwendungsstellen berücksichtigt, erhalten wir durch die Regelanwendung für den Kanal, der die beiden Blöcke verbindet, ein zu dem Blockdiagramm korrespondierendes Klassendiagramm wie es in der Abbildung 3.26 zu sehen ist. Dieses Klassendiagramm – zusammen mit dem Korrespondenzmodell – ist das Ergebnis der Modelltransformation.

Die Modelltransformation funktioniert mit denselben TGG-Regeln auch in entgegen gesetzter Richtung, das heißt, wir können ein Klassendiagramm in ein korrespondierendes Blockdiagramm übersetzen. Diese Form der Modelltransformation wird aufgrund der vorherigen Festlegung *Rückwärtstransformation* genannt – auch wenn hier nun genauso gut das Klassendiagramm als Quell- und das Blockdiagramm als Zielmodell definiert werden kann. Aus diesem Grund sprechen wir in dieser Arbeit von Domänen statt von Quell- und Zielmodellen und legen für jede Modelltransformation fest, welche Domäne das Quellmodell und welche Domäne das Zielmodell der Übersetzung ist.

In unserem Beispiel funktioniert die Übersetzung problemlos, da die TGG-Regeln eindeutig sind. Dies ist jedoch nicht immer der Fall. Andere Transformationen können mehrdeutig und damit nicht-deterministisch sein (vgl. z. B. [KW07]). Bei der Spezifikation der Korrespondenzbeziehungen stellt die Mehrdeutigkeit kein Problem dar. Sie ist sogar häufig erwünscht [Bec07]. Für eine effiziente Ausführung der Modelltransformation wird aber eine Strategie benötigt, wie mit Nicht-Determinismus umgegangen werden soll. An dieser Stelle wollen wir uns allerdings nicht im Detail damit befassen, da auf dieses Problem noch später eingegangen wird.

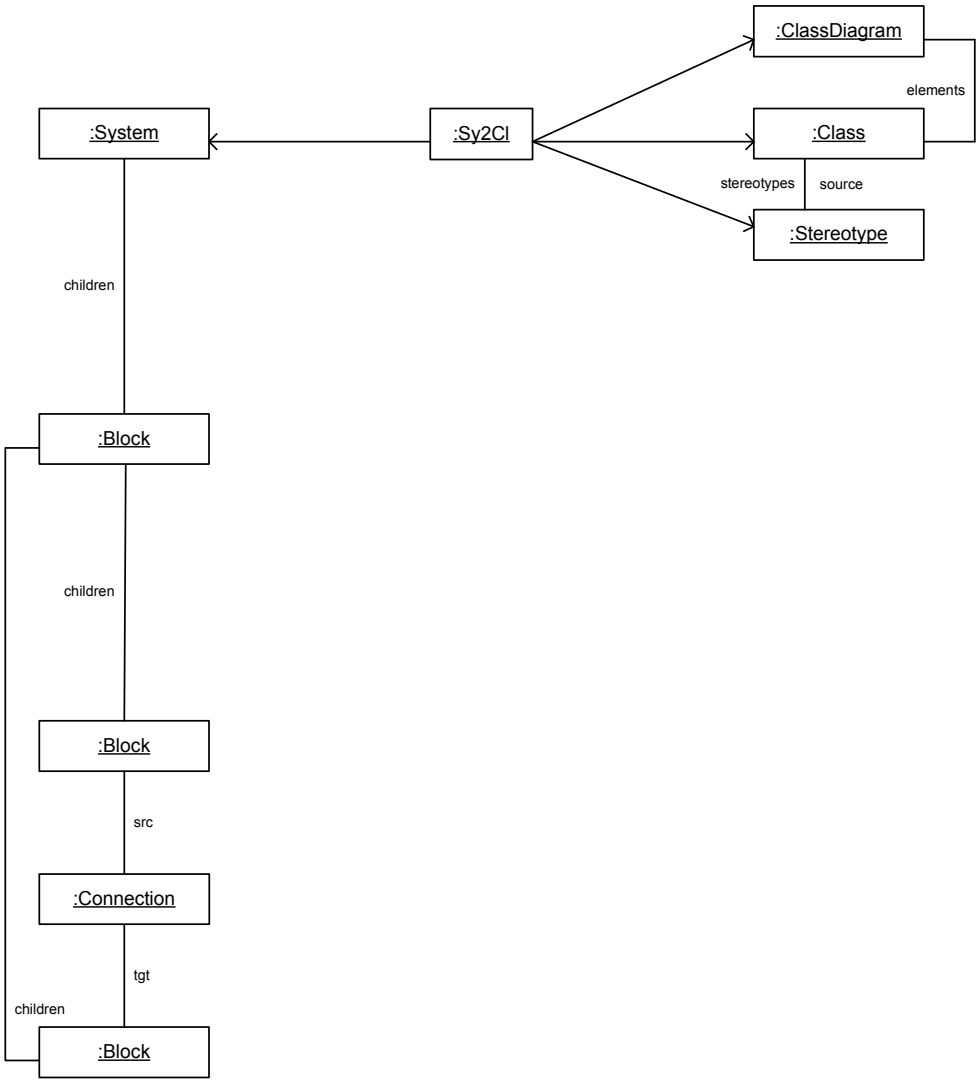


Abbildung 3.23: Modelltransformation: Initiale Startsituation

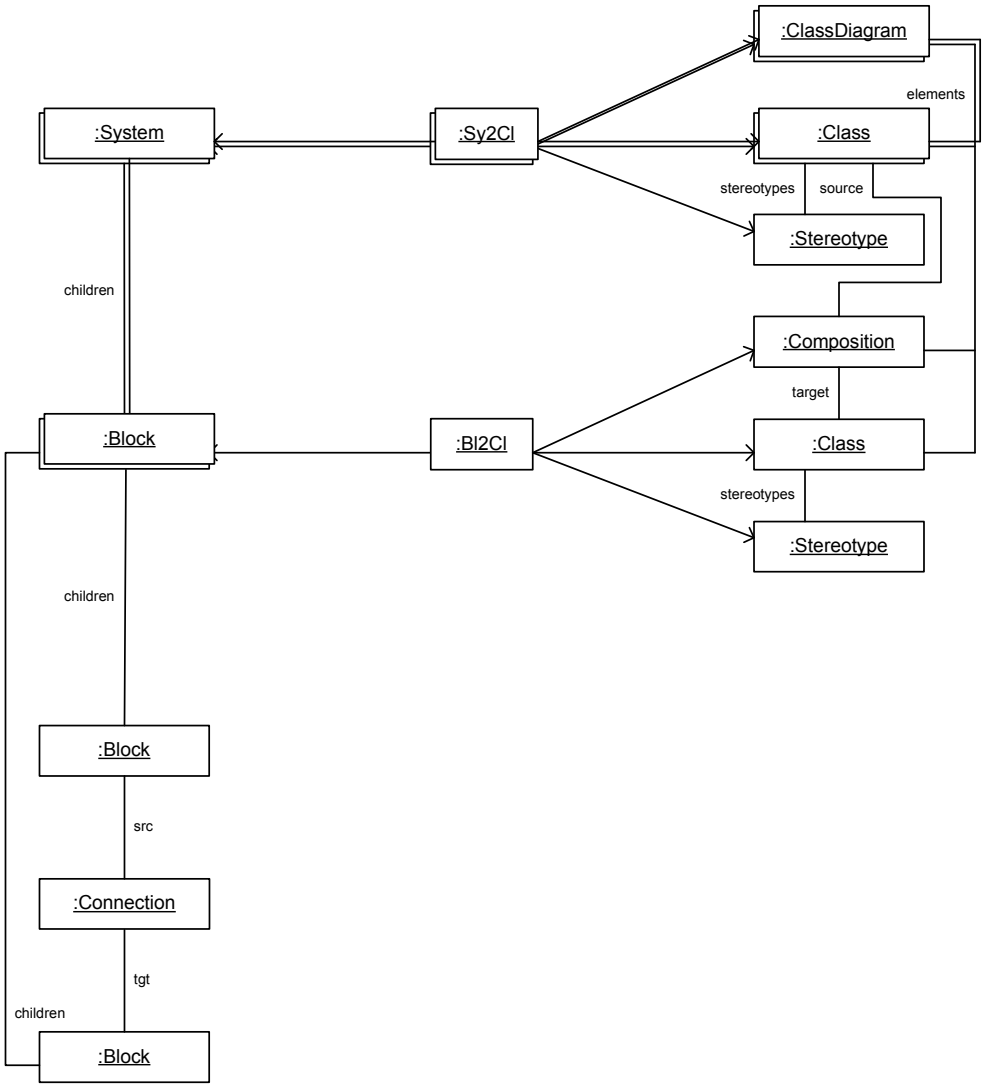


Abbildung 3.24: Modelltransformation: Anwendung der Regel *Block2Class*

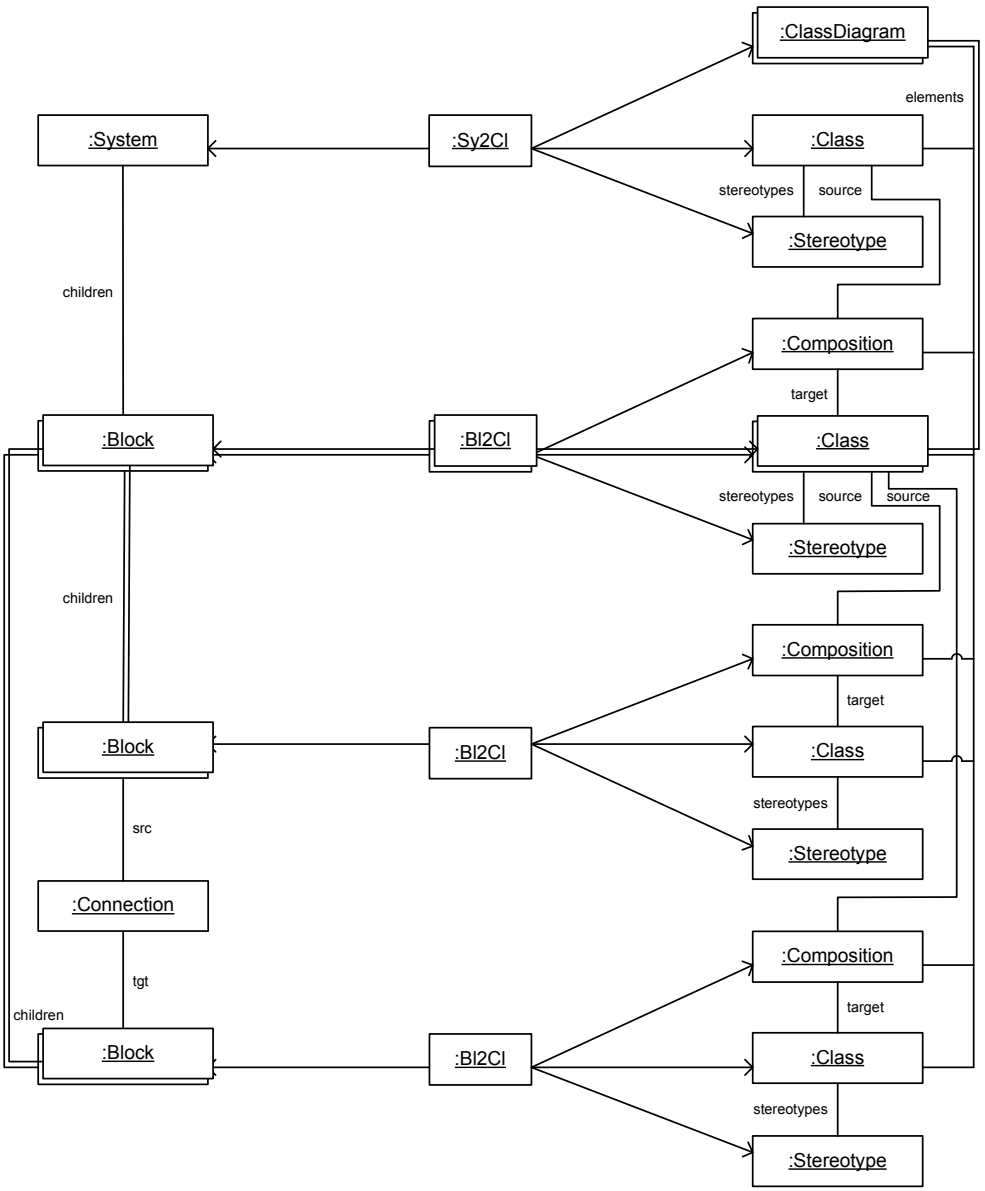


Abbildung 3.25: Modelltransformation: Zweifache Anwendung der Regel *Block2Class*

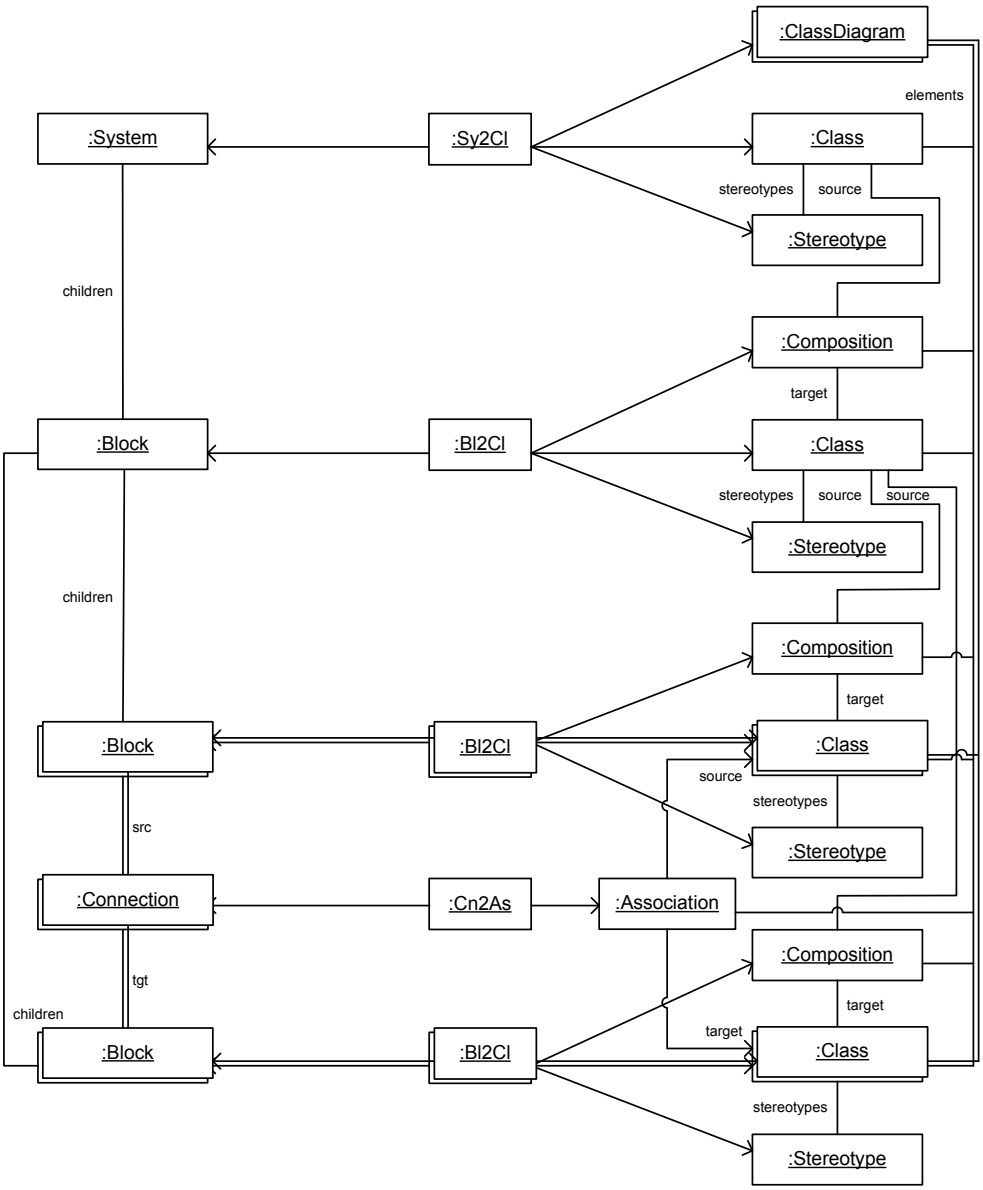


Abbildung 3.26: Modelltransformation: Anwendung der Regel *Channel2Assoc*

3.3.2 Modellintegration

Ein zweites Anwendungsszenario ergibt sich, wenn zwei Modelle gegeben sind und wir die Korrespondenzobjekte zwischen den zueinander in Beziehung stehenden Modellelementen erzeugen möchten. Dieses Szenario entspricht technisch gesehen der Modelltransformation. Der einzige Unterschied ist, dass nun beide Modelle gegeben sind. Diese Modelle können – wie in Abbildung 3.23 dargestellt – mithilfe des TGG-Axioms erweitert werden. Auf diese Anfangssituation können nun wiederum unsere TGG-Regeln angewendet werden. Diesmal suchen wir jedoch in beiden Domänen nach zueinander korrespondierenden Elementen und erzeugen lediglich die Korrespondenzobjekte zwischen zueinander in Beziehung stehenden Modellelementen. In den Abbildungen 3.28-3.30 ist die Anwendung der TGG-Regeln illustriert. Dieses Szenario wird als *Modellintegration* bezeichnet.

In dem in Abbildung 3.30 gezeigten Anwendungsszenario konnten alle Modellelemente der unterschiedlichen Modelle zueinander in Beziehung gesetzt werden. Damit korrespondieren beide Modelle zueinander vollständig – sie sind bezüglich der spezifizierten TGG-Regeln zueinander synchron. Allerdings kann es auch bei diesem Anwendungsszenario vorkommen, dass nicht zwischen allen Modellelementen eine Korrespondenzbeziehung hergestellt werden kann. In diesem Fall sind die Modelle nicht zueinander synchron.

3.3.3 Modellsynchronisation

Die Modellsynchronisation ist das umfassendste Anwendungsszenario. Ausgangspunkt dieses Szenarios sind zwei Modelle, zwischen denen bereits ein entsprechendes Korrespondenzmodell existiert. Dieses Korrespondenzmodell kann durch eine Modelltransformation oder eine Modellintegration entstanden sein. Werden die Modelle vom Benutzer geändert, so sorgt die Modellsynchronisation dafür, dass die modifizierten Modelle wieder miteinander abgeglichen werden. Hierbei wird – sofern notwendig – auch das Korrespondenzmodell aktualisiert. Ein Beispiel für die Modellsynchronisation haben wir bereits in Kapitel 2 gesehen. Wie die Modellsynchronisation technisch auf Grundlage der TGG-Regeln durchgeführt wird, erklärt Kapitel 5.

3.4 Zusammenfassung

In diesem Kapitel haben wir uns mit der Spezifikation von Modellbeziehungen beschäftigt. Hierzu haben wir TGGs eingesetzt. Die grundlegende

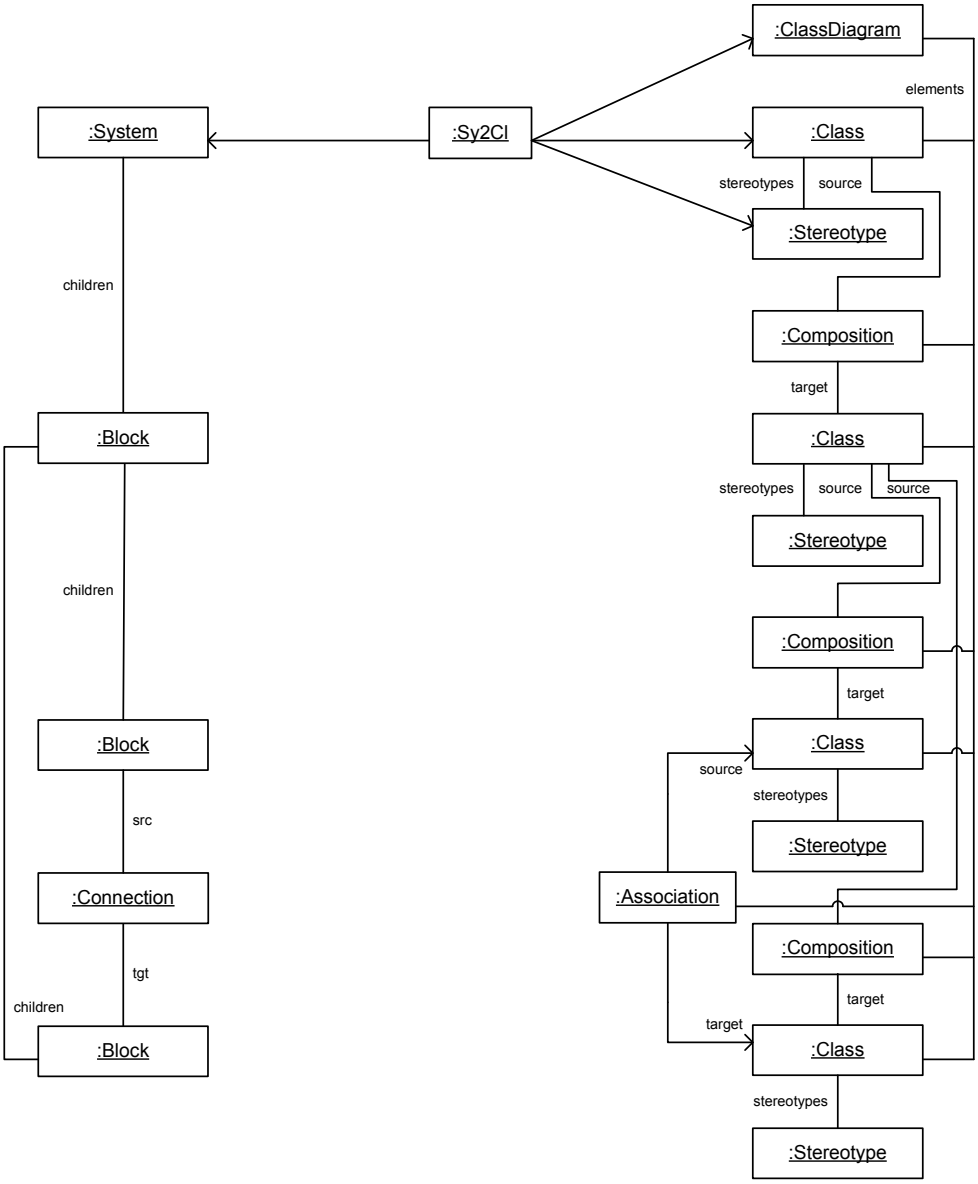


Abbildung 3.27: Modellintegration: Initiale Startsituation

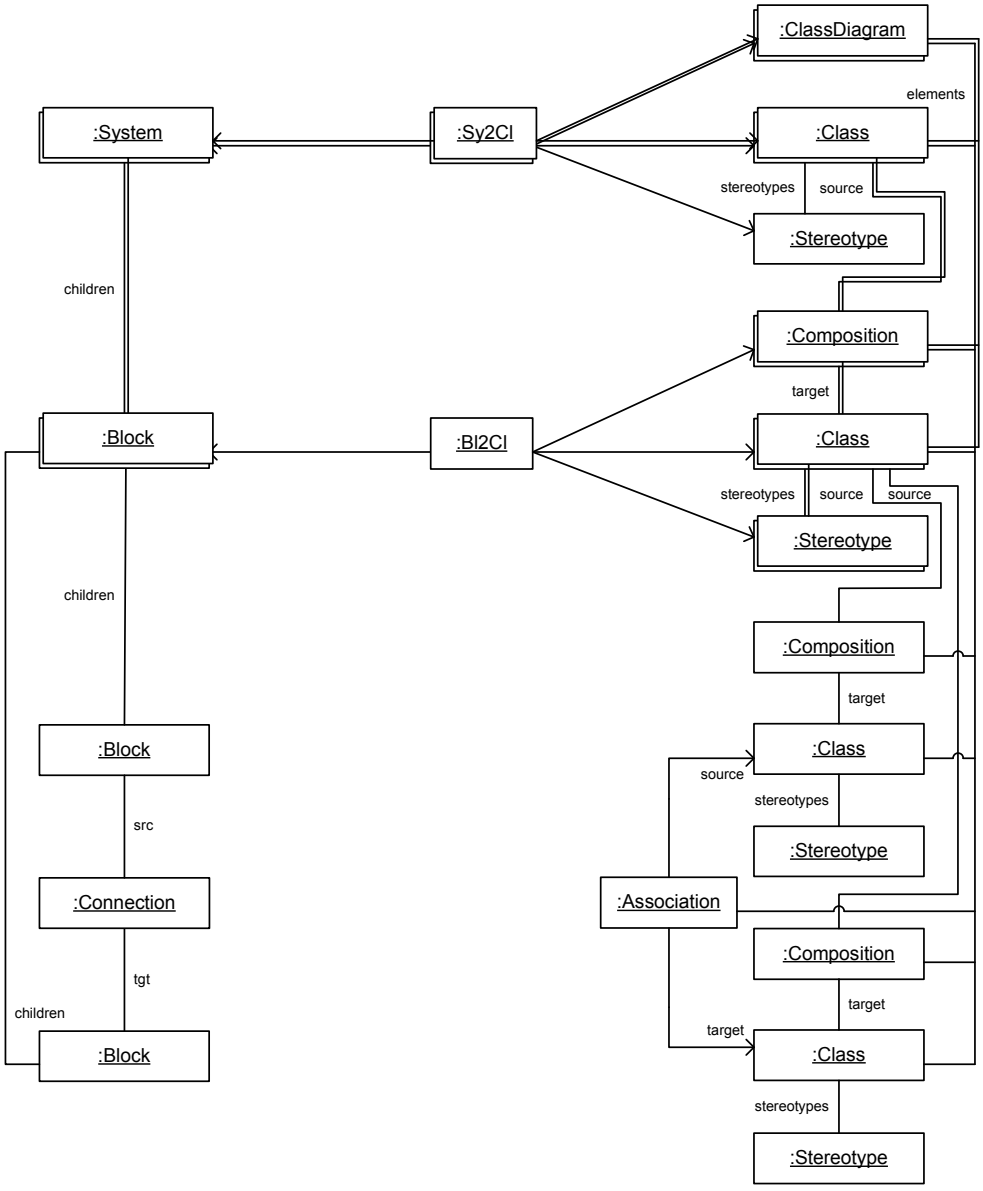


Abbildung 3.28: Modellintegration: Anwendung der Regel *Block2Class*

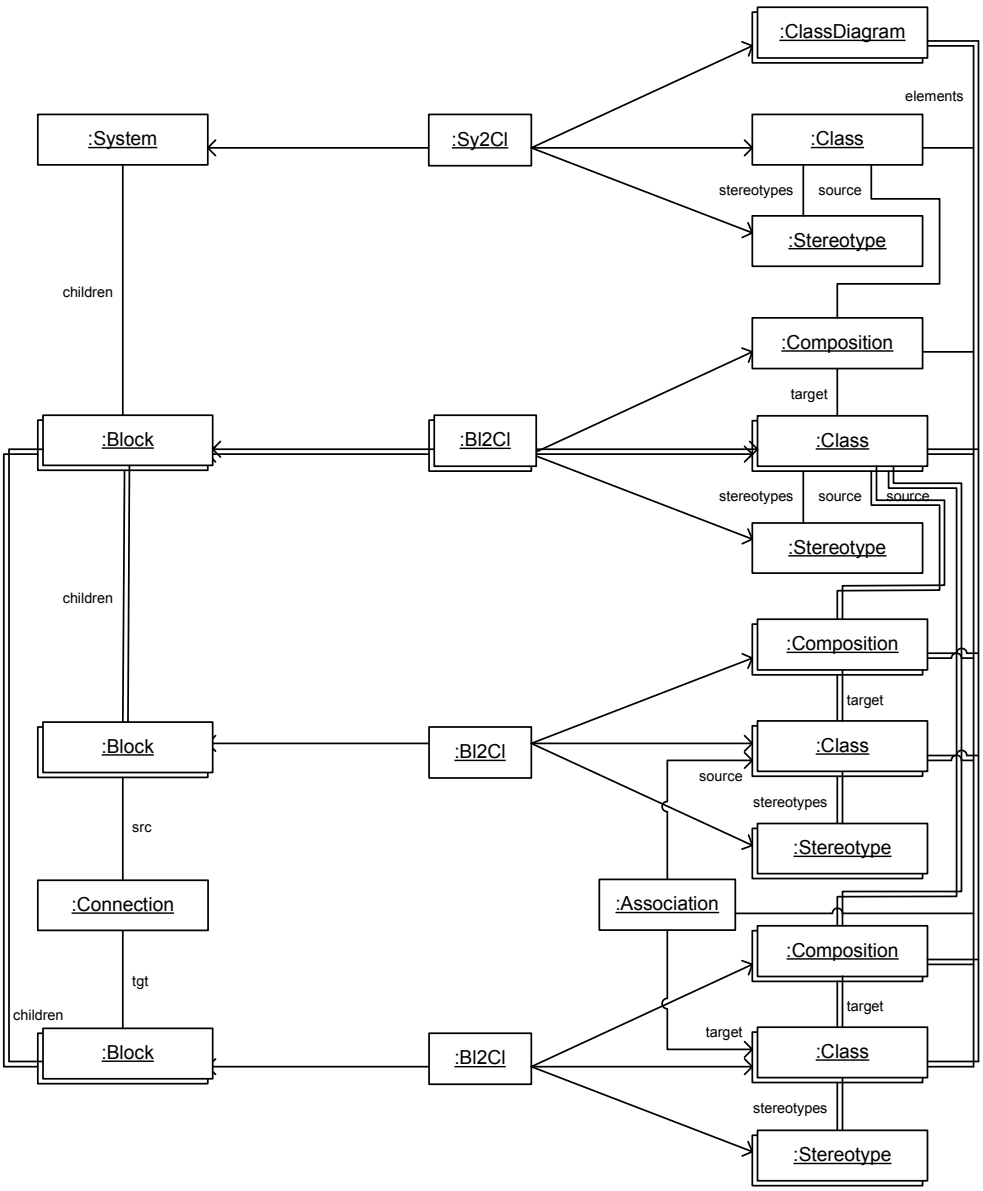


Abbildung 3.29: Modellintegration: Zweifache Anwendung der Regel *Block2Class*

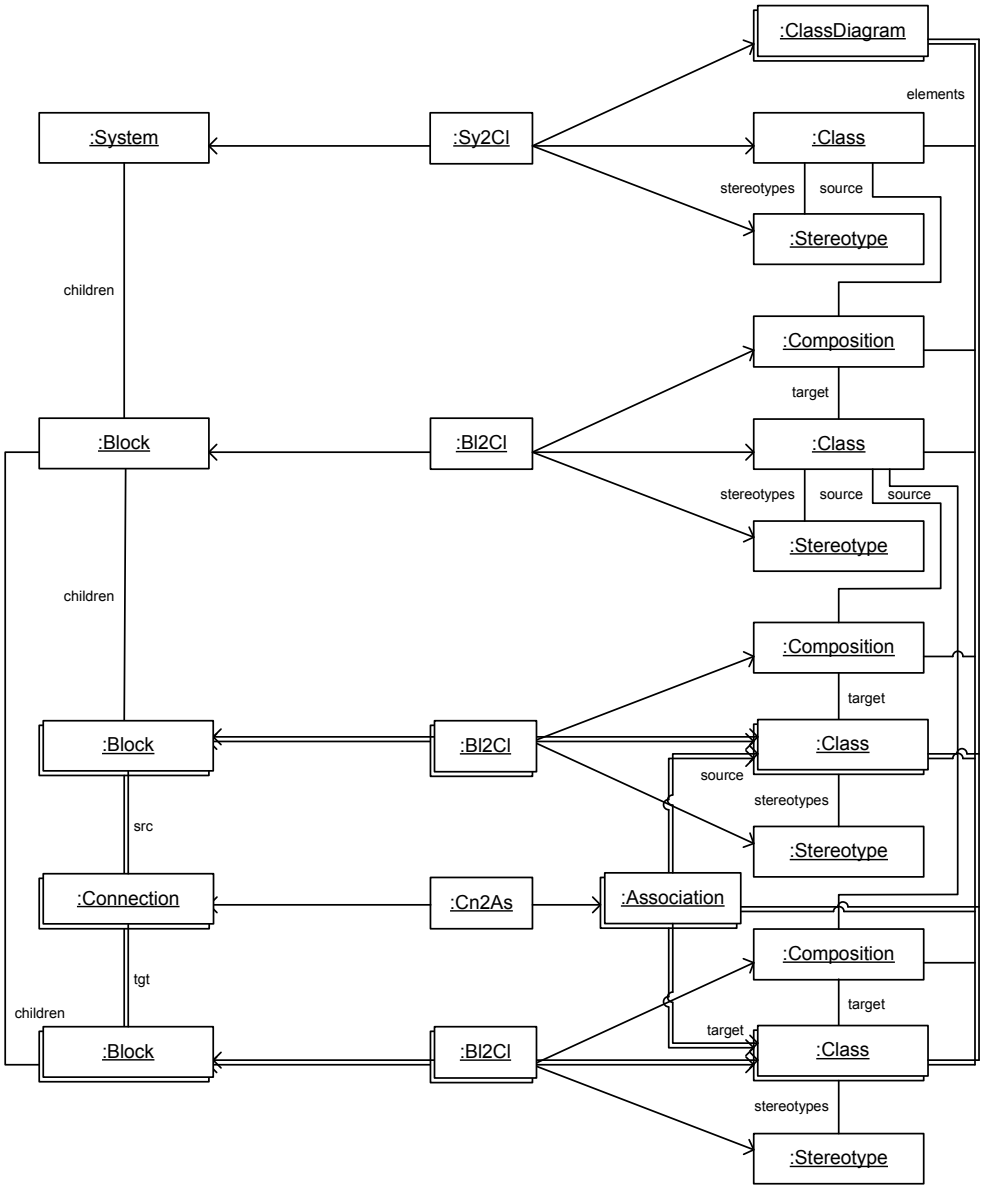


Abbildung 3.30: Modellintegration: Anwendung der Regel *Channel2Assoc*

Idee und die dahinter stehenden Prinzipien haben wir an einem Beispiel beschrieben. Dabei definiert eine Menge von TGG-Regeln – zusammen mit einem TGG-Axiom – die Korrespondenzbeziehungen zwischen den Elementen zweier Modelle. Die Spezifikation der Korrespondenzbeziehungen erfolgt weiterhin in der abstrakten Syntax der beteiligten Modellierungssprachen. Im Hinblick auf die noch vorzustellende Modellsynchronisation haben wir jedoch die Notation und Semantik von Bedingungen geändert sowie ein Konzept zur Wiederverwendung von Objekten eingeführt.

Grundsätzlich ist eine Spezifikation von Korrespondenzregeln nützlich, um Korrespondenzbeziehungen zwischen Modellen explizit zu machen und dadurch zu dokumentieren. In unserem Ansatz erfolgt die Spezifikation graphisch. Die visuell erfassten Korrespondenzbeziehungen sind dadurch leichter für Menschen nachvollziehbar als zum Beispiel Korrespondenzbeziehungen, die nur in einer textuellen Spezifikationssprache hinterlegt sind (vergleiche dazu Spezifikationen in der textuellen Syntax einiger Modelltransformationssprachen, wie zum Beispiel QVT–Relations [QVT08]).

Die Spezifikation von Korrespondenzregeln mit TGGs bietet jedoch noch einige weitere Vorteile. Zunächst ist die Spezifikation der Korrespondenzregeln formal und erfolgt in einer lokalen und deklarativen Art und Weise. Auf dieser Grundlage wird eine inkrementelle und bidirektionale Arbeitsweise der noch später vorzustellenden Modellsynchronisation ermöglicht. Insbesondere bedeutet dies, dass die TGG-Regeln operationalisiert und zur Parametrisierung eines Synchronisationswerkzeugs herangezogen werden können. Zudem hat die bidirektionale und damit richtungsunabhängige Regelspezifikation gegenüber Ansätzen, bei denen für jede Richtung eine eigene Regel notwendig ist, den Vorteil, dass Inkonsistenzen zwischen den unterschiedlichen Richtungen vermieden werden. Ein weiterer Vorteil ist, dass durch die formalen Eigenschaften einer TGG sich verschiedene Möglichkeiten und Ansätze zur Validierung und formalen Verifikation der Korrespondenzbeziehungen eröffnen. Einige dieser Möglichkeiten werden wir in Kapitel 6 aufzeigen. Bevor wir allerdings den Synchronisationsmechanismus sowie Möglichkeiten zur Validierung und Verifikation vorstellen, betrachten wir in Kapitel 4 zuerst einige auf TGGs basierende Spezifikationsvarianten, mit denen die Handhabung weiter vereinfacht wird.

Kapitel 4

Spezifikationsvarianten

Im vorangegangenen Kapitel haben wir die Technik der Tripel-Graph-Grammatiken kennen gelernt, mit der wir Modell-zu-Modell Beziehungen spezifiziert haben. Darauf aufbauend beschäftigen wir uns in diesem Kapitel mit drei Spezifikationsvarianten. In Abschnitt 4.1 fokussieren wir auf die Spezifikation von Modell-zu-Text Beziehungen und zeigen, wie diese Beziehungen mit Tripel-Graph-Grammatiken spezifiziert werden. In Abschnitt 4.2 zeigen wir, wie Korrespondenzbeziehungen mit Hilfe von Beispielzuordnungen definiert werden, um daraus die TGG-Regeln automatisch zu synthetisieren. In Abschnitt 4.3 hingegen gehen wir auf den Transformationsansatz MOF 2.0 Query/View/Transformation (QVT) ein.

4.1 Spezifikation von Modell-zu-Text Beziehungen

Neben Modell-zu-Modell Beziehungen spielen in der modellbasierten Softwareentwicklung Modell-zu-Text Beziehungen eine wichtige Rolle. In der Literatur wird von Modell-zu-Text Beziehungen gesprochen, wenn statt eines graphischen Modells lediglich Textartefakte erzeugt werden [GSCK04, GPR05, CH06]. Die Modell-zu-Text Beziehungen werden beispielsweise dazu verwendet, um aus einem Modell eine textuelle Beschreibung zur Dokumentation des Softwaresystems oder textuelle Konfigurationsdateien automatisch zu erstellen. Die am weitesten verbreitete Anwendung der Texterzeugung ist jedoch die Codegenerierung. Bei der Codegenerierung wird aus einem spezifizierten Modell der zur Implementierung benötigte Code automatisch aus dem Modell abgeleitet und in eine Textdatei geschrieben, so dass bereits existierende Werkzeuge, die eine Textdatei als Eingabe erwarten, weiterhin genutzt werden können.

Die Unterscheidung zwischen einer Modelltransformation und einer Codegenerierung findet überwiegend aufgrund der Art der Sprachdefinition statt, die zur Beschreibung der Zielsprache verwendet wird [CH06]. Während sich zur Sprachdefinition visueller Modellierungssprachen überwiegend Metamodelle durchgesetzt haben, werden Programmiersprachen im Regelfall durch Grammatiken definiert. Ist die Zielsprache durch ein Metamodell definiert, wird die Übersetzung als Modelltransformation bezeichnet. Handelt es sich bei der Zielsprache hingegen um eine Programmiersprache, die durch eine Grammatik definiert ist, wird von Codegenerierung oder auch Modell-zu-Text Transformationen gesprochen [GPR05].

Die Unterscheidung auf Grundlage der Sprachdefinition ist jedoch ungeeignet, weil grundsätzlich sowohl für eine Programmiersprache ein Metamodell angegeben[1] als auch eine visuelle Modellierungssprache durch eine (Graph-) Grammatik definiert werden kann [Roz97]. Daher sollte auf eine solche Unterscheidung verzichtet werden. Diese Haltung ist umso nachvollziehbarer, wenn man bedenkt, dass Code ebenfalls als ein Modell des zu implementierenden Softwaresystems angesehen werden kann. Es wäre daher sinnvoll, wenn keine Unterscheidung zwischen der Spezifikation von Modell-zu-Modell und Modell-zu-Text Beziehungen gemacht würde und die Spezifikationen in einer einheitlichen Notation durchgeführt werden könnten. Dies ist zurzeit jedoch nicht der Fall.

4.1.1 Existierende Techniken

In diesem Abschnitt beschäftigen wir uns mit zwei weit verbreiteten Techniken zur Spezifikation von Modell-zu-Text Beziehungen. Diese Ansätze werden überwiegend zur Codegenerierung eingesetzt. Dabei handelt es sich um sehr technisch orientierte Ansätze, die durch zahlreiche Rahmenwerke (engl. Frameworks) unterstützt werden. An dieser Stelle gehen wir jedoch nicht auf die Details der verschiedenen Rahmenwerke ein. Stattdessen stellen wir nur die zugrunde liegenden Konzepte vor, um anschließend die damit verbundenen Vor- und Nachteile zu diskutieren.

Direkte Programmierung

Eine Möglichkeit zur Realisierung einer automatischen Codegenerierung besteht darin, den benötigten Codegenerator direkt in einer Programmiersprache von Hand zu implementieren. Hierzu durchläuft der Codegenerator die

[1] siehe zum Beispiel das Eclipse Metamodell zur Programmiersprache Java

interne Struktur eines Modells und erzeugt aus den Daten der traversierten Modellartefakte den dazugehörigen Code. Die Erzeugung der Textartefakte erfolgt zumeist durch einfache `println`-Anweisungen, deren Ausgabe in eine Textdatei umgeleitet wird. Die auf diesem Ansatz basierenden Codegeneratoren werden daher häufig auch als *Line-Printer* bezeichnet [GPR05].

Die Implementierung eines solchen Codegenerators kann durch den Einsatz von Entwurfsmustern vereinfacht werden. Beispielsweise kann zur Traversierung der Modelle das *Visitor*-Entwurfsmuster eingesetzt werden [GHJV94]. Dieses Entwurfsmuster erlaubt es, die Operationen zur Codegenerierung an einer einzigen Stelle zu kapseln und erleichtert dadurch sowohl die Programmierung als auch die spätere Wartung des Codegenerators. Zusätzlich können durch den Einsatz dieses Entwurfsmusters weitere Codegeneratoren nachträglich hinzugefügt werden, ohne dazu die bestehende Implementierung ändern zu müssen.

Zur Unterstützung der Implementierung können außerdem Rahmenwerke verwendet werden, die auf die Codegenerierung – oder ganz allgemein auf die Generierung von Textartefakten – spezialisiert sind. Die Rahmenwerke bieten eine API an, um den Zugriff auf die interne Modellrepräsentation und die damit verbundene Navigation im Modell zu vereinfachen, so dass dadurch die Programmierung des Codegenerators insgesamt erleichtert wird. Ein prominentes Beispiel für ein solches Rahmenwerk ist Jamda [Jam]. Dieses Rahmenwerk ist auf die Entwicklung von Codegeneratoren spezialisiert, die Code aus UML-Modellen erzeugen.

Die direkte Programmierung eines Codegenerators ist der flexibelste Ansatz. Allerdings ist mit der Programmierung von Hand ein sehr hoher Aufwand verbunden. Dieser Aufwand lässt sich zwar durch den Einsatz spezieller Entwurfsmuster und Rahmenwerke reduzieren, die eigentliche Entwicklung des Codegenerators findet aber weiterhin auf einem sehr niedrigen Abstraktionsniveau statt. Zudem sind die Rahmenwerke auf ganz bestimmte Metamodelle oder Technologien spezialisiert, wodurch ihre Verwendung nur in einem bestimmten Kontext sinnvoll beziehungsweise möglich ist.

Die Anpassung und Wartung eines solchen Codegenerators erfolgt direkt im Programmtext. Um die Zuordnung zwischen einem Modell und dem daraus generierten Code zu verstehen, muss die Implementierung des Codegenerators analysiert werden. Der Zusammenhang zwischen einem Modell und dem daraus zu generierenden Code ist dabei nur sehr schwer erkennbar, so dass sich die Anpassung und Wartung der Codegenerierung als sehr schwierig erweist. Eine Anpassung der Codegenerierung durch einen Benutzer ist daher unrealistisch.

In einem engen Zusammenhang mit der Zuordnung von Modell-zu-Text Beziehungen steht die Nachverfolgbarkeit (engl. Traceability). Wie wir bereits in Abschnitt 1.2.3 gesehen haben, ist die Nachverfolgbarkeit sehr nützlich. Sofern die Zuordnung nicht explizit bei der Generierung gespeichert wird, ist nach der Codegenerierung nicht direkt erkennbar, aus welchen Modellartefakten der Code erzeugt wurde. Insbesondere wird aufgrund der fehlenden Zuordnung eine Synchronisation zwischen Modell und Code erschwert. Sofern eine solche Synchronisation benötigt wird, muss sie zusätzlich von Hand implementiert werden. Soll die Synchronisation nicht ganz unabhängig vom generierten Code erfolgen, so erhöht dies die Komplexität des Codegenerators. Eine vom Codegenerator unabhängige Implementierung hingegen birgt die Gefahr, dass das Forward-Engineering – in diesem Fall also die Codegenerierung – inkonsistent zum Reverse-Engineering implementiert und die Synchronisation damit fehlerhaft ist. Die Komplexität dieser Aufgabe wird dadurch belegt, dass heutige Werkzeuge keine zufriedenstellende Lösung zur Synchronisation von Modell und Code anbieten.

Zusammenfassend kann man sagen, dass die direkte Programmierung eines Codegenerators nur dann empfehlenswert ist, wenn entweder die Codegenerierung extrem effizient sein muss oder wenn der Zusammenhang zwischen den Modellen und dem daraus zu generierendem Code so komplex ist, dass er durch andere Ansätze nicht abgebildet werden kann.

Textschablonen

Wegen der Nachteile der direkten Programmierung haben sich in der Praxis Ansätze auf der Grundlage von Textschablonen zur Codegenerierung durchgesetzt. Zu den Rahmenwerken, die Textschablonen unterstützen, gehören unter anderem *Java Emitter Template* (JET) [JET], *AndroMDA* [AMD], *Velocity Template Engine* [VTE], *openArchitectureWare* [OAW], *ArcStyler* [ARC], *OptimalJ* [OpJ] und *Codagen Architect* [CoG]. Mit der im Januar 2008 veröffentlichten Spezifikation der *MOF Model to Text Transformation Language* (MOFM2T) liegt auch ein Standard der Object Management Group (OMG) vor [MTT08], in dem eine Spezifikationssprache für Modell-zu-Text Transformation auf der Grundlage von Textschablonen definiert wird.

Abbildung 4.1 zeigt schematisch den Ansatz zur Codegenerierung mit Textschablonen. Die Eingabe für die Codegenerierung ist – neben einer Menge von vordefinierten Textschablonen – ein Modell, für das Code erzeugt werden soll. Die Anwendung der Textschablonen auf das Modell er-

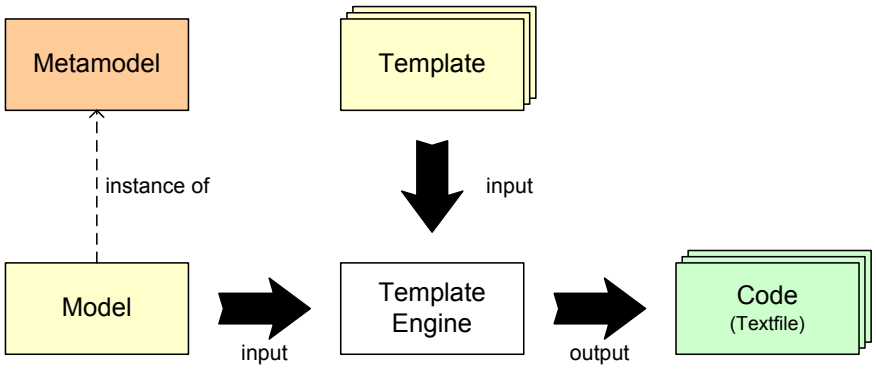


Abbildung 4.1: Codegenerierung mit Textschablonen

folgt durch ein Rahmenwerk, das als *Template Engine* bezeichnet wird. Die Ausgabe besteht in der Regel aus einer Menge von Textdateien, die den erzeugten Code enthalten.

Bei einer Textschablone (engl. Template) handelt es sich um eine Textvorlage mit vordefinierten Textfragmenten, durch die ein Codegerüst vorgegeben wird. Dieses Codegerüst wird im Rahmen der Codegenerierung mit Daten aus dem Modell angereichert und in einer Textdatei gespeichert. In der Abbildung 4.2 ist beispielhaft ein Auszug aus einer Textschablone der *Velocity Template Engine* dargestellt [VTE]. Die vollständige Textschablone ist in der Studienarbeit von Markus von Detten zu finden [Det06]. Sie wurde im Rahmen des ISILEIT-Projektes zur Generierung von Code für Speicherprogrammierbare Steuerungen (SPSen) verwendet. Der hier gezeigte Ausschnitt der Textschablone besteht aus zu erzeugenden Textfragmenten der SPS-Programmiersprache *Strukturierter Text* (ST) [IEC03], Platzhaltern sowie Anweisungen, die unter anderem zur Steuerung der Codegenerierung verwendet werden.

Platzhalter sind durch ein vorangestelltes `$`-Zeichen zu erkennen. Sie werden während der Codegenerierung durch konkrete Werte ersetzt. Aufgrund der Tatsache, dass es sich hierbei um eine Java-basierte Template Engine handelt, können auf einem Platzhalter, dem während der Ausführung ein Java-Objekt zugeordnet wurde, Methoden aufgerufen werden. Nach dem Aufruf wird der Rückgabewert der Methode für den Platzhalter eingesetzt (vgl. zum Beispiel `$state.getName()`). Die Anweisungen hingegen sind an einem vorangestellten `#`-Zeichen zu erkennen. Dabei kann es sich um Anweisungen zur Steuerung der Codegenerierung handeln, wie zum Beispiel

```
...
CASE state OF
  ...
  #foreach($state in $automaton)
     #getId($state) (* $state.getName() *) :
     #if($velocityCount == 1)
        #foreach($transition in $state.getOutgoingTransitions())
           IF #getMapping($transition.getTrigger())
           THEN
             #foreach($action in $transition.getActions())
             #getMapping($action)
             #end

             state := #getId($transition.getTarget());
        #end
     ...
  #end
END_CASE;
...
```

Abbildung 4.2: Ausschnitt aus einer Textschablone

`#foreach` und `#if`, oder um benutzerdefinierte Direktiven, wie zum Beispiel `#getMapping` und `#getId`, die in Form von Makros hinterlegt wurden.

Im Gegensatz zur direkten Programmierung ist eine Anpassung der Codegenerierung bei diesem Ansatz relativ einfach möglich. Hierzu reicht es häufig aus, die Textfragmente der Textschablone zu verändern und an die speziellen Bedürfnisse anzupassen. In einigen Fällen muss allerdings zusätzlich ein Algorithmus bereit gestellt werden, der die zur Codegenerierung benötigten Informationen aus einem Modell extrahiert. Zwar existieren auch Rahmenwerke, bei denen die Traversierung der Modelle bereits implementiert ist [JET, AMD], allerdings sind diese auf eine bestimmte Technologie, wie zum Beispiel das Eclipse Modeling Framework (EMF) oder eine bestimmte Modellierungssprache, wie zum Beispiel die Unified Modeling Language (UML), spezialisiert. Eine Codegenerierung für andere Technologien oder Modellierungssprachen ist damit nicht realisierbar.

Die Zuordnung der Modell-zu-Text Beziehungen ist bei diesem Ansatz nur schwer zu erkennen. Der Zusammenhang zwischen dem generierten Text

und der dazu verwendeten Textschablone ist hingegen recht gut erkennbar, auch wenn dies durch die zusätzlichen Anweisungen in der Textschablone erschwert wird. Auffällig an dem Ansatz ist auch, dass zwar die Syntax der Modelle durch ein Metamodell vorgegeben ist, der erzeugte Text aber lediglich aus Textfragmenten besteht, die nicht typisiert sind. Damit kann einerseits beliebiger Text erzeugt werden. Andererseits wird bei der Generierung von Code der erzeugte Programmtext keiner syntaktischen Überprüfung unterzogen. Somit ist es mit diesem Ansatz möglich syntaktisch fehlerhafte Programme zu erzeugen.

Grundsätzlich lässt sich die Entwicklung eines Codegenerators durch den Einsatz von Textschablonen vereinfachen. Wie schon bei der direkten Programmierung werden bei diesem Ansatz allerdings nur unidirektionale Modell-zu-Text Transformationen unterstützt, die weder eine Nachverfolgbarkeit gewährleisten noch eine inkrementelle Synchronisation der Modell-zu-Text Beziehungen ermöglichen.

4.1.2 Spezifikation mit Tripel-Graph-Grammatiken

Die Technik der Tripel-Graph-Grammatiken kann auch zur Spezifikation von Modell-zu-Text Beziehungen eingesetzt werden. Wie schon bei der Spezifikation von Modell-zu-Modell Beziehungen können auch hier die TGGs bidirektional ausgeführt werden. Durch das Korrespondenzmodell bleibt die Zuordnung zwischen den Modell- und den Textartefakten nach einer Transformation bestehen. Damit ist die Nachverfolgbarkeit einer Transformation gegeben. Daher sind TGGs sowohl zur Codegenerierung als auch zur Synchronisation von Modell und Code geeignet.

Im ersten Teil dieses Abschnitts stellen wir die Spezifikation und Synchronisation von Modell-zu-Text Beziehungen mit TGGs vor. Die Spezifikation unterscheidet sich nicht von der Spezifikation von Modell-zu-Modell Beziehungen. Allerdings sind einige Besonderheiten bei der Ausführung zu beachten, um eine Synchronisation zwischen einem Modell und dem dazugehörigen Text durchführen zu können. Ist hingegen eine Synchronisation nicht notwendig, so kann der Spezifikationsaufwand reduziert werden, indem TGGs mit Textschablonen kombiniert werden. Mit dieser Spezifikationsvariante beschäftigen wir uns im zweiten Teil dieses Abschnitts.

Direkte Spezifikation

Die Spezifikation mit TGGs erfolgt auf der Grundlage von Metamodellen. Bei der Spezifikation von Modell-zu-Modell Beziehungen sind die Metamodelle durch die beiden Modellierungssprachen gegeben. Handelt es sich hingegen um Modell-zu-Text Beziehungen, so muss für den Text zunächst eine geeignete Repräsentation gefunden werden, auf deren Basis eine solche Spezifikation statt finden kann. Dies bedeutet, dass zur Spezifikation von Modell-zu-Text Beziehungen neben dem Metamodell für die Modellierungssprache auch ein Metamodell für den zu generierenden Text vorhanden sein muss.

Damit die Textartefakte einer Sprache oder einer Beschreibung maschinell auswertbar sind, müssen sie in einer geeigneten internen Repräsentation vorliegen. Bei einem einfachen Text ist häufig eine anwendungsspezifische Datenstruktur vorhanden, die zum Beispiel aus Kapiteln, Absätzen, Überschriften, Zeichen und Formatierungen bestehen kann. Diese Datenstruktur kann bereits als ein Metamodell angesehen und zur Spezifikation der Modell-zu-Text Beziehungen eingesetzt werden. Handelt es sich bei dem Text hingegen um Code, so wird der Code in den meisten Fällen in Form eines abstrakten Syntaxbaums (engl. Abstract Syntax Tree (AST)) repräsentiert.

Der Code liegt in der Regel zeilenweise als eine einfache Folge von Zeichen vor. Ein abstrakter Syntaxbaum hingegen repräsentiert die syntaktische Struktur und ist allgemein betrachtet eine Zwischendarstellung (engl. Intermediate Representation, (IR)) von Code [ASU86]. Diese Zwischendarstellung wird mithilfe eines *Parsers* erzeugt, der zur Erkennung und Überprüfung der Syntax eines Programms eingesetzt wird.

Heutige Parser werden nicht von Hand programmiert, sondern mit einem *Parsergenerator* erzeugt. Hierzu erhält der Parsergenerator als Eingabe die Grammatik der Sprache, aus der er dann einen Parser für diese Sprache automatisch generiert. Neben der Erkennung und Prüfung der Syntax wird durch den generierten Parser auch der abstrakte Syntaxbaum erzeugt. Die hierfür notwendigen Klassen werden häufig durch den Parsergenerator selbst (wie zum Beispiel im Fall des Parsergenerators *ANTLR* [Par07]) bereitgestellt oder durch ein externes Werkzeug erzeugt (wie zum Beispiel im Fall des *Java Tree Builders* (JTB) für den Parsergenerator JavaCC [JTB, JCC]). Aufgrund der Tatsache, dass die abstrakte Syntax einer Sprache gleichwertig mit einem Metamodell zu setzen ist, können diese Klassen als Metamodell der Sprache angesehen und zur Spezifikation der Modell-zu-Text Beziehungen verwendet werden. Abbildung 4.3 zeigt beispielhaft einen Ausschnitt für das Java-Metamodell von Eclipse [JDT]. Andere Beispiele für solche Meta-

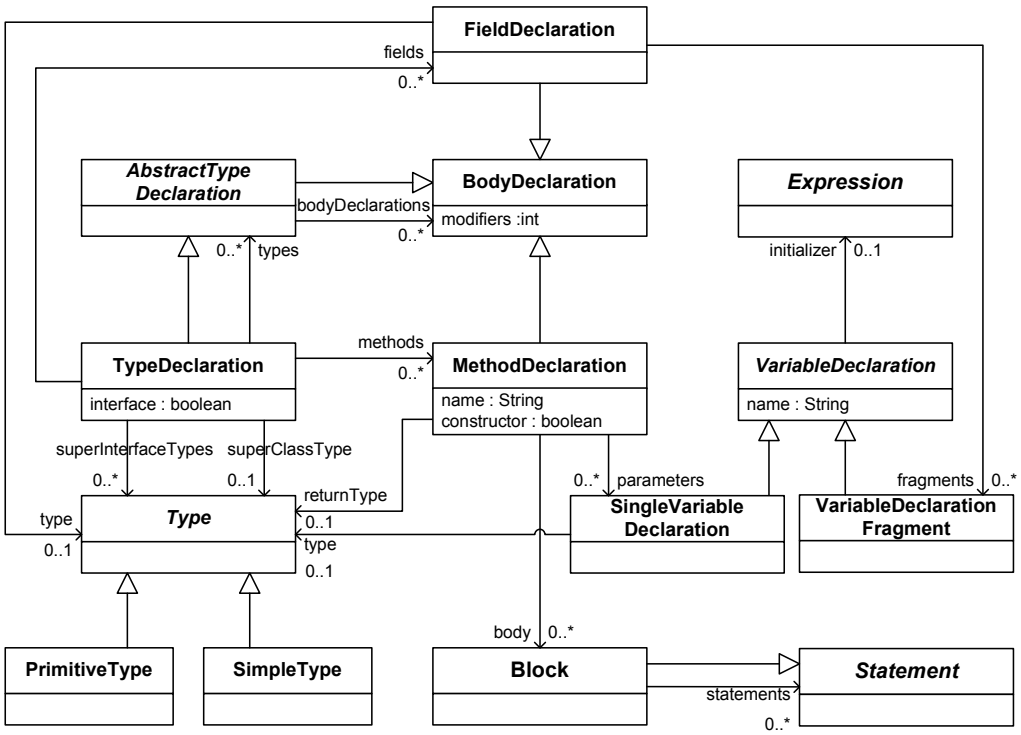


Abbildung 4.3: Ausschnitt aus dem Eclipse Java-Metamodell

modelle sind die – wenn auch nicht vollständigen – Metamodelle für Java, Cobol, PL/I, C und C++ der Object Management Group (OMG). Diese Metamodelle wurden im Rahmen verschiedener OMG-Standards veröffentlicht [OMG04, OMG04].

Ist ein Metamodell für eine textuelle Sprache gefunden, so kann die Spezifikation der Modell-zu-Text Beziehungen mit Hilfe der TGGs stattfinden. In Abbildung 4.4 ist eine Beispielregel dargestellt[2]. Diese TGG-Regel basiert auf dem UML-Metamodell für Klassendiagramme aus Abbildung 3.1 sowie dem Java-Metamodell von Eclipse aus Abbildung 4.3. Sie setzt Attribute einer Klasse zu Fragmenten eines abstrakten Syntaxbaums für Java-Code in Beziehung. Sind weitere TGG-Regeln definiert, so kann aus einem Modell anhand dieser TGG-Regeln ein abstrakter Syntaxbaum erstellt und somit eine Codegenerierung durchgeführt werden.

Abbildung 4.5 illustriert die prinzipielle Arbeitsweise der Codegenerie-

[2]Diese TGG-Regel ist recht übersichtlich. Aufgrund der sehr umfangreichen Syntax bestehen die meisten TGG-Regeln aus 20-30 Knoten.

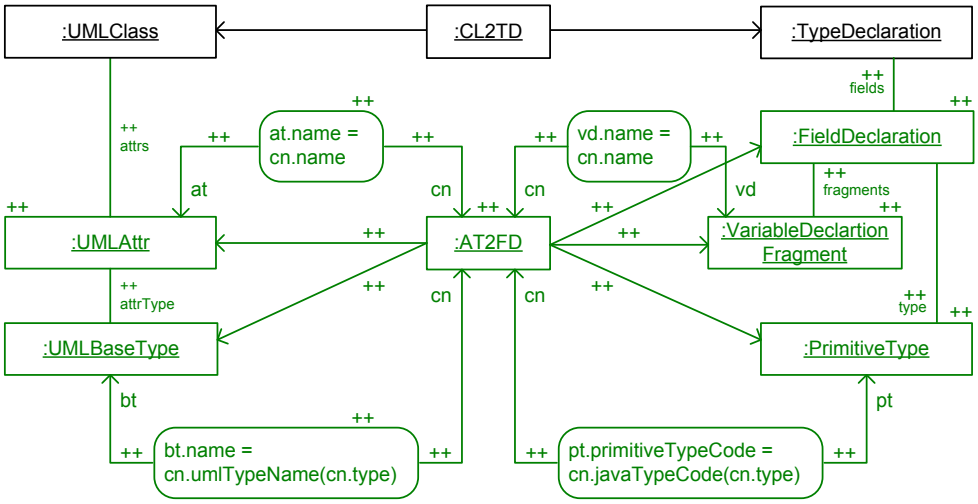


Abbildung 4.4: Beispiel für die Spezifikation von Modell-zu-Text Beziehungen mit einer TGG-Regel

rung sowie der Synchronisation auf Grundlage der mit TGGs spezifizierten Modell-zu-Text Beziehungen. In unserem Beispiel nehmen wir an, dass zunächst nur ein Modell gegeben ist. Nach der Ausführung der TGG-Regeln auf dem Modell (Schritt 1) existiert sowohl ein abstrakter Syntaxbaum als auch ein Korrespondenzmodell, welches die Zuordnung zwischen Elementen des Modells und des abstakten Syntaxbaums explizit verwaltet. Der abstrakte Syntaxbaum repräsentiert die syntaktische Struktur des generierten Codes. Für die weitere Verarbeitung durch Werkzeuge, wie zum Beispiel Editoren und Compiler, wird der Code jedoch in seiner konkreten Syntax, also seiner textuellen Darstellung, benötigt.

Um eine textuelle Darstellung aus einem abstrakten Syntaxbaum zu erhalten, werden sogenannte *Unparser*[3] verwendet (Schritt 2) [ELI]. Analog zu Parsern existieren auch für Unparser entsprechende Generatoren [Kas94, ELI, TXL], die aus einer Spezifikation den Unparser automatisch erstellen.

Für unser Beispiel müssen wir keinen Unparser explizit erstellen. Er wird bereits durch die *Java Development Tools (JDT)* der Eclipse-Entwicklungsumgebung zur Verfügung gestellt [JDT]. Dieser Unparser hat die Eigenschaft, dass nach dem Parsen von Java-Code und einem soforti-

[3]Unparser werden beispielsweise sehr häufig im Rahmen der automatischen Codeformatierung (engl. Pretty-Printer) eingesetzt.

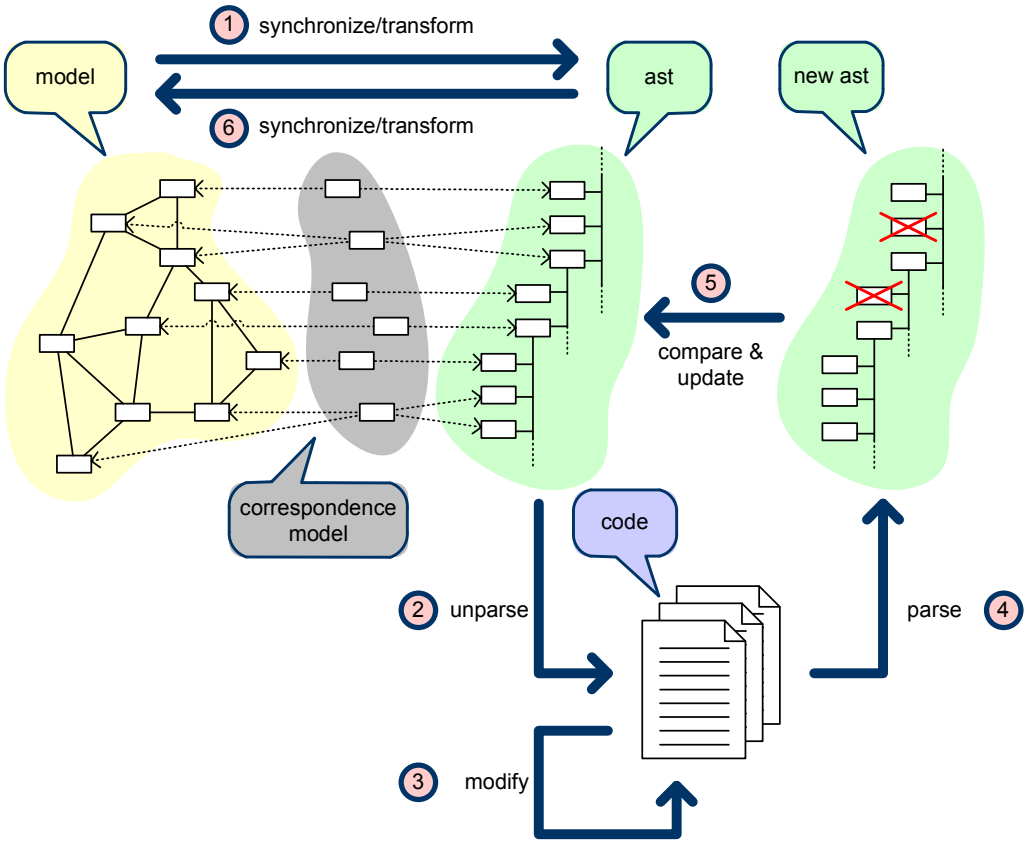


Abbildung 4.5: Codegenerierung und Synchronisation von Modell-zu-Text Beziehungen

gen Unparsen keine Änderungen an der textuellen Repräsentation des Codes entstehen. Dies wird durch entsprechende Annotationen im abstrakten Syntaxbaum erreicht [KL03], so dass bestehende Einrückungen, Leerzeichen, Kommentare sowie Klammerungen von Ausdrücken beim Unparsen berücksichtigt werden und die originale Formatierung des Codes nicht verloren geht. Im Zusammenhang mit der Synchronisation von Modell und Code ist dies besonders vorteilhaft, weil lokale Änderungen im abstrakten Syntaxbaum sich ebenfalls nur lokal an den betroffenen Stellen im Code auswirken. Bei einem einfachen abstrakten Syntaxbaum ohne Annotationen fehlen diese zusätzlichen Informationen, so dass auch von Änderungen nicht betroffene Teile des Codes durch den Unparser umformatiert werden.

Beim heutigen Stand der modellbasierten Softwareentwicklung müssen wir immer noch davon ausgehen, dass Änderungen nicht nur am Modell durchgeführt werden, sondern dass auch der generierte Code bearbeitet wird (Schritt 3). Beispielsweise muss ein Entwickler nach der initialen Codegenerierung häufig weitere Methoden hinzufügen, um zusätzliche Funktionalität zu integrieren. In einem iterativ-inkrementellen Softwareentwicklungsprozess ist darüber hinaus zu berücksichtigen, dass nach der Codegenerierung (und den eventuell durchgeführten Änderungen im Code) ebenso Änderungen im Modell vorgenommen werden. Änderungen im Modell können beispielsweise durch neue Erkenntnisse aus der Analysetätigkeit notwendig werden. Im Rahmen der Synchronisation müssen daher sowohl Änderungen im Modell als auch im Code berücksichtigt werden.

Zur Bearbeitung von Code werden Texteditoren eingesetzt. Grundsätzlich kann hier in *syntaxgesteuerte Texteditoren* und *konventionelle Texteditoren* unterschieden werden. Syntaxgesteuerte Texteditoren unterstützen die korrekte Erstellung von Code gemäß der formalen Syntax der zugrundeliegenden Programmiersprache. Bei einem solchen Editor wird bei der Bearbeitung des Codes die abstrakte Syntax des Programms direkt manipuliert. Zur Darstellung des Programmtextes gegenüber dem Benutzer wird ein Unparser verwendet, der aus dem abstrakten Syntaxbaum die textuelle Repräsentation des Codes erzeugt. Im Gegensatz dazu wird der Code in einem konventionellen Texteditor intern als eine Folge von Zeichen verwaltet, die durch Benutzereingaben verändert wird. Damit hat die Bearbeitung des Programmtextes in einem konventionellen Editor keinen direkten Einfluss auf den abstrakten Syntaxbaum.

Bei der Synchronisation mit TGGs ist es notwendig, dass die durch das Korrespondenzmodell hergestellte Zuordnung zwischen Modell und Code erhalten bleibt. Daher ist ein syntaxgesteuerter Texteditor zur Manipulation

des Codes vorzuziehen. Syntaxgesteuerte Texteditoren sind jedoch immer nur auf eine bestimmte Programmiersprache spezialisiert. Darüber hinaus legen sie dem Benutzer viele Restriktionen auf, so dass die Arbeit mit ihnen gewöhnungsbedürftig ist. Hinzu kommt, dass sie aufwändiger zu implementieren sind. Aus diesen Gründen haben sich syntaxgesteuerte Texteditoren in der Praxis nicht durchsetzen können – sie sind daher kaum verfügbar.

Im Gegensatz dazu existieren auf dem Markt sehr viele konventionelle Texteditoren, die zumeist frei erhältlich sind. Aufgrund der Tatsache, dass ein konventioneller Texteditor sich nicht an einer bestimmten Syntax orientieren muss, ist er wesentlich flexibler und universeller einsetzbar. Allerdings müssen zur Synchronisation die Änderungen im Texteditor in den abstrakten Syntaxbaum übertragen werden. Hierzu kann, wie bereits beschrieben, ein Parser eingesetzt werden. Um die Zuordnungen zum Modell nicht zu verlieren und nur die tatsächlich von den Codeänderungen betroffenen Elemente des abstrakten Syntaxbaums zu verändern, darf der bestehende abstrakte Syntaxbaum jedoch nicht einfach durch einen neuen abstrakten Syntaxbaum ersetzt werden. Stattdessen muss der bestehende abstrakte Syntaxbaum aktualisiert werden, das heißt, ein inkrementeller Parser wäre hier von Vorteil.

Heutige Entwicklungsumgebungen kompilieren den Code bereits inkrementell. Einen inkrementellen Parser besitzen sie meistens jedoch nicht. Bei der inkrementellen Kompilierung werden, nachdem der Code in einer Datei verändert wurde, die Abhängigkeiten zu anderen Dateien untersucht. Anschließend werden die veränderte Datei und die mit ihr in Beziehung stehenden Dateien neu kompiliert. Alle anderen Dateien bleiben unberührt. Bei dieser Art der inkrementellen Kompilierung wird ein inkrementeller Parser gar nicht benötigt. Dies mag auch der Grund sein, warum erste Arbeiten zum inkrementellen Parsen erst in den 80er Jahren aufgenommen und veröffentlicht wurden [GM80]. Seitdem wurden verschiedene Lösungen zum inkrementellen Parsen vorgeschlagen, wie zum Beispiel in [WG98]. Praxistaugliche Parsergeneratoren, die inkrementell arbeitende Parser erzeugen, gibt es zurzeit dennoch nicht.

Für batch-artig arbeitende Parser existieren hingegen sehr viele Parsergeneratoren, die zudem bereits mit fertigen Grammatiken für einige weit verbreitete Programmiersprachen ausgeliefert werden. Um die damit erzeugten Parser bei der Synchronisation von Modell und Code nutzen zu können, müssen wir dafür sorgen, dass der bestehende abstrakte Syntaxbaum beim wiederholten Parsen nicht immer wieder neu erzeugt wird, sondern die Änderungen in den bereits bestehenden abstrakten Syntaxbaum inkrementell eingepflegt werden. Hierzu wird der Code zunächst mit ei-

nem batch-artig arbeitenden Parser geparst (Schritt 4). Anschließend wird der dabei neu erzeugte abstrakte Syntaxbaum mit dem bereits existierenden abstrakten Syntaxbaum verglichen. Die ermittelten Unterschiede zwischen den beiden Syntaxbäumen werden dazu verwendet, um den bereits existierenden abstrakten Syntaxbaum zu aktualisieren und dadurch an den neuen abstrakten Syntaxbaum anzugleichen (Schritt 5). Nach der Aktualisierung des abstrakten Syntaxbaums erfolgt eine Synchronisation mit dem Modell (Schritt 6), so dass Code und Modell wieder miteinander abgeglichen sind. Nach diesem Prinzip kann eine Synchronisation in beide Richtungen erfolgen.

Kombinierte Spezifikation

Falls eine bidirektionale Synchronisation zwischen einem Modell und dem daraus generierten Code nicht benötigt wird, kann der Aufwand zur Spezifikation der Modell-zu-Text Beziehungen reduziert werden, indem die TGGs mit Textschablonen kombiniert werden. Durch den Einsatz der Textschablonen muss kein feingranulares Metamodell zur Repräsentation der abstrakten Syntax der zugrundeliegenden textuellen Sprache erstellt werden. Stattdessen wird ein Metamodell verwendet, das in den meisten Fällen nicht ganz so umfangreich ausfällt wie das Metamodell der textuellen Sprache. Dadurch sinkt sowohl die Anzahl der zu spezifizierenden TGG-Regeln als auch die Anzahl der in einer TGG-Regel zu spezifizierenden Objekte, was insgesamt den Aufwand zur Realisierung einer Codegenerierung signifikant reduziert.

Beispielautomat Die kombinierte Spezifikation aus TGGs und Textschablonen betrachten wir an einem Beispiel. Bei dem Beispiel handelt es sich um die Codegenerierung für Speicherprogrammierbare Steuerungen (SPSen), die bereits in Abschnitt 2.1 erwähnt wurde. Hierbei wird die Steuerung der Hardwarekomponenten eines Fertigungssystems mit mehreren *I/O-Automaten* [LT89] modelliert. Aus den I/O-Automaten wird anschließend SPS-Code generiert. Die Implementierung der I/O-Automaten erfolgt in der Sprache *Strukturierter Text* (ST) [IEC03], wobei wir in unserem Beispiel davon ausgehen, dass die generierte Implementierung der I/O-Automaten vollständig und damit eine Nachbearbeitung des ST-Codes nicht notwendig ist. Somit ist eine Synchronisation in nur eine Richtung ausreichend.

Abbildung 4.6 zeigt einen einfachen I/O-Automaten, der zur Beschreibung der Steuerung eines *Startaktors* verwendet wird. In dem Materialflusssystem des ISILEIT-Projekts (vergleiche Abschnitt 2.1) sorgt ein solcher Startaktor in jeder der vier Station dafür, dass Shuttles wieder gestartet werden können.

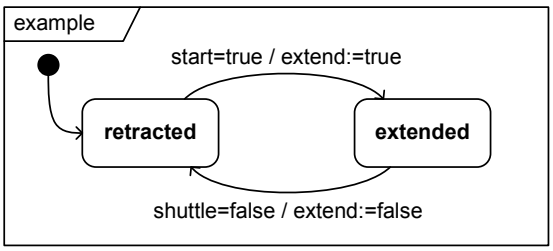


Abbildung 4.6: Beispielautomat in konkreter Syntax

Hierzu besitzt jeder Shuttle zwei Näherungssensoren. Der erste Näherungssensor tastet die Schiene während der Fahrt ab und reagiert auf an den Schienen angebrachte *Steuernocken*. Wird ein entsprechender Steuernocken vom Näherungssensor detektiert, so unterbricht der Shuttle die Stromversorgung zu seinem Niederspannungsmotor und der Shuttle hält an. Der zweite Näherungssensor reagiert hingegen auf einen an der Schiene montierten Startaktor. Dieser Startaktor fährt zum Starten eines Shuttles mit Hilfe eines pneumatischen Zylinders einen Metallnocken aus. Wird dieser Metallnocken vom zweiten Näherungssensor detektiert, so wird die Stromversorgung zum Niederspannungsmotor wieder hergestellt und der Shuttle gestartet. Damit nachfolgende Shuttles nach dem Anhalten nicht sofort wieder gestartet werden, muss der Metallnocken des Startaktors wieder eingefahren werden, sobald ein gestarteter Shuttle die Station verlassen hat. Der I/O-Automat aus Abbildung 4.6 beschreibt die Steuerung dieses Metallnockens.

In unserem Beispiel besteht der I/O-Automat aus den beiden Zuständen `retracted` und `extended`, die über zwei Transitionen miteinander verbunden sind. Der I/O-Automat befindet sich initial im Zustand `retracted`. In diesem Zustand ist der Metallnocken des Startaktors eingefahren, so dass vorbeifahrende Shuttles durch den ebenfalls an der Schiene angebrachten Steuernocken zunächst angehalten werden. Soll ein Shuttle wieder gestartet werden, so sendet die Umgebung das Signal `start`. Liegt das Signal vor (`start=true`), so wechselt der I/O-Automat vom Zustand `retracted` in den Zustand `extended`. Während dieses Zustandsübergangs signalisiert der I/O-Automat dem Startaktor, dass der Metallnocken ausgefahren werden soll (`extend:=true`). Sobald der Shuttle die Station verlassen hat, wird der I/O-Automat darüber benachrichtigt (`shuttle=false`). Daraufhin wechselt der I/O-Automat zurück in den Zustand `retracted`. Während dieses Zustandsübergangs wird der Metallnocken eingefahren (`extend:=false`), so dass vorbeifahrende Shuttles angehalten werden.

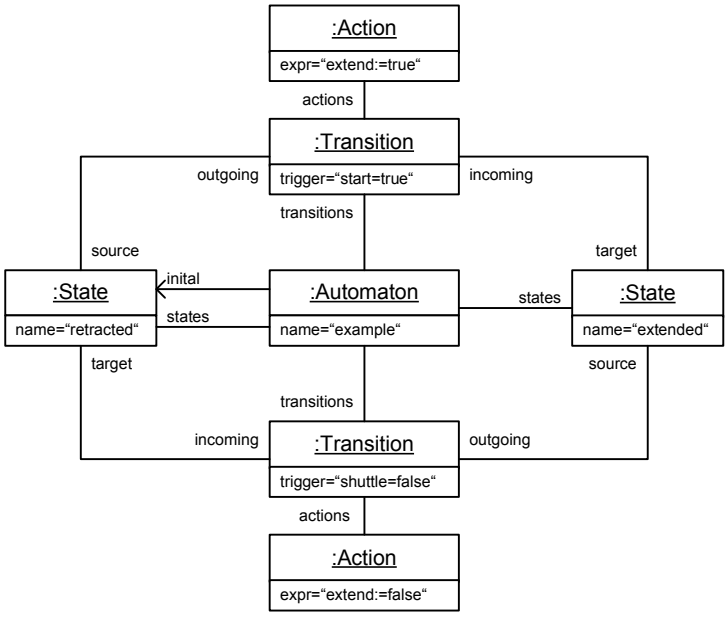


Abbildung 4.7: Beispielautomat in abstrakter Syntax (Objektdiagramm)

Der I/O-Automat in Abbildung 4.6 ist in seiner konkreten Syntax dargestellt. Derselbe I/O-Automat ist in Abbildung 4.7 in seiner abstrakten Syntax zu sehen. Diese beruht auf dem Metamodell für I/O-Automaten, das im Anhang A abgebildet ist (siehe Abbildung A.12, Seite 284). Zur Spezifikation der Modell-zu-Text Beziehungen mit TGGs und Textschablonen werden des Weiteren ein Metamodell zur Repräsentation der Textschablonen sowie ein Metamodell für das Korrespondenzmodell benötigt. Auch diese Metamodelle befinden sich im Anhang A (siehe Abbildungen A.13 und A.14, Seiten 284 ff.). Auf Grundlage dieser Metamodelle werden die entsprechenden TGG-Regeln zur Codegenerierung spezifiziert.

Beispielregel In Abbildung 4.8 ist eine Beispielregel dargestellt. Grundsätzlich unterscheidet sich die Beispielregel von den bisherigen TGG-Regeln nur durch die zusätzlich vorhandenen Textschablonen. In der TGG-Regel werden die Objekte der beteiligten Sprachen weiterhin durch Korrespondenzobjekte zueinander in Beziehung gesetzt. Allerdings haben wir bei der Spezifikation auf Attribute in den Korrespondenzobjekten verzichtet und die Attributbedingungen zwischen den Objekten direkt spezifiziert.

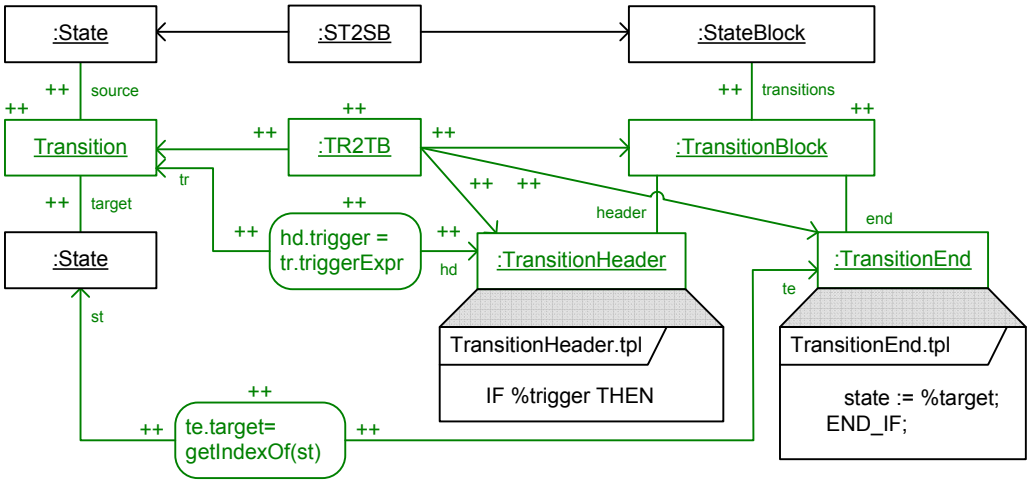


Abbildung 4.8: Beispiel für die Kombination einer TGG-Regel mit einer Textschablone

Die Beispielregel beschreibt, welche ST-Codefragmente zu einer Transition des I/O-Automaten generiert werden sollen. Hierzu wird das Objekt vom Typ `Transition` über das Korrespondenzobjekt vom Typ `TR2TB` zu den drei Objekten vom Typ `TransitionBlock`, `TransitionHeader` und `TransitionEnd` in Beziehung gesetzt. Den beiden Objekten vom Typ `TransitionHeader` und `TransitionEnd` ist jeweils eine Textschablone zugeordnet. Die Textschablonen enthalten den zu generierenden ST-Code sowie Platzhalter zur Parametrisierung der Textschablonen. Die Attributbedingungen beschreiben den Zusammenhang zwischen den Attributwerten der beiden Sprachen und werden letztendlich dazu verwendet, die Platzhalter in den Textschablonen durch konkrete Werte aus dem Modell des I/O-Automaten zu ersetzen.

In der hier gezeigten Beispielregel ist der ST-Code in der Textschablone `TransitionHeader.tpl` für die Überprüfung der Triggerbedingung zuständig. Die Textschablone `TransitionEnd.tpl` enthält den ST-Code, um nach einem Zustandsübergang den aktuellen Zustand des I/O-Automaten, der in der Variablen `state` gespeichert wird, zu aktualisieren. Zusätzlich wird in der Textschablone die bedingte Anweisung aus der vorherigen Textschablone abgeschlossen. Um eine vollständige Implementierung des hier vorgestellten I/O-Automaten generieren zu können, werden weitere TGG-Regeln benötigt. Die zusätzlichen TGG-Regeln befinden sich im Anhang A (siehe Seite 275 ff.).

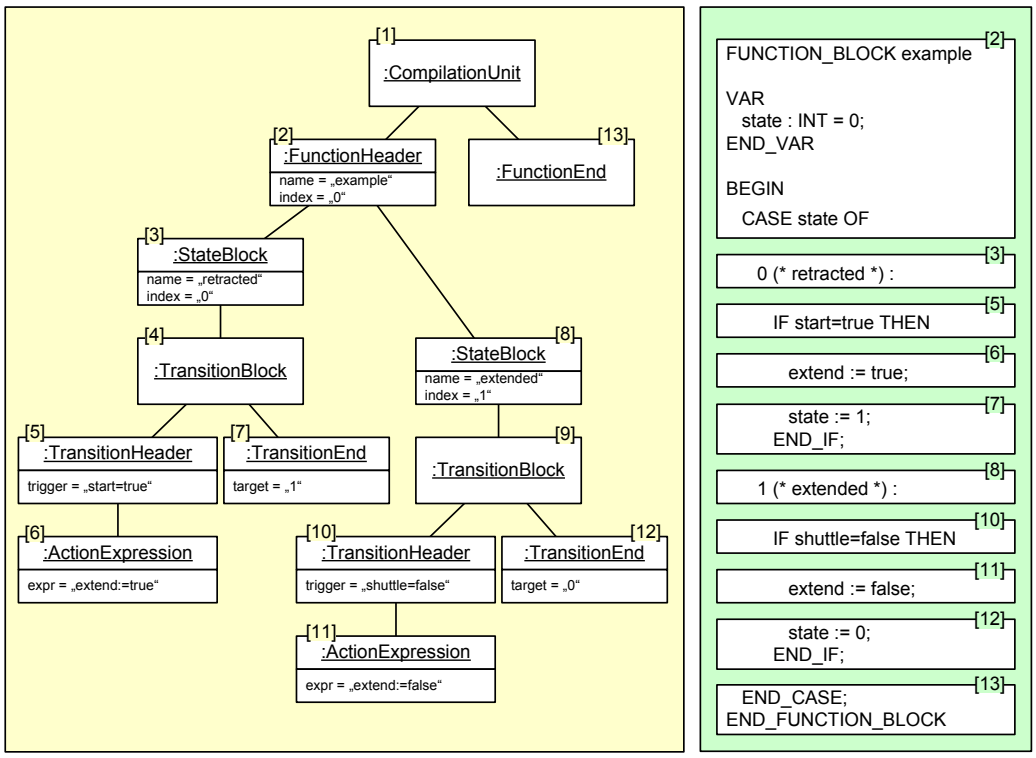


Abbildung 4.9: Ergebnis der Übersetzung in Strukturierten Text

Ausführung Auf der Grundlage dieser TGG-Regeln und der dazugehörigen Textschablonen wird die Codegenerierung in zwei Schritten durchgeführt. Zunächst wird der I/O-Automat aus Abbildung 4.7 in die Repräsentation für Textschablonen – wie bereits in Abschnitt 3.3.1 beschrieben – übersetzt. Dabei werden auch die Attributbedingungen berücksichtigt. Der dabei entstandene abstrakte Syntaxbaum ist in der linken Hälfte der Abbildung 4.9 dargestellt (gelb unterlegter Bereich). Anschließend wird der Syntaxbaum in preorder traversiert. Dabei wird die dem besuchten Objekt zugeordnete Textschablone instanziiert und die darin enthaltenen Platzhalter durch die Attributwerte aus dem besuchten Objekt ersetzt. Der durch die Instanziierung der Textschablonen entstandene ST-Code wird in dieser Reihenfolge aneinandergehängt und ergibt die Implementierung des I/O-Automaten. Das Ergebnis dieses Schrittes ist in der rechten Hälfte der Abbildung 4.9 zu sehen (grün unterlegter Bereich).

Die Zahlen an den Objekten sowie den instanziierten Textschablonen sind in der Abbildung 4.9 lediglich aus Präsentationsgründen aufgeführt. Sie ge-

ben einerseits die Traversierungsreihenfolge im abstrakten Syntaxbaum wieder. Andererseits setzen sie die Objekte des abstrakten Syntaxbaums zu den instanziierten Textschablonen in Beziehung. Für diejenigen Objekte, denen keine Textschablone in der dazugehörigen TGG-Regel zugeordnet wurde, wird auch keine Textschablone instanziiert (vergleiche Wurzelobjekt vom Typ `CompilationUnit` und TGG-Regel in Abbildung A.15, Seite 286). Dies erklärt auch die lückenhafte Nummerierung der instanziierten Textschablonen in der rechten Hälfte der Abbildung 4.9.

4.1.3 Gegenüberstellung

Die Codegenerierung auf der Grundlage einer Spezifikation von Modell-zu-Text Beziehungen mit TGGs und Textschablonen bietet gegenüber anderen Ansätzen zur Codegenerierung (vergleiche Abschnitt 4.1.1) sowohl einige Vorteile als auch einen Nachteil.

Gegenüber der Codegenerierung durch direkte Programmierung und der Codegenerierung mit Textschablonen hat der Einsatz von TGGs den Vorteil, dass die Zuordnung zwischen den Modellartefakten und dem aus ihnen generierten Code formal spezifiziert ist. Diese Formalisierung kann beispielsweise herangezogen werden, um die Codegenerierung formal zu verifizieren (vergleiche Kapitel 6). Zudem ist die Zuordnung zwischen Modellelementen und den dazugehörigen Codefragmenten leichter nachvollziehbar. Gleichzeitig wird diese Zuordnung bei der Codegenerierung im Korrespondenzmodell gespeichert, so dass die Nachverfolgbarkeit auch nach der Codegenerierung sichergestellt ist. Dieser Mechanismus muss nicht erst aufwändig programmiert oder zusätzlich spezifiziert werden – er ist bereits im Formalismus der TGGs verankert und kann damit automatisch bereitgestellt werden.

Ein weiterer Vorteil wird deutlich, wenn man eine Codegenerierung mit Textschablonen betrachtet, die ohne TGGs spezifiziert wurde (vergleiche Abschnitt 4.1.1). Diese Textschablonen enthalten neben dem zu generierenden Code zusätzliche Anweisungen, die der Steuerung der Codegenerierung und der Abfrage von Modelleigenschaften dienen. Diese Anweisungen sind mit dem zu generierenden Code vermischt und machen diese Textschablonen schwer lesbar. Im Gegensatz dazu enthalten die Textschablonen im kombinierten Ansatz keine Anweisungen zur Steuerung der Codegenerierung. Die Codegenerierung wird im kombinierten Ansatz implizit durch die TGG-Regeln gesteuert. Die Textschablonen enthalten nur noch das Codegerüst mit dazugehörigen Platzhaltern. Dadurch sind die Textschablonen aus dem kombinierten Ansatz leichter lesbar.

Ein allgemeiner und nicht zu unterschätzender Vorteil beim Einsatz von TGGs ist, dass sowohl die Spezifikation von Modell-zu-Modell als auch die Spezifikation von Modell-zu-Text Beziehungen auf der Grundlage eines einzigen Formalismus erfolgt. Auch wenn die Spezifikation der Modell-zu-Text Beziehungen im kombinierten Ansatz um Textschablonen erweitert ist, so ist die Notation zum größten Teil bereits bekannt und muss nicht grundlegend neu erlernt werden.

Gegenüber der Spezifikation mit TGGs ohne Textschablonen hat der kombinierte Ansatz aus TGGs und Textschablonen den Vorteil, dass nur ein Metamodell zur Repräsentation der Textschablonen spezifiziert werden muss. Dieses Metamodell ist nicht so umfangreich wie ein vollständiges Metamodell einer Programmiersprache. Damit sinkt auch der Spezifikationsaufwand. Zusätzlich wird der zu generierende Code in den Textschablonen – bis auf die zur Parametrisierung benötigten Platzhalter – in seiner konkreten Syntax angegeben. Dies ist deutlich einfacher, als den zu generierenden Code in seiner abstrakten Darstellung zu spezifizieren.

Ein Nachteil des kombinierten Ansatzes gegenüber der Spezifikation mit TGGs ohne Textschablonen besteht jedoch darin, dass die Synchronisation nur in eine Richtung ausgeführt werden kann. Damit können Änderungen im Code nicht an das Modell weitergegeben werden, um das Modell an den geänderten Code anzupassen. Dieser Nachteil ist jedoch bei der direkten Programmierung und bei der Codegenerierung mit Textschablonen ebenfalls vorhanden (vergleiche Abschnitt 4.1.1). Für eine Synchronisation in beide Richtungen ist keiner dieser Ansätze geeignet. Falls eine bidirektionale Synchronisation benötigt wird, müssen die TGG-Regeln ohne Textschablonen auf der Grundlage eines feingranularen Metamodells der Programmiersprache – wie in Abschnitt 4.1.2 beschrieben – spezifiziert werden. Um den damit verbundenen Aufwand zu reduzieren, kann der nachfolgend beschriebene Ansatz zur Spezifikation durch Beispielzuordnungen eingesetzt werden.

4.2 Spezifikation durch Beispielzuordnungen

Bisher wurden Korrespondenzbeziehungen zwischen zwei Modellen in der abstrakten Syntax spezifiziert, die durch die Metamodelle der beteiligten Modellierungssprachen definiert werden. Metamodelle sind allerdings nicht immer so einfach wie in unserem Beispiel der vorangegangenen Kapitel. In den meisten Fällen sind Metamodelle recht groß und enthalten viele Konzepte, die nur schwer zu ihrer konkreten Repräsentation in der Modellie-

rungssprache zugeordnet werden können. Ein gutes Beispiel hierfür stellt die *Unified Modelling Language*(UML) [UML05] dar, in der die Metaklasse *Property* sowohl zur Repräsentation von Attributen als auch zur Repräsentation von Assoziationsenden herangezogen wird. Die Semantik und die graphische Darstellung hängen somit vom Kontext ab, in dem eine Instanz dieser Metaklasse verwendet wird. Insbesondere in solchen Fällen kann eine auf Metamodellen basierende Spezifikation der Korrespondenzbeziehungen kompliziert werden.

Um die Spezifikation der Korrespondenzbeziehungen zu vereinfachen, stellen wir in diesem Abschnitt einen Ansatz vor, in dem die Korrespondenzbeziehungen durch Zuordnungen von Beispielen vorgenommen werden. Die Beispiele werden dabei in der konkreten Syntax der Modelle angegeben. Handelt es sich dabei um eine visuelle Modellierungssprache, so entspricht die konkrete Syntax der graphischen Darstellung dieser Modellierungssprache. Bei einer textuellen Sprache wird die konkrete Syntax hingegen durch Text repräsentiert. In beiden Fällen ist die konkrete Syntax meistens geläufiger als die zugrundeliegende abstrakte Syntax der Sprachen. Dadurch ist es wesentlich einfacher, die Korrespondenzbeziehungen mit Beispielen in der konkreten Syntax der Sprachen zu spezifizieren. Aus diesen Beispielen können anschließend TGG-Regeln automatisch synthetisiert werden. Die Spezifikation kann somit deutlich komfortabler durchgeführt werden.

Im folgenden Abschnitt präsentieren wir die grundlegende Idee und das Lösungsprinzip, welches im Rahmen der Diplomarbeit von Alexander Geburzi umgesetzt wurde [Geb06]. Anschließend stellen wir informell den Algorithmus zur Regelsynthese an einem Beispiel vor. Danach gehen wir auf notwendige Erweiterungen des Regelsynthesealgorithmus ein, damit die im vorangegangenen Kapitel eingeführten Konzepte für Attribute, Bedingungen und wiederverwendbare Objekte behandelt werden können. Wir schließen diesen Abschnitt mit einigen Betrachtungen und Empfehlungen für den Einsatz dieser Methode in der Praxis.

4.2.1 Idee und Lösungsprinzip

Die zugrundeliegende Idee unseres Ansatzes besteht darin, dass der Benutzer die Korrespondenzbeziehungen zwischen zwei Modellen spezifiziert, indem er eine Menge von Beispielzuordnungen angibt. Eine Beispielzuordnung besteht aus zwei zueinander korrespondieren Modellen. Über die Zuordnung der Modelle wird die semantische Beziehung beziehungsweise Korrespondenz zwischen diesen beiden Modellen ausgedrückt.

Abbildung 4.10 zeigt eine Beispielzuordnung, die eine Korrespondenz zwischen einem Blockdiagramm und einem Klassendiagramm definiert. Die Beispielzuordnung wird in der konkreten Syntax der beteiligten Modellierungssprachen angegeben. In diesem Sinne haben wir bereits einige Beispielzuordnungen auch in Abbildung 2.4 gezeigt.

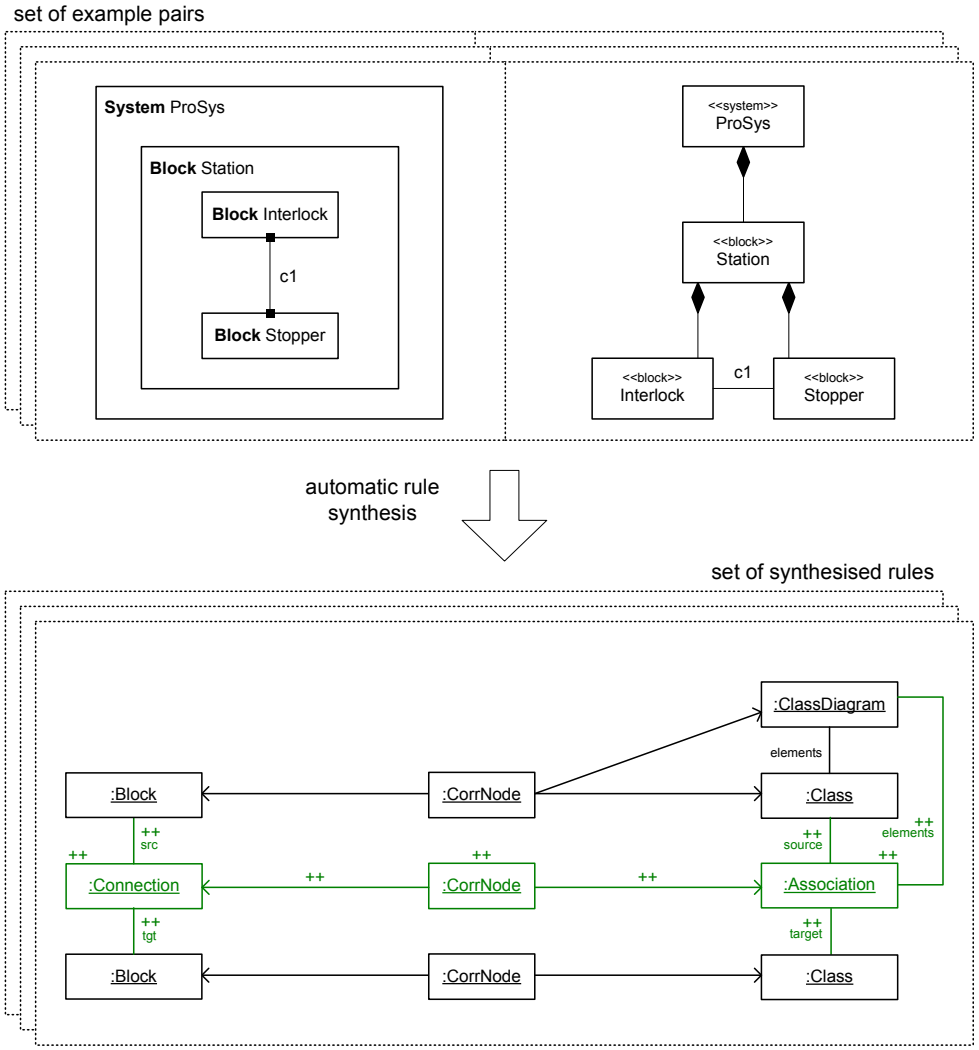


Abbildung 4.10: Überblick zur Spezifikation mit Beispielzuordnungen

Um die so spezifizierten Beispielzuordnungen für die noch später vorzustellende Modellsynchronisation nutzen zu können, werden daraus automatisch TGG-Regeln synthetisiert. Dies wird erreicht, indem die Beispielzuordnun-

gen analysiert und die Unterschiede zwischen den gegebenen Beispielzuordnungen identifiziert werden. Diese Unterschiede werden anschließend genutzt, um daraus entsprechende TGG-Regeln zu konstruieren. Für diese Art der Regelsynthese werden daher mindestens zwei Beispielzuordnungen benötigt, die sowohl einige Gemeinsamkeiten als auch Unterschiede in den darin enthaltenen Konzepten aufweisen. Auch wenn bei diesem Ansatz einige Einschränkungen und Anforderungen beachtet werden müssen, so bietet dieser Ansatz aus Sicht des Benutzers den Vorteil, dass detaillierte Kenntnisse über den Aufbau der abstrakten Syntax der Modelle nicht nötig sind.

Für die automatische Regelsynthese müssen die Beispielzuordnungen in einen gemeinsamen TGG-Formalismus übersetzt werden. In Abbildung 4.11 ist eine solche Übersetzung zu sehen. Die Beispielzuordnung setzt ein leeres System zu einer Klasse in Beziehung, die mit dem Stereotyp ≪system≫ gekennzeichnet ist. Die Übersetzung dieser Beispielzuordnung beruht auf der Tatsache, dass jedes Modell in konkreter Syntax auch eine abstrakte Syntax besitzt, die auf einem dazugehörigen Metamodell basiert. In einem Modellierungswerkzeug wird durch die Editieroperationen des Benutzers im Hintergrund die Repräsentation des Modells in abstrakte Syntax aufgebaut. Tatsächlich ist es sogar so, dass in einem Werkzeug die graphische Repräsentation eines Modells auf Grundlage seiner abstrakten Syntax erzeugt wird.

Diese Tatsache kann ausgenutzt werden, um beide Modelle in den gemeinsamen TGG-Formalismus zu übersetzen. In Abbildung 4.11 werden zum Beispiel die Objekte `ClassDiagram`, `Class` und `Stereotype` in Objekte des Typs `TGGObject` übersetzt. Der Typ eines Objekts wird als Attributwert des Objekts `TGGObject` hinterlegt. Die Links zwischen den Objekten werden in Objekte des Typs `TGGLink` übersetzt. Auch hier wird der Typ der zugrundeliegenden Assoziation als Attributwert dieser Objekte gespeichert. In der vorliegenden Implementierung erfolgt die Übersetzung mit einer fest definierten Modelltransformation, die ebenfalls auf TGGs basiert. Allerdings hat sich gezeigt, dass die Übersetzungsregeln sehr allgemein sind, so dass alternativ eine generische Übersetzung ebenfalls möglich ist [Geb06].

Vergleicht man in Abbildung 4.11 die graphische Repräsentation einer TGG-Regel mit der graphischen Repräsentation der abstrakten Syntax eines Modells, so stellt man fest, dass beide Repräsentationen sehr ähnlich zueinander sind. Zwischen beiden Repräsentationen existiert jedoch ein fundamentaler Unterschied. Beispielsweise kann im TGG-Formalismus ein `TGGObject` der linken und der rechten Regelseite oder nur der rechten Regelseite zugewiesen werden. Wird das Objekt nur der rechten Seite zugewiesen, wird es mit ++ annotiert. Außerdem können im TGG-Formalismus die Ob-

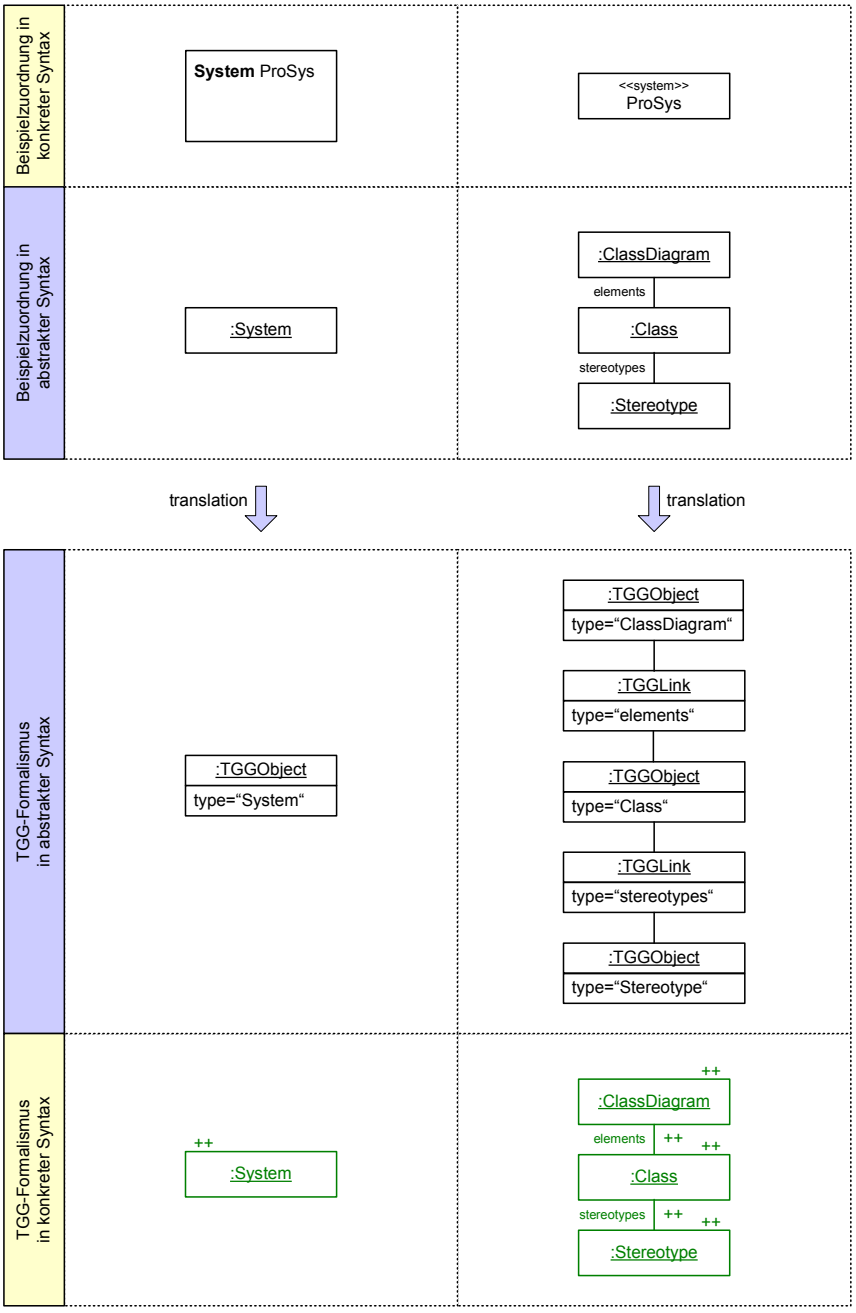


Abbildung 4.11: Beispielzuordnung 1 (inklusive der Übersetzung in den TGG-Formalismus)

jekte verschiedenen Domänen zugeordnet werden. Nach der Übersetzung der Beispielzuordnungen wird die Regelsynthese auf der abstrakten Syntax des TGG-Formalismus durchgeführt.

Nachdem wir vorgestellt haben, wie die Beispielzuordnungen in einen gemeinsamen TGG-Formalismus überführt werden, stellen wir den grundlegenden Algorithmus der Regelsynthese an einem Beispiel vor. Aus didaktischen Gründen werden wir hierzu aber nicht die abstrakte Syntax, sondern die einfachere und benutzerfreundlichere, konkrete Syntax des TGG-Formalismus verwenden.

4.2.2 Regelsynthese

Um den grundlegenden Algorithmus der Regelsynthese zu erklären, starten wir mit einer leeren Menge von Beispielzuordnungen und erweitern diese Menge Schritt für Schritt. Auch wenn wir hier aus Präsentationsgründen so vorgehen, ist – wie wir später noch sehen werden – die Regelsynthese auch in der Lage, eine gegebene Menge von Beispielzuordnungen auf einmal zu verarbeiten. Der hier vorgestellte iterative Prozess ist keine Voraussetzung für die Regelsynthese – auch wenn er dem Benutzer erlaubt, inkrementell vorzugehen und für ihn dadurch sehr komfortabel ist. Wir veranschaulichen den Algorithmus zur Regelsynthese mithilfe des bereits bekannten Beispiels und beginnen mit einer sehr einfachen Beispielzuordnung.

Beispielzuordnung 1

Die erste Beispielzuordnung besteht aus einem leeren System und einer Klasse mit dem Stereotyp ≪system≫. Diese Zuordnung wurde bereits in Abbildung 4.11 gezeigt. In einem ersten Schritt wird diese Beispielzuordnung in den gemeinsamen TGG-Formalismus überführt. Entsprechend den Metamodellen wird ein System durch ein Objekt des Typs `System` repräsentiert. Eine Klasse wird hingegen durch ein Objekt des Typs `Class` dargestellt und durch ein Objekt des Typs `ClassDiagram` über den Link `elements` referenziert. Die Kennzeichnung einer Klasse mit einem Stereotyp erfolgt durch ein Objekt des Typs `Stereotype`, welches über den Link `stereotypes` an diese Klasse gehängt wird (vergleiche dazu die Abbildungen 3.1 und 3.2 auf den Seiten 57 und 58). Während der Übersetzung in den TGG-Formalismus werden die Objekte den zugehörigen Domänen zugeordnet. Zusätzlich werden alle Objekte und Links mit ++ gekennzeichnet, das heißt, alle Objekte und Links werden der rechten Regelseite einer TGG-Regel zugeordnet.

Nach der Übersetzung der Beispielzuordnung in den gemeinsamen TGG-Formalismus wird überprüft, ob bereits synthetisierte Regeln existieren, die auf diese Struktur angewendet werden können. Aufgrund der Tatsache, dass es sich hierbei um unsere erste Beispielzuordnung handelt, müssen keine weiteren TGG-Regeln betrachtet werden. Daher wird einfach nur ein Korrespondenzobjekt vom Typ `CorrNode` zwischen den extrahierten Objekten der beiden Domänen eingefügt. In Abbildung 4.12 ist das Ergebnis der Regelsynthese aus der ersten Beispielzuordnung zu sehen.

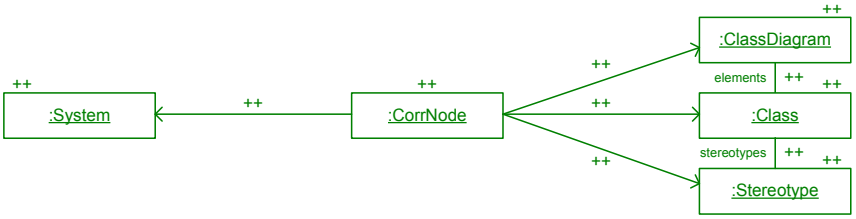


Abbildung 4.12: Synthese des Axioms

Im Augenblick befinden sich alle Objekte und Links nur auf der rechten Regelseite der synthetisierten TGG-Regel. Die linke Regelseite ist noch leer. Daher entspricht die synthetisierte Regel noch nicht der Struktur, die wir für TGG-Regeln kennen gelernt haben. Für die weitere Verarbeitung werden wir diese Regel so lassen wie sie ist. Am Ende der Regelsynthese werden allerdings die ++ Markierungen entfernt, das heißt, die Objekte und Links werden auch der linken Regelseite zugeordnet. Dadurch erhalten wir das bereits in Abbildung 3.9 vorgestellte Axiom. Dieser Schritt wird ausgeführt, da unsere Beispielzuordnung nur ein Korrespondenzobjekt enthält und damit zwangsläufig nur ein Axiom darstellen kann.

Beispielzuordnung 2

Wir setzen die Regelsynthese fort, indem wir eine zweite Beispielzuordnung erstellen. Die zweite Beispielzuordnung beschreibt, wie ein Block innerhalb eines Systems auf Elemente eines Klassendiagramms abgebildet wird. Die Zuordnung ist in der oberen Hälfte der Abbildung 4.13 zu sehen. Die Regelsynthese startet wiederum mit der Übersetzung der gegebenen Beispielzuordnung in den gemeinsamen TGG-Formalismus. Das Ergebnis der Übersetzung ist in der unteren Hälfte der Abbildung 4.13 dargestellt. Die so erhaltene Struktur ist Grundlage der nachfolgenden Schritte.

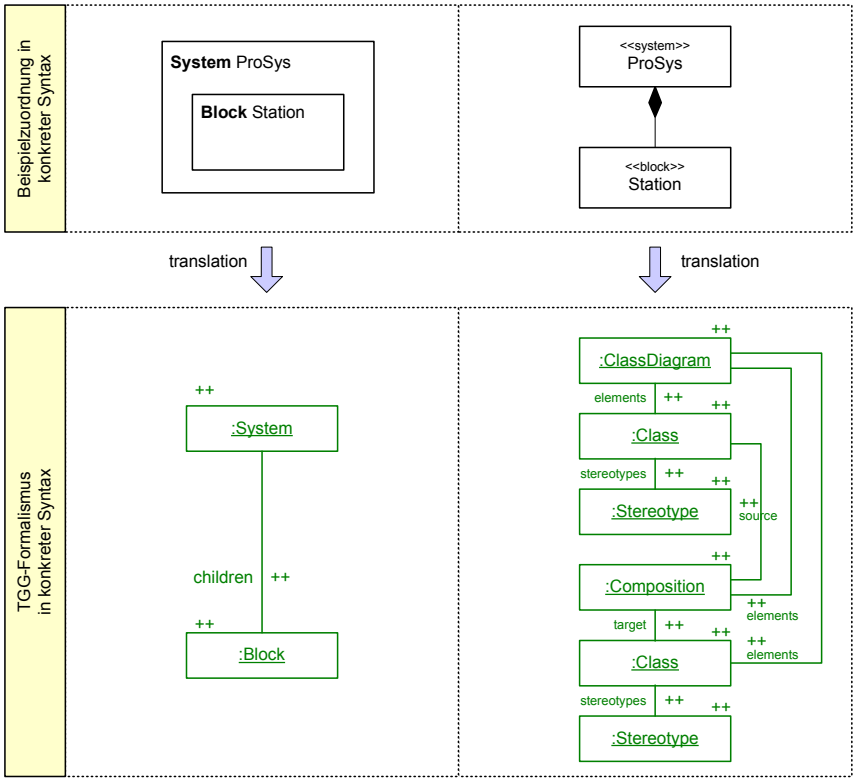


Abbildung 4.13: Beispielzuordnung 2

Der Synthesealgorithmus überprüft nun, ob eine Regel existiert, die auf die aus der Beispielzuordnung extrahierte Struktur angewendet werden kann. In unserem Fall existiert bisher nur die Regel aus Abbildung 4.12. Daher versucht die Regelsynthese die Objekte dieser Regel mit den Objekten der neuen Struktur zu matchen. Aufgrund der Tatsache, dass das Korrespondenzobjekt der Regel noch mit ++ gekennzeichnet ist, wird es bei diesem Matching nicht berücksichtig. Ein gültiges Matching ist in der oberen Hälfte der Abbildung 4.14 zu sehen.

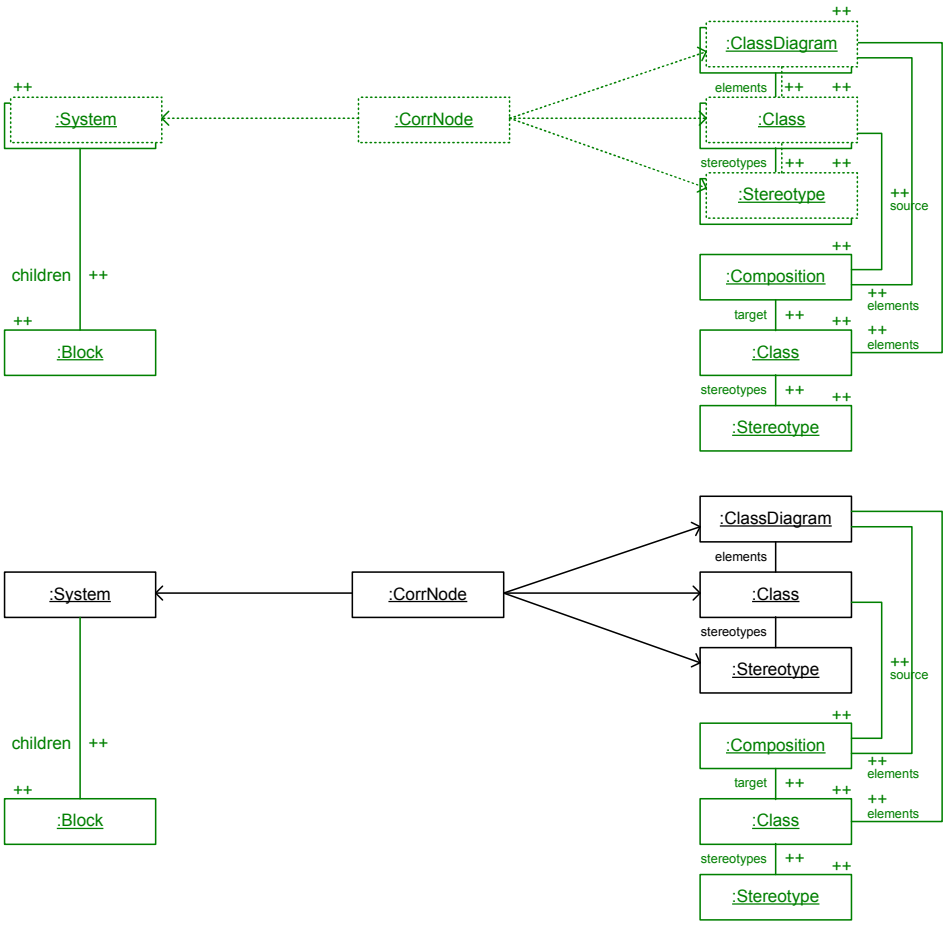


Abbildung 4.14: Regelsynthese aus Beispielzuordnung 2 – Schritte 1 und 2

Aufgrund dieses Matchings werden die übereinstimmenden Objekte der Beispielzuordnung auch der linken Regelseite der neuen TGG-Regel zugeordnet, das heißt, die Markierung mit ++ wird entfernt. Zusätzlich wird

das fehlende aber durch die vorher synthetisierte Regel geforderte Korrespondenzobjekt hinzugefügt. Da das hinzugefügte Korrespondenzobjekt nun Objekte verbindet, die sich sowohl auf der linken als auch auf der rechten Seite der TGG-Regel befinden, wird dieses Objekt auf beiden Seiten der Regel hinzugefügt. Das Ergebnis dieses Schrittes ist in der unteren Hälfte der Abbildung 4.14 zu sehen.

Bei der Synthese wird versucht, eine Regel möglichst häufig anzuwenden. In unserem Beispiel kann die bereits synthetisierte Regel aus Abbildung 4.12 aber nur einmal auf die extrahierte Struktur angewendet werden. Daher wird der Synthesealgorithmus fortgesetzt. Aufgrund der Tatsache, dass bisher allerdings nur eine einzige Regel existiert und diese bereits angewendet wurde, müssen keine weiteren Regeln überprüft werden.

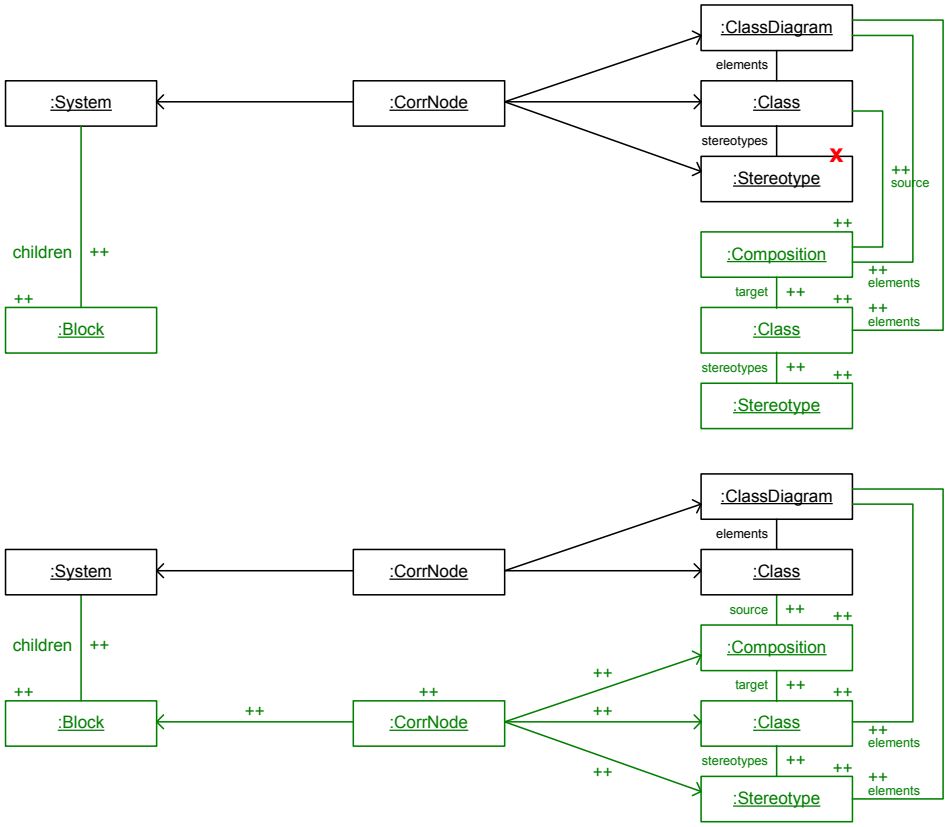


Abbildung 4.15: Regelsynthese aus Beispielzuordnung 2 – Schritte 3 und 4

Die bisher synthetisierte Regelstruktur enthält noch ein Objekt vom Typ `Stereotype`, welches in der Abbildung 4.15 mit einem `x` gekennzeichnet ist.

Die Anwesenheit dieses zusätzlichen Objekts wirkt sich für die Anwendbarkeit der TGG-Regel nicht weiter störend aus, da aufgrund des angewendeten Axioms sichergestellt ist, dass ein solches Objekt tatsächlich auch vorhanden ist. Damit ist das Objekt aber auch redundant, das heißt, es könnte genauso gut auch weggelassen werden. Bei der späteren Ausführung der TGG-Regeln hängt die Ausführungsgeschwindigkeit insbesondere von der Anzahl der in einer Regel enthaltenen Objekte und Links ab, da diese überprüft und gebunden werden müssen. Daher ist es sinnvoll, redundante Objekte zu eliminieren. Solche Objekte können daran erkannt werden, dass sie keine Links zu neu erzeugten, das heißt, mit ++ markierten, Objekten haben. Der Synthesealgorithmus löscht daher solche Objekte und die dazu inzidenten Links aus der synthetisierten TGG-Regel.

Im letzten Schritt wird nun ein neues Korrespondenzobjekt erzeugt und zu unserer Struktur hinzugefügt. Dieses Korrespondenzobjekt verbindet alle verbliebenen Objekte, die noch mit ++ markiert sind. Daher wird dieses Korrespondenzobjekt ebenfalls mit ++ markiert. Die daraus resultierende, finale TGG-Regel ist in der unteren Hälfte der Abbildung 4.15 dargestellt.

Beispielzuordnung 3

Bisher haben wir durch die Beispielzuordnung definiert, wie ein System und ein Block innerhalb dieses Systems auf Elemente eines Klassendiagramms abgebildet werden. Diese beiden Fälle werden durch die synthetisierten TGG-Regeln bereits abgedeckt. Allerdings fehlt noch eine Regel, die einen Block, der innerhalb eines anderen Blocks enthalten ist, zu Elementen eines Klassendiagramms zuordnet. Daher setzen wir die Regelsynthese fort, indem wir eine weitere Beispielzuordnung angeben, die genau diesen Fall abdeckt. In der oberen Hälfte der Abbildung 4.16 ist unsere dritte Beispielzuordnung dargestellt. Die Beispielzuordnung wird ebenfalls für die Regelsynthese vorbereitet, indem sie in den gemeinsamen TGG-Formalismus übersetzt wird. Die übersetzte Beispielzuordnung ist in der unteren Hälfte der Abbildung 4.16 zu sehen.

Der Regelsynthesealgorithmus überprüft nun wieder, ob bereits synthetisierte Regeln auf die neue Beispielzuordnung anwendbar sind. Aufgrund der Tatsache, dass die neue Beispielzuordnung nur aus Objekten und Links besteht, die mit ++ markiert sind, kann im Augenblick nur die Regel aus Abbildung 4.12 angewendet werden. Abbildung 4.17 illustriert die Anwendung dieser Regel durch die gestrichelt dargestellten Objekte und Links. Die ++ Markierung der gebundenen Objekte und Links wird entfernt und das

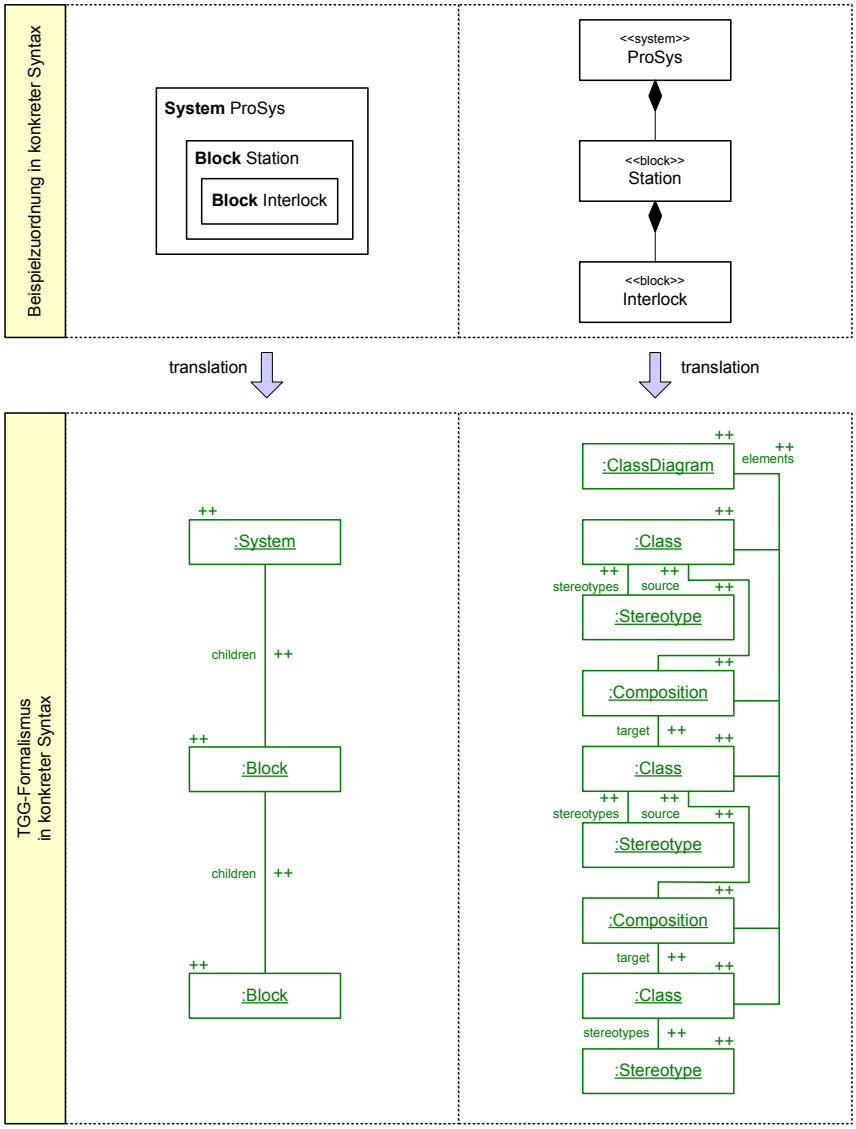


Abbildung 4.16: Beispielzuordnung 3

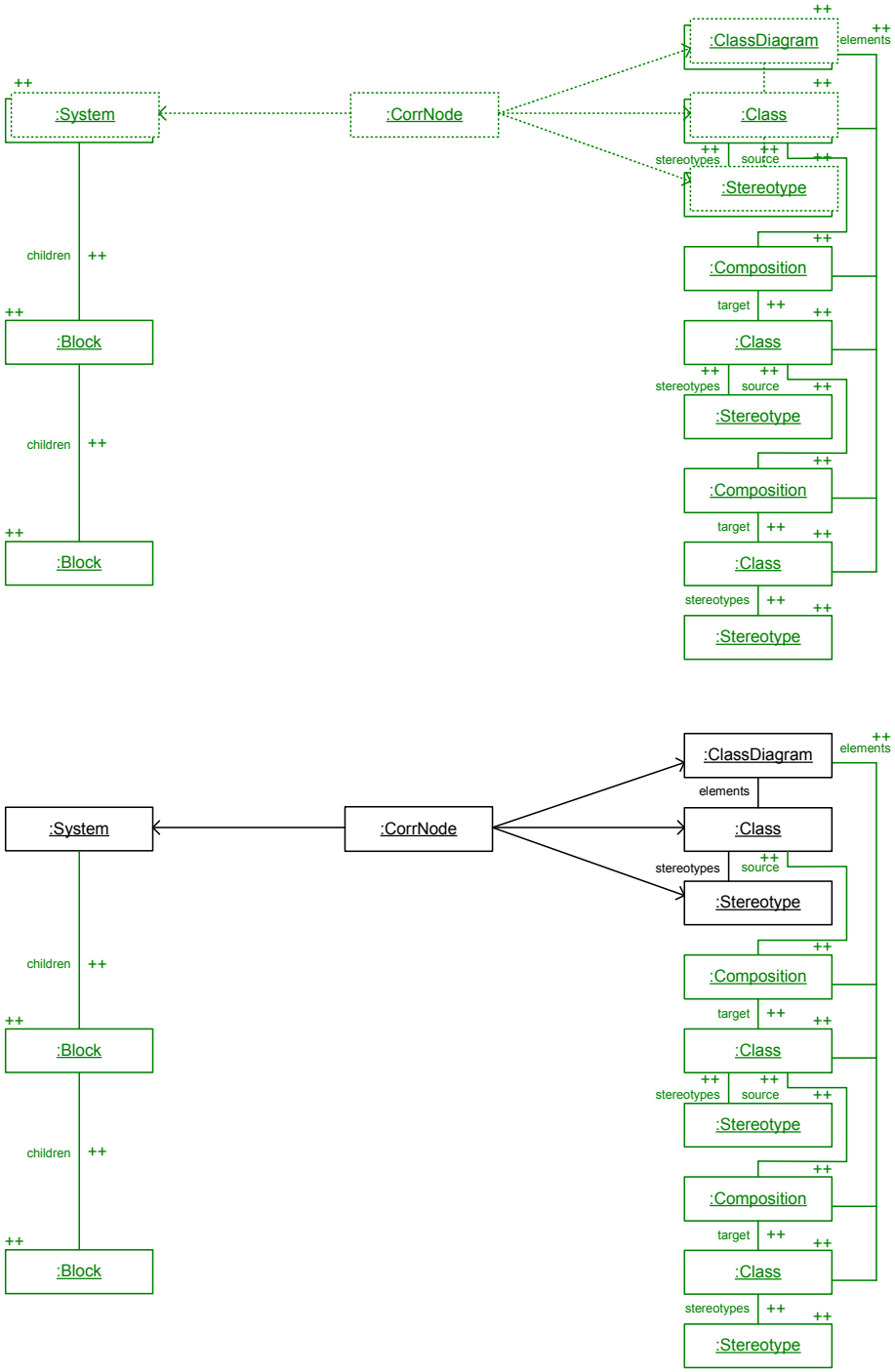


Abbildung 4.17: Regelsynthese aus Beispielzuordnung 3 – Schritte 1 und 2

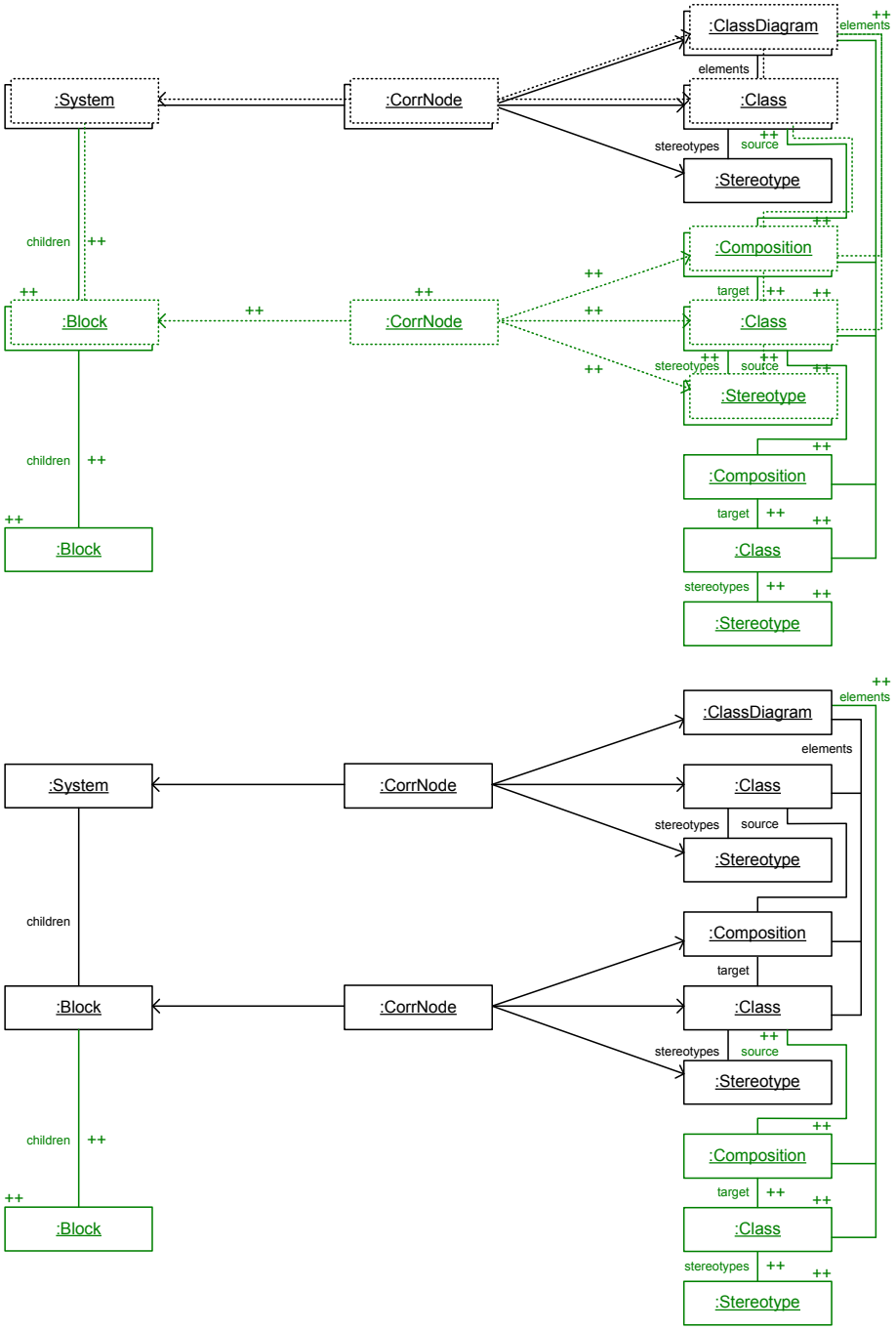


Abbildung 4.18: Regelsynthese aus Beispielzuordnung 3 – Schritte 3 und 4

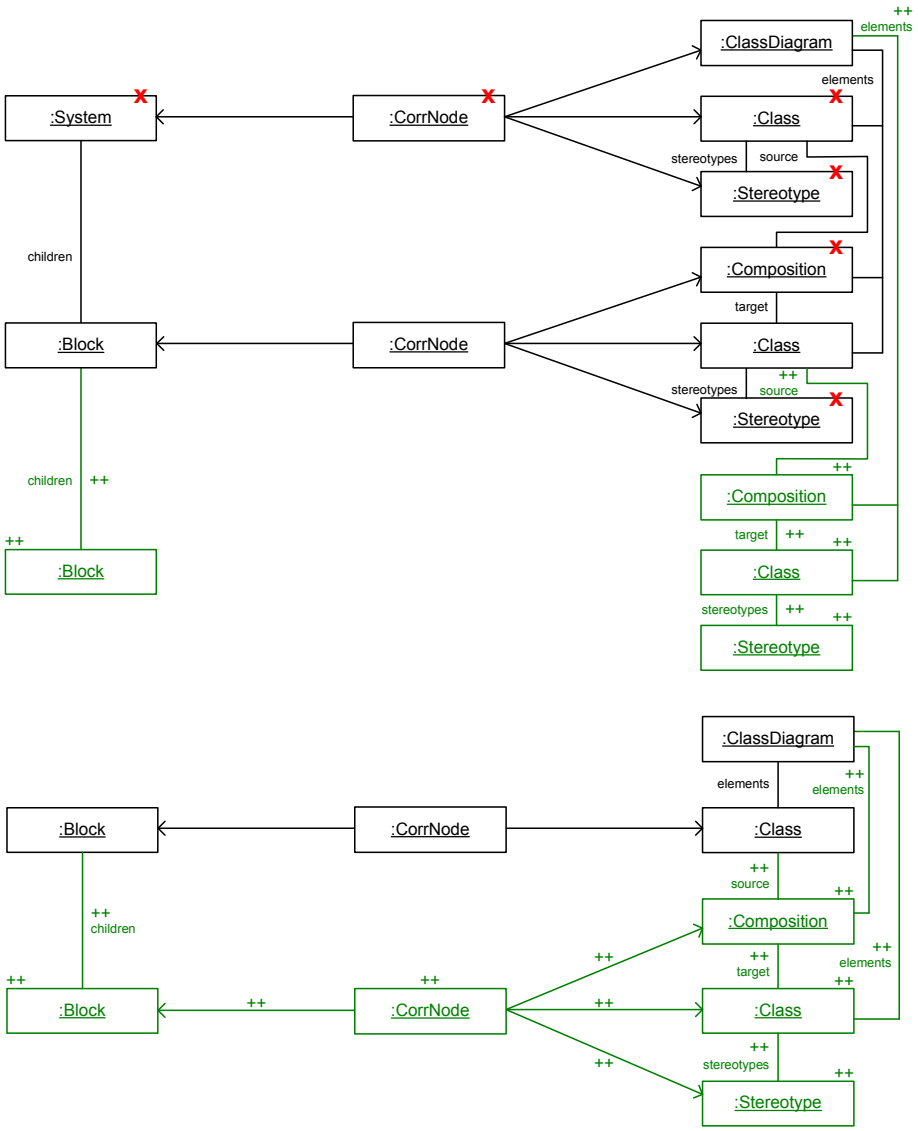


Abbildung 4.19: Regelsynthese aus Beispielzuordnung 3 – Schritte 5 und 6

fehlende Korrespondenzobjekt zwischen diesen Objekten eingefügt. Die aus dieser Regelanwendung entstandene Struktur ist in Abbildung 4.17 zu sehen.

Die vorliegende Struktur besitzt nun einige Objekte und Links mit und ohne ++ Markierungen. Daher kann nun auch die synthetisierte TGG-Regel aus Abbildung 4.15 angewendet werden. In Abbildung 4.18 ist die Regelanwendung und das daraus hervorgegangene neue Korrespondenzobjekt zu sehen. Die ++ Markierungen der gebundenen Objekte und Links – sofern sie vorhanden waren – wurden entfernt.

Aufgrund der durchgeführten Regelanwendung sind nur einige wenige mit ++ markierte Objekte und Links übrig geblieben. Da nun keine weiteren bereits synthetisierten Regeln auf diese Objekte angewendet werden können, werden die Objekte über ein neu hinzugefügtes Korrespondenzobjekt miteinander in Beziehung gesetzt. Auch hier wird wieder das neue Korrespondenzobjekt mit ++ markiert, da es nur Objekte mit einer ++ Markierung miteinander verbindet und dies der Struktur von TGG-Regeln entspricht.

Die synthetisierte TGG-Regel entspricht bereits der Struktur von TGG-Regeln. Allerdings sind hier – wie schon im vorherigen Fall – einige Objekte vorhanden, die nicht direkt mit den mit ++ markierten Objekten verbunden sind. Diese Objekte wurden in Abbildung 4.19 mit einem `x` gekennzeichnet.

Alle mit einem `x` markierten Objekte und die dazu inzidenten Links werden aus der synthetisierten TGG-Regel entfernt. Dies betrifft auch alle Korrespondenzobjekte, die keine oder nur Links zu einer einzigen Domäne besitzen. In unserem Beispiel betrifft dies das mit `x` gekennzeichnete Korrespondenzobjekt, da nach dem Löschen des Objektes vom Typ `System` dieses Korrespondenzobjekt nur noch Links zu Elementen des Klassendiagramms besitzt. Daher wird es ebenfalls gelöscht. Die endgültige TGG-Regel ist in Abbildung 4.19 dargestellt.

Beispielzuordnung 4

Bisher waren die Beispielzuordnungen relativ einfach gehalten. Sie haben einige wenige Elemente eines Blockdiagramms zu Elementen eines Klassendiagramms in Beziehung gesetzt. Im Folgenden schauen wir uns eine etwas komplexere Beispielzuordnung an, die in der oberen Hälfte der Abbildung 4.20 zu sehen ist. Diese Beispielzuordnung zeigt auf der linken Seite zwei Blöcke, die innerhalb eines Systems angeordnet und über einen Kanal miteinander verbunden sind. Die korrespondierenden Elemente im Klassendiagramm sind auf der rechten Seite der Abbildung 4.20 zu sehen.

Wie in den vorangegangenen Beispielzuordnungen wird auch diese Bei-

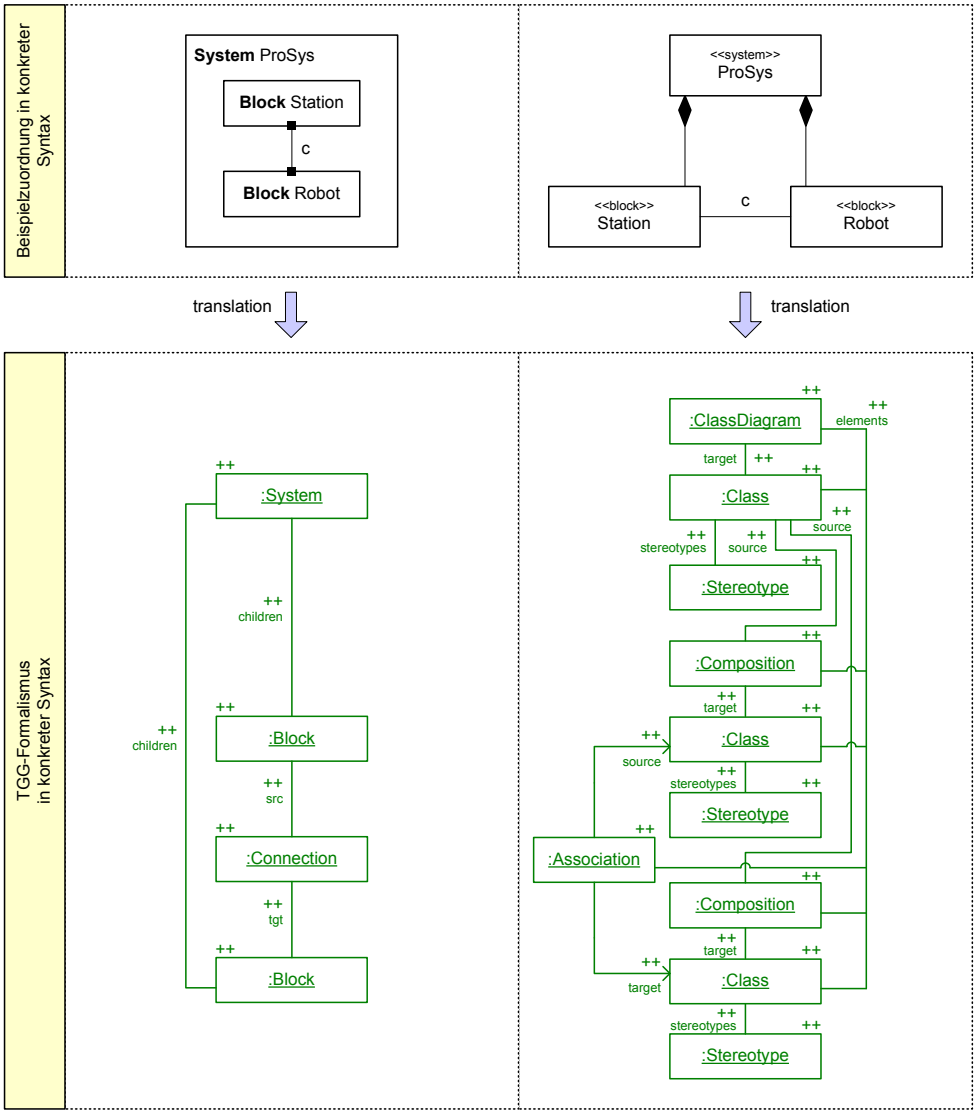


Abbildung 4.20: Beispielzuordnung 4

spielzuordnung zunächst in den TGG-Formalismus übersetzt. Die aus der Übersetzung resultierende Struktur ist in der unteren Hälfte der Abbildung 4.20 dargestellt. Hierbei ist zu erkennen, dass der Kanal, der in der graphischen Syntax als eine einfache Linie zwischen den beteiligten Blöcken dargestellt wurde, nun durch ein eigenständiges Objekt vom Typ `Connection` mit dazugehörigen Links zu den beiden Objekten des Typs `Block` repräsentiert wird. Entsprechend wird auch die Assoziation zwischen den beiden Klassen als ein eigenständiges Objekt vom Typ `Association` repräsentiert.

Die Regelsynthese verläuft nun nach dem bereits aus den anderen Beispielzuordnungen bekannten Schema. Zunächst wird die erste Regel, das heißt, das Axiom aus Abbildung 4.12, angewendet. Die Anwendung und das Ergebnis sind in der Abbildung 4.21 zu sehen.

Nach diesem Syntheseschritt überprüft die Regelsynthese, ob die zuvor synthetisierte Regel aus Abbildung 4.15 angewendet werden kann. Im Gegensatz zu den zuvor vorgestellten Beispielzuordnungen kann diese Regel sogar zwei Mal auf die vorliegende Struktur angewendet werden. Die erste Anwendung der Regel ist in der Abbildung 4.22 dokumentiert. Die zweite Regelanwendung wird in der Abbildung 4.23 illustriert.

In beiden Fällen werden ein Block und eine Klasse mit dazugehörigen Modellelementen durch ein Korrespondenzobjekt vom Typ `CorrNode` zueinander in Beziehung gesetzt. Dabei hat der Synthesealgorithmus, wie in den Abbildungen 4.22 und 4.23 gezeigt, den oberen Block mit der weiter oben dargestellten Klasse und den unteren Block mit der unteren Klasse in Beziehung gesetzt. Ebenso hätte der Synthesealgorithmus den oberen Block mit der unteren Klasse und den unteren Block mit der darüber liegenden Klasse in Beziehung setzen können. Der Grund für diese Auswahl liegt an den zusätzlichen Attributwerten der Objekte, die zum Beispiel den Namen eines Blocks und den Namen einer Klasse enthalten. Diese Attributwerte können zur Ermittlung einer Korrespondenzbeziehung herangezogen werden, indem beispielsweise nur Objekte mit gleichen Attributwerten zueinander zugeordnet werden. Auf die dazu notwendigen Erweiterungen unseres Synthesealgorithmus gehen wir aber erst in Abschnitt 4.2.3 ein. Für den Augenblick reicht uns die hier zunächst willkürlich erscheinende Zuordnung der Modellelemente.

Nach der zweimaligen Anwendung der Regel aus Abbildung 4.15 enthält unsere Struktur zwei weitere Korrespondenzobjekte. Außerdem existieren Objekte, die keine Links zu Objekten besitzen, die mit ++ gekennzeichnet sind. Diese Objekte sind redundant. Die beschriebene Situation illustriert Abbildung 4.24, in der die redundanten Objekte mit einem `x` markiert sind.

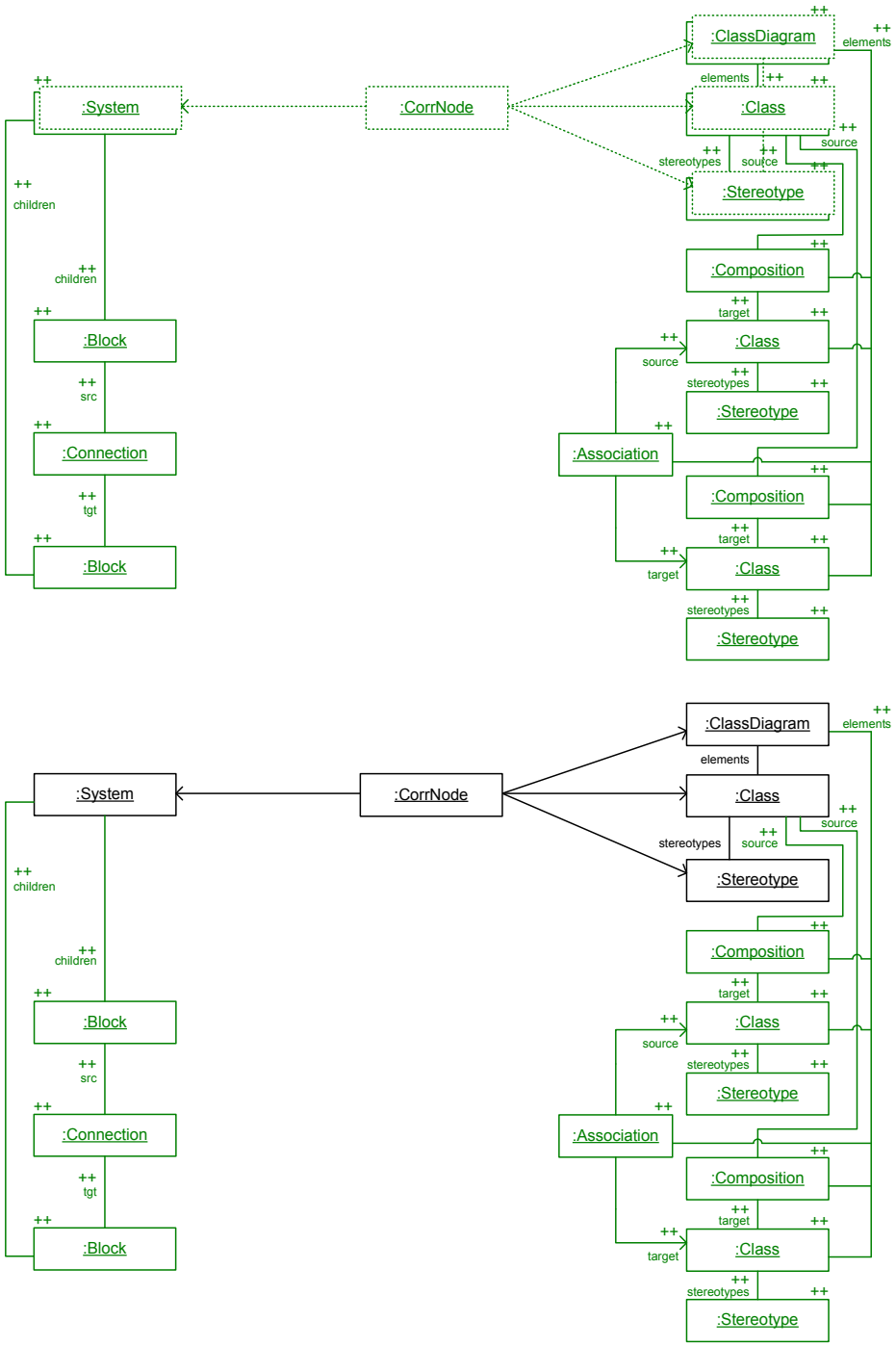


Abbildung 4.21: Regelsynthese aus Beispielzuordnung 4 – Schritte 1 und 2

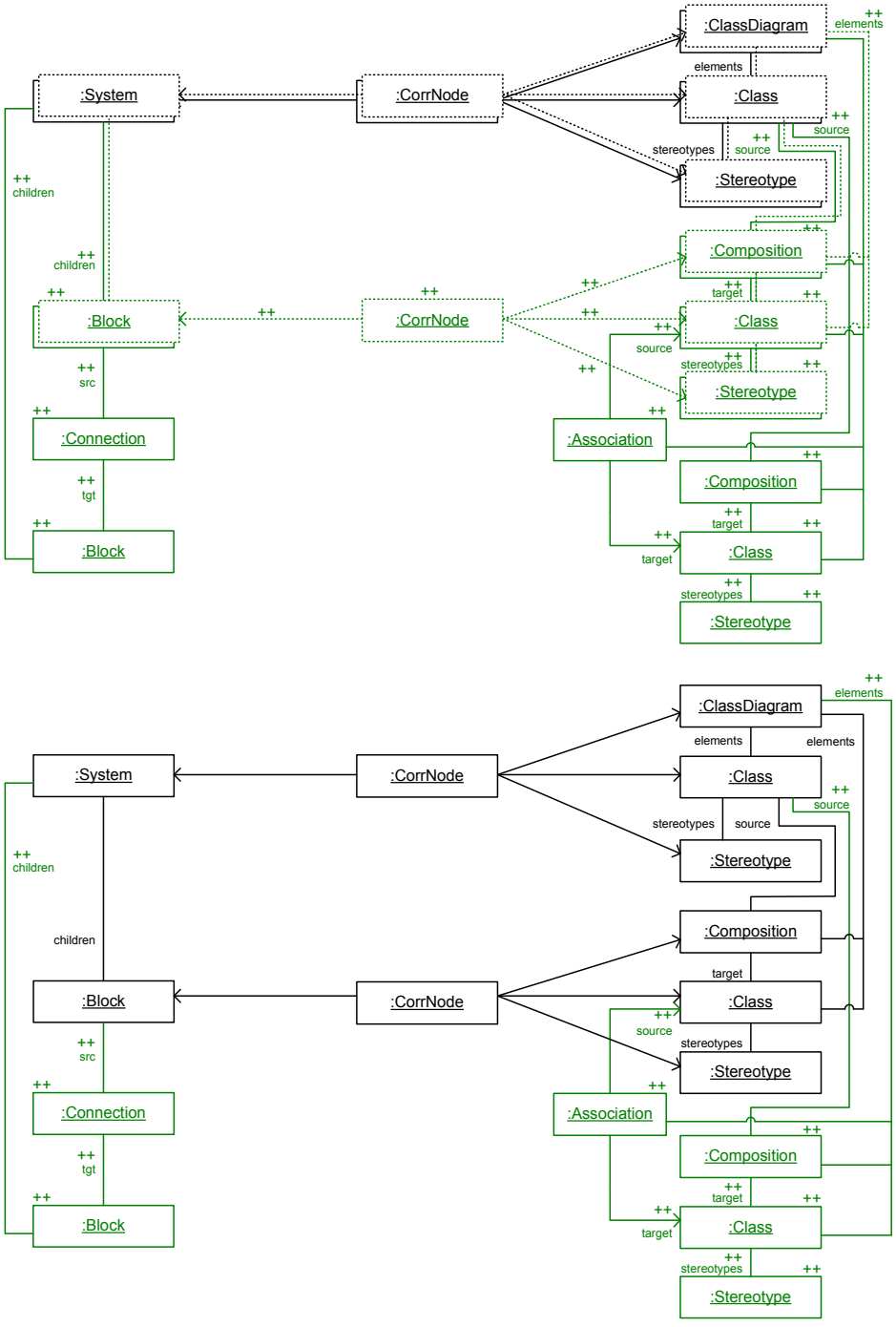


Abbildung 4.22: Regelsynthese aus Beispielzuordnung 4 – Schritte 3 und 4

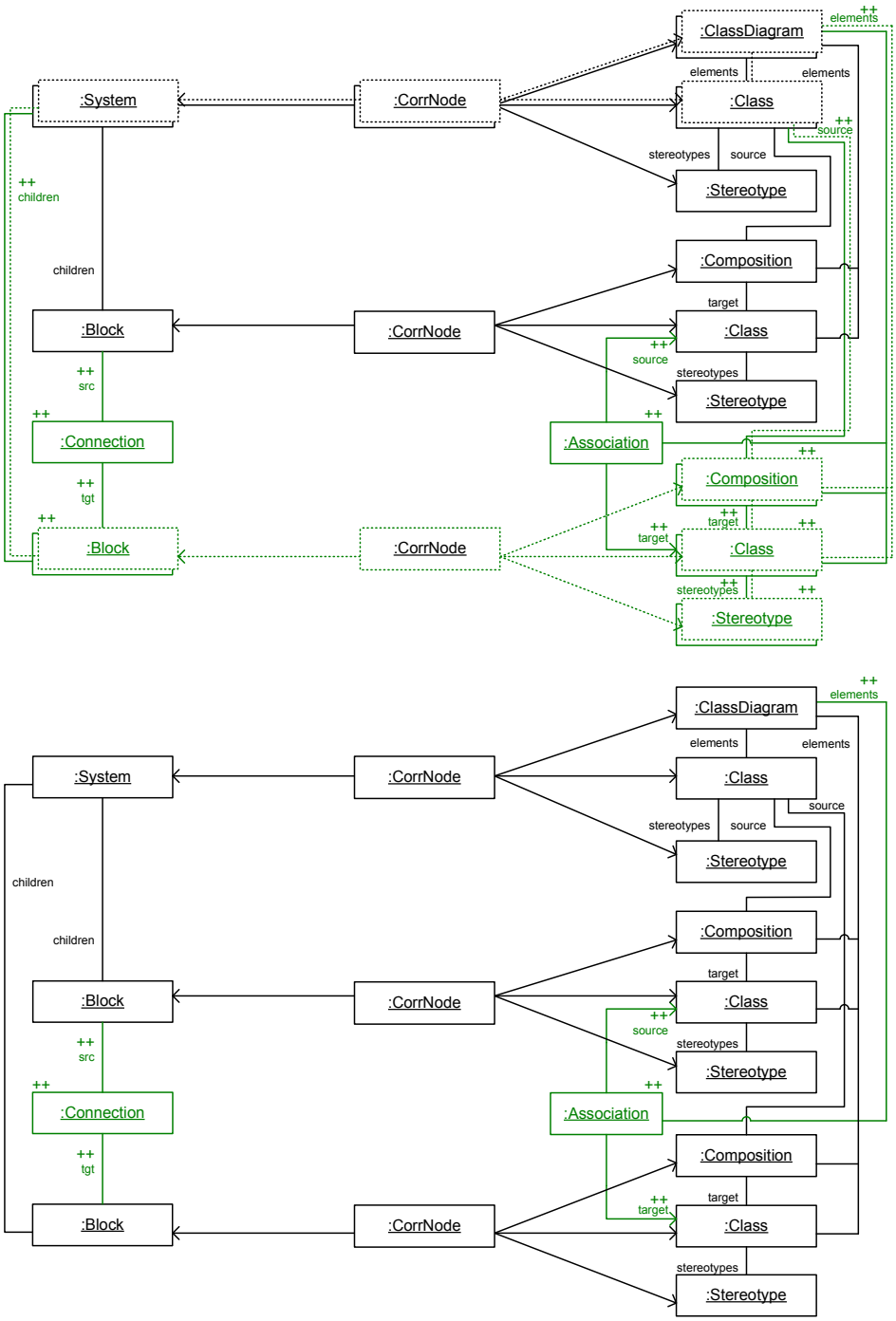


Abbildung 4.23: Regelsynthese aus Beispielzuordnung 4 – Schritte 5 und 6

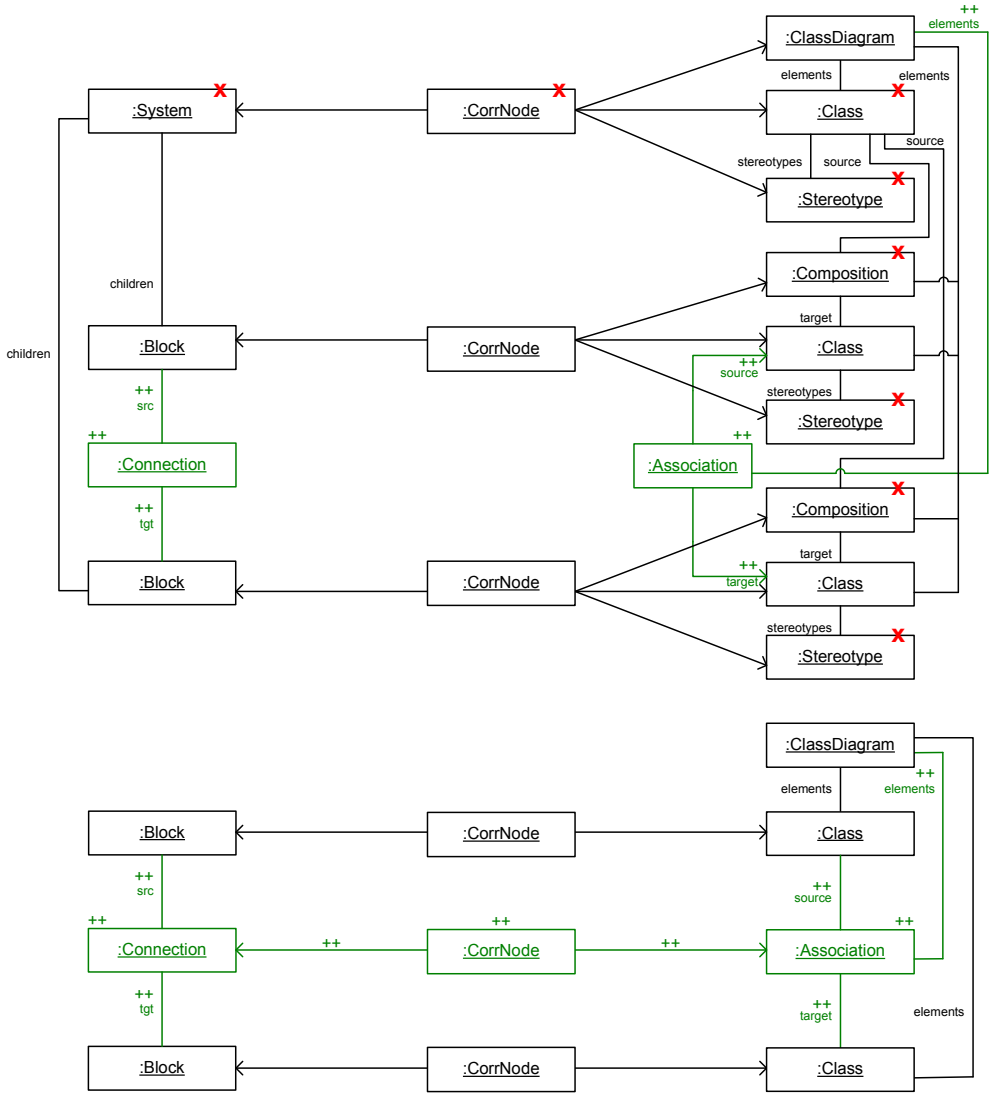


Abbildung 4.24: Regelsynthese aus Beispielzuordnung 4 – Schritte 7 und 8

Aufgrund der Tatsache, dass keine weiteren bisher synthetisierten Regeln auf diese Struktur angewendet werden können, entfernt der Regelsynthesealgorithmus zunächst die redundanten Objekte und alle dazu inzidenten Links. Anschließend fügt er ein neues Korrespondenzobjekt zwischen den verbliebenen und mit ++ gekennzeichneten Objekten ein. Damit ist die Regelsynthese für diese Beispielzuordnung abgeschlossen. Das Ergebnis ist in der unteren Hälfte der Abbildung 4.24 dargestellt.

4.2.3 Erweiterungen

Bisher haben wir in unserem Synthesealgorithmus lediglich Objekte und Links berücksichtigt. Attributbedingungen, negative Anwendungsbedingungen sowie wiederverwendbare Objekte wurden nicht betrachtet. Im Folgenden beschreiben wir, wie diese Konzepte von der Regelsynthese unterstützt werden.

Attributbedingungen

Die Modellelemente in den Beispielzuordnungen enthalten im Regelfall auch Attribute, die mit konkreten Werten belegt sind. Beispielsweise besitzen das System und die Klasse in der Beispielzuordnung aus Abbildung 4.11 einen Namen, der in dem Attribut `name` gespeichert wird. Ebenso wird der Typ des Stereotyps durch einen konkreten Attributwert repräsentiert. Um automatisch korrekte und sinnvolle TGG-Regeln zu synthetisieren, muss der Algorithmus die Attribute in die Synthese mit einbeziehen. Dazu müssen bereits bei der Übersetzung der Beispielzuordnungen in den gemeinsamen TGG-Formalismus die Attributwerte der Objekte beachtet werden. Unter Einbeziehung der Attribute führt dann eine Übersetzung der Beispielzuordnung aus Abbildung 4.11 zu dem Objektdiagramm in Abbildung 4.25. In diesem Objektdiagramm besitzen sowohl das Objekt vom Typ `System` als auch das Objekt vom Typ `Class` ein Attribut `name` vom Typ `String`, die beide mit der Zeichenfolge 'ProSys' belegt sind.

In unserem Ansatz verwenden wir eine sehr einfache Heuristik, die auf der Gleichheit von Zeichenfolgen beziehungsweise Attributwerten im Allgemeinen basiert. Auf Grundlage dieser Heuristik können wir in unserem Beispiel schlussfolgern, dass beide Attribute zueinander in Beziehung stehen. Das bedeutet, dass ein System nur dann zu einer Klasse zugeordnet werden kann, wenn sowohl das System als auch die Klasse den gleichen Namen aufweisen.

Abbildung 4.25: Beispielzuordnung mit Attributwerten

Folglich synthetisieren wir aus der Beispielzuordnung Attributbedingungen, wie sie in Abbildung 4.26 zu sehen sind.

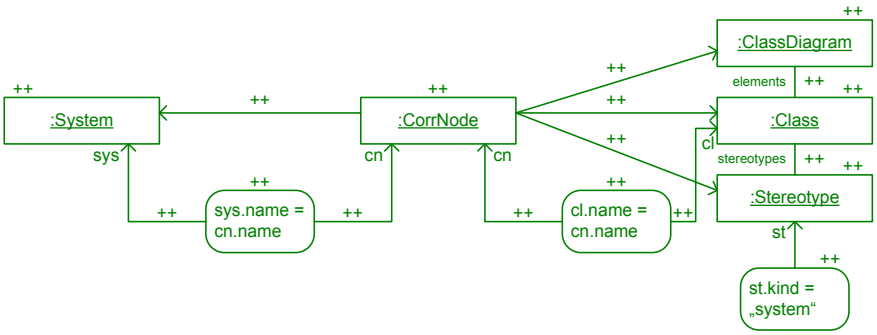


Abbildung 4.26: Synthetisiertes Axiom mit Attributbedingungen

In den Fällen, in denen eine solche Relation zwischen Attributen nicht hergestellt werden kann, synthetisieren wir eine einfache Attributbedingung mit dem konkreten Attributwert der Beispielzuordnung. In unserem Beispiel betrifft dies das Attribut `kind`, welches den Attributwert 'system' im Objekt des Typs `Stereotype` aufweist. Dieser Wert wird in keinem anderen Attribut verwendet. Daher wird hierfür eine einfache Attributbedingung synthetisiert. Sie ist ebenfalls in Abbildung 4.26 dargestellt.

In unserem Beispiel werden für das Attribut eines Stereotypen nur die Werte 'system', 'block' und 'process' verwendet. Diese Attributwerte werden in den Beispielzuordnungen explizit eingesetzt. In einigen Fällen kann aber der Wertebereich eines Attributes aus sehr vielen verschiedenen Werten bestehen, so dass eine Aufzählung dieser konkreten Attributwerte durch die Angabe verschiedener Beispielzuordnungen nicht mehr praktikabel ist. Insbesondere in den Fällen, wo in einer Beispielzuordnung mehrere Attribute

vorhanden sind, steigt auch die Anzahl möglicher Kombinationen. Die Angabe von Beispielzuordnungen, die all diese Kombinationen berücksichtigen, ist zwar theoretisch möglich, aber in der Praxis nicht mehr zu bewältigen. Zudem ist es häufig aus technischen Gründen notwendig, dass in einer Beispielzuordnung ein Attribut mit einem Wert belegt ist, der Attributwert selbst aber für eine korrekte Zuordnung der Modellelemente irrelevant ist.

Diesbezüglich kann die Attributsynthese und Heuristik in unserem Ansatz noch verbessert werden. Beispielsweise könnte eine intelligentere Attributsynthese aufgrund von zwei Beispielzuordnungen, die sich nur durch einen Attributwert in einem Objekt unterscheiden, entschließen, dieses Attribut in der synthetisierten TGG-Regel zu ignorieren. Ebenso wäre es hilfreich, wenn neben der Überprüfung von Zeichenketten auf Gleichheit auch Teilzeichenketten gesucht werden würden, um Präfixe beziehungsweise Suffixe der Attributwerte bestimmen zu können. Diese Strategien sind in unserem Ansatz noch nicht implementiert.

Im Augenblick werden diese Probleme in unserem Ansatz durch die Interaktion mit dem Benutzer gelöst, das heißt, der Synthesealgorithmus macht auf Grundlage der Heuristik Vorschläge für mögliche Attributbedingungen, aber die endgültige Entscheidung muss vom Benutzer der Regelsynthese getroffen werden. In diesem Sinne ist die hier beschriebene Regelsynthese nur semi-automatisch. Allerdings waren bisher in den meisten unserer Beispiele die Vorschläge korrekt und mussten nur noch durch die Benutzer bestätigt werden.

Negative Anwendungsbedingungen

Neben einfachen Attributbedingungen können in TGG-Regeln auch *Negative Anwendungsbedingungen* nützlich sein. Diese Bedingungen haben wir in Abschnitt 3.2.2 bereits vorgestellt. Hier wird nun gezeigt, wie solche Bedingungen mit unserem Ansatz automatisch synthetisiert werden können.

Im Folgenden wollen wir eine Beispielzuordnung angeben, bei der ein Block in einem System zu Elementen eines Klassendiagramms in Beziehung gesetzt wird. Im Gegensatz zu der Beispielzuordnung aus Abbildung 4.13 darf allerdings die neue Zuordnung nur dann stattfinden, wenn das System keinen Prozess enthält.

Um eine TGG-Regel mit einer solchen Anwendungsbedingung synthetisieren zu können, müssen wir zusätzlich zur Beispielzuordnung noch ein Beispiel angeben, das ausdrückt, wann diese Zuordnung nicht stattfinden darf. Diese Anwendungsbedingung geben wir in der konkreten Syntax der Block-

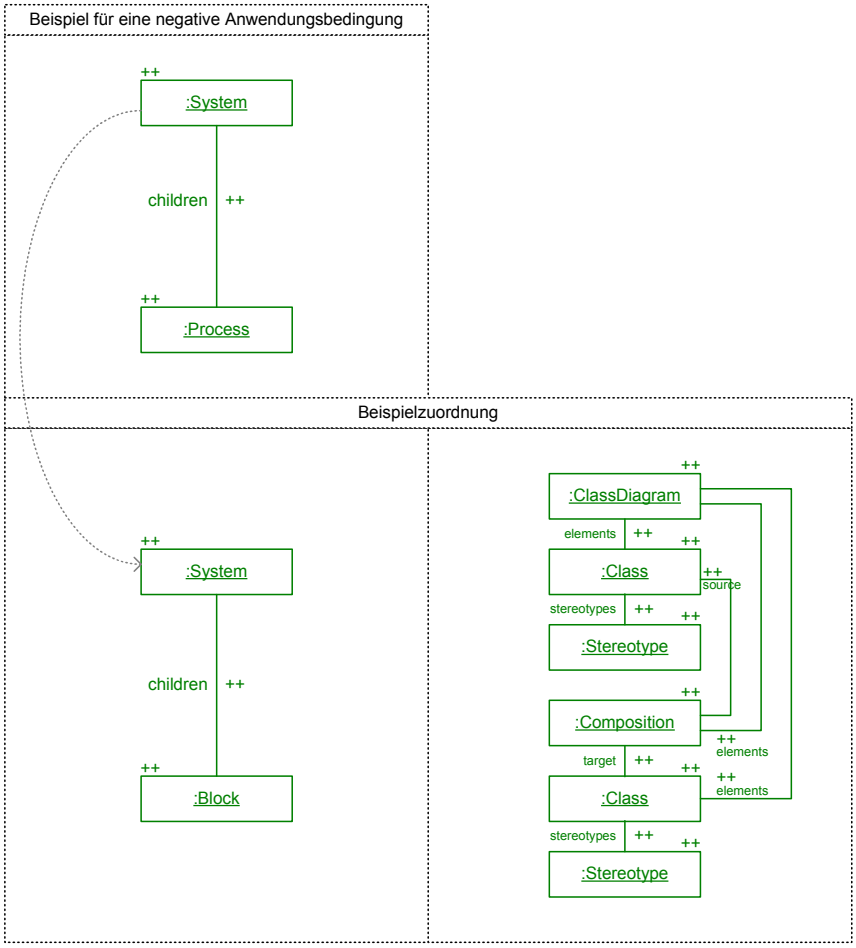


Abbildung 4.27: Beispielzuordnung mit Einschränkung

diagramme an, indem wir ein System mit einem darin enthaltenen Prozess spezifizieren. Die Beispielzuordnung und die Anwendungsbedingung werden nun in den gemeinsamen TGG-Formalismus übersetzt. Sie sind in Abbildung 4.27 dargestellt.

Zur Synthese einer Anwendungbedingung versucht der Algorithmus eine Beziehung zwischen den Objekten der Anwendungsbedingung und den Objekten im Blockdiagramm zu finden. Dabei wird eine Übereinstimmung zwischen den beiden Objekten des Typs `System` festgestellt. Im weiteren Verlauf der Synthese werden beide Objekte zusammengefasst (in Abbildung 4.27 ist dies durch den gestrichelten Pfeil angedeutet). Das verbleibende Objekt und

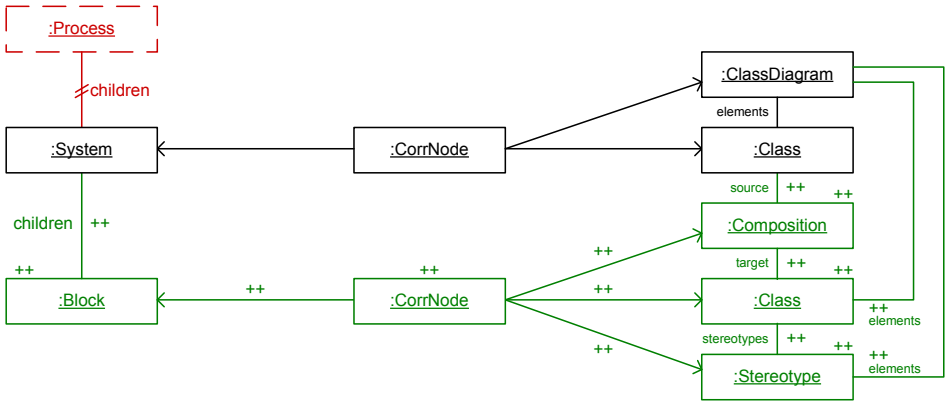


Abbildung 4.28: Synthetisierte Regel mit Negativer Anwendungsbedingung

der Link der Anwendungsbedingung werden als negative Anwendungsbedingung gekennzeichnet. Die Regelsynthese wird wie vorher beschrieben fortgesetzt und resultiert in der TGG-Regel, die in Abbildung 4.28 gezeigt wird. Hierbei haben wir zur Darstellung der negativen Anwendungsbedingung die Kurzschreibweise verwendet.

In unserem Beispiel wurde eine negative Anwendungsbedingung synthetisiert, die besagt, dass kein Link zu einem Objekt des Typs `Process` existieren darf (vergleiche mittlere Anwendungsbedingung in Abbildung 3.18, Seite 80). Um eine Bedingung zu synthetisieren, die fordert, dass kein Link zwischen zwei ganz bestimmten Objekten existiert (vergleiche obere Bedingung der Abbildung 3.18, Seite 80), hätte auch der Prozess an ein Objekt der Beispielzuordnung gebunden werden müssen.

Wiederverwendbare Objekte

In Abschnitt 3.2.2 haben wir gezeigt, dass es durchaus Szenarien gibt, in denen ein Modellelement nur dann erzeugt werden soll, wenn das Modellelement in dieser Form noch nicht existiert. Dies kann kann beispielsweise durch die Spezifikation von zwei TGG-Regeln erfolgen, bei der eine entsprechende Fallunterscheidung mit Hilfe von negativen Anwendungsbedingungen ausgedrückt wird. Allerdings wird durch diese Art der Spezifikation die Komplexität der einzelnen TGG-Regeln erhöht. Darüber hinaus leidet durch die Fallunterscheidung, für die zwei separate TGG-Regeln spezifiziert werden müssen, die Übersichtlichkeit und Verständlichkeit des gesamten TGG-Regelsatzes.

Um die Komplexität der TGG-Regeln gering zu halten und redundante TGG-Regeln zu vermeiden, haben wir daher in Abschnitt 3.2.2 wiederverwendbare Objekte eingeführt. Während bei der direkten Spezifikation von TGG-Regeln die Modellelemente einfach durch den Benutzer als wiederverwendbare Objekte markiert werden, müssen bei der automatischen Regelsynthese solche Modellelemente gesondert behandelt werden. Um bei der automatischen Regelsynthese zwischen herkömmlichen Modellelementen und solchen, die wiederzuverwenden sind, unterscheiden zu können, muss der Benutzer bereits im Vorfeld der Regelsynthese die Typen der wiederzuverwendenden Objekte zu einer hierfür speziell vorgesehenen Menge hinzufügen. Dadurch werden während der automatischen Regelsynthese alle Instanzen, deren Typ in dieser Menge enthalten ist, automatisch als wiederverwendbare Objekte in den synthetisierten TGG-Regeln ausgewiesen.

4.2.4 Reihenfolgeunabhängigkeit

Bisher wurden die Beispielzuordnungen in einer sehr vorteilhaften Reihenfolge angegeben – die zuvor synthetisierten TGG-Regeln konnten immer auf die nachfolgende Beispielzuordnung angewendet werden. Aus diesem Grund konnte der Regelsynthesealgorithmus TGG-Regeln synthetisieren, die sehr ähnlich zu den von Hand spezifizierten TGG-Regeln sind. Falls die Reihenfolge der Beispielzuordnungen nicht so vorteilhaft gewählt ist wie in unserem Beispiel, kann der bisher präsentierte Algorithmus keine optimalen TGG-Regeln erzeugen. Dies wird an dem folgenden Szenario kurz verdeutlicht.

Beispiel für Reihenfolgeabhängigkeit Für das folgende Beispiel nehmen wir an, dass die Reihenfolge, in der die Beispielzuordnungen dem Synthesealgorithmus zugeführt werden, geändert wird. Wir fangen wieder mit der Beispielzuordnung an, die ein leeres System einer Klasse mit einem Stereotypen zuordnet (siehe Abbildung 4.11 auf Seite 122). Jetzt wird allerdings zuerst die Beispielzuordnung aus Abbildung 4.20 angegeben und anschließend die Beispielzuordnung aus Abbildung 4.13. Aus den in dieser Reihenfolge gegebenen Beispielzuordnungen erzeugt unser Synthesealgorithmus wieder TGG-Regeln. Die erste Beispielzuordnung führt wie im vorherigen Fall zu dem synthetisierten Axiom aus Abbildung 4.12. Aus der zweiten Beispielzuordnung wird hingegen die in Abbildung 4.29 gezeigte TGG-Regel synthetisiert, während aus der dritten Beispielzuordnung die in der unteren Hälfte der Abbildung 4.15 dargestellte TGG-Regel entsteht.

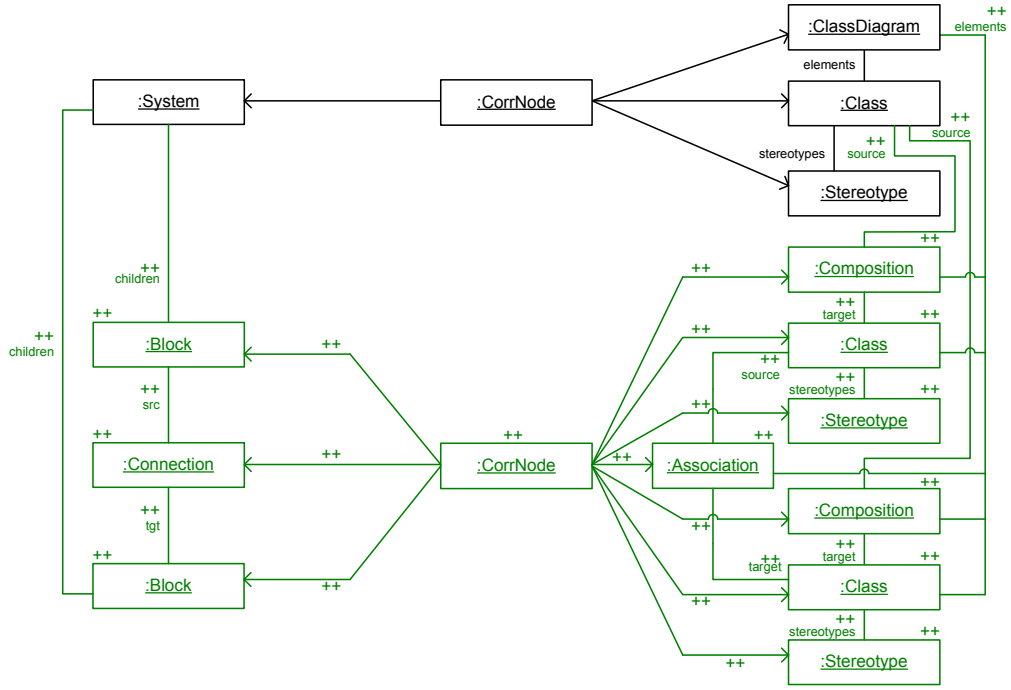


Abbildung 4.29: TGG-Regel resultierend aus geänderter Reihenfolge der Beispielzuordnungen

Die in der unteren Hälfte der Abbildung 4.24 gezeigte TGG-Regel und die TGG-Regel aus der Abbildung 4.29 wurden beide aus denselben Beispielzuordnungen synthetisiert. Trotzdem sind die synthetisierten TGG-Regeln verschieden. Dies liegt an der unterschiedlichen Reihenfolge, in der die Beispielzuordnungen dem Synthesealgorithmus zugeführt werden. Aufgrund der neuen Reihenfolge existiert zu dem Zeitpunkt, an dem die Regelsynthese aus der zweiten Beispielzuordnung stattfindet, nur die synthetisierte Regel aus Abbildung 4.12. Daher kann nur diese Regel auf die Struktur der zweiten Beispielzuordnung angewendet werden. Im Gegensatz dazu lagen im ersten Fall bereits zwei Regeln vor, die auf die neue Struktur angewendet werden konnten. Bei der neuen Reihenfolge existiert zu diesem Zeitpunkt nur eine Regel, so dass weniger Objekte gebunden werden, was wiederum dazu führt, dass die daraus synthetisierte TGG-Regel mehr Objekte enthält und sich dadurch von der TGG-Regel aus Abbildung 4.24 unterscheidet. Diese Abhängigkeit von der Reihenfolge der gegebenen Beispielzuordnungen ist nicht gewünscht. Die Regelsynthese sollte unabhängig von der Reihenfolge der gegebenen Beispielzuordnungen immer zu identischen TGG-Regeln führen.

Reihenfolgeunabhängigkeit durch Algorithmuserweiterung Damit der Synthesealgorithmus von der gegebenen Reihenfolge der Beispielzuordnungen unabhängig wird, muss er erweitert werden. In der erweiterten Version unserer Regelsynthese werden nicht nur bereits synthetisierte Regeln auf die neue TGG-Regel angewendet, sondern auch die neu synthetisierte TGG-Regel auf die Menge der bereits synthetisierten TGG-Regeln. Wenn dabei eine Übereinstimmung der neuen TGG-Regel mit einer der bereits synthetisierten TGG-Regeln festgestellt wird, verfährt der Algorithmus dabei wie im gewöhnlichen Fall. Die ++ Markierungen der durch die Regelanwendung gebundenen Objekte und Links werden entfernt, zusätzlich benötigte Korrespondenzobjekte hinzugefügt und Objekte, die keine direkte Verbindung zu Objekten mit ++ Markierungen besitzen, gelöscht.

Aufgrund dieser Erweiterung wird in unserem Beispiel nach der Synthese der dritten TGG-Regel (siehe Abbildung 4.15) versucht, diese TGG-Regel auf die bereits zuvor synthetisierten TGG-Regeln anzuwenden. Dabei wird festgestellt, dass die neue TGG-Regel auf die im zweiten Schritt synthetisierte TGG-Regel aus Abbildung 4.29 anwendbar ist. Daher wird die ++ Markierung der gebundenen Objekte entfernt und ein neues Korrespondenzobjekt hinzugefügt. Alle nicht benötigten Objekte werden entfernt. Dadurch entsteht eine TGG-Regel, wie wir sie bereits aus Abbildung 4.24 kennen.

Die Erweiterung unseres Synthesealgorithmus hat einen weiteren Effekt. Vergleichen wir die von Hand spezifizierten TGG-Regeln mit den bisher automatisch synthetisierten Regeln, so stellen wir fest, dass die TGG-Regel aus Abbildung 4.15 in der von Hand spezifizierten Menge der TGG-Regeln nicht vorkommt. Dies liegt daran, dass dieser Fall bereits durch die TGG-Regel aus Abbildung 4.18 abgedeckt wird. Damit wird die automatisch synthetisierte TGG-Regel aus Abbildung 4.15 nicht benötigt. Genau genommen ist sogar die zuvor automatisch synthetisierte Regelmenge mehrdeutig, da nun bei einem Objekt des Typs `System` sowohl die TGG-Regel aus Abbildung 4.15 als auch die TGG-Regel aus Abbildung 4.18 anwendbar ist. Dies liegt daran, dass im Metamodell für Blockdiagramme die Klasse `System` von der Klasse `Block` erbt. Damit kann ein Objekt des Typs `Block` an ein Objekt des Typs `System` gebunden werden. Diese Mehrdeutigkeit von TGG-Regeln ist in den allermeisten Fällen nicht gewünscht und sollte nach Möglichkeit vermieden werden, da dies bei der späteren Anwendung der TGG-Regeln zu unterschiedlichen Ergebnissen führen kann.

Durch die Erweiterung unseres Synthesealgorithmus in der bereits beschriebenen Art und Weise werden solche mehrdeutigen TGG-Regeln erkannt und automatisch eliminiert. In unserem Beispiel wird die synthetisierte TGG-Regel, wie sie in Abbildung 4.19 zu sehen ist, sofort als neue TGG-Regel betrachtet und in die Synthese einbezogen. Daher versucht der Regelsynthesealgorithmus diese TGG-Regel auf die bereits vorhandenen TGG-Regeln anzuwenden. Dabei kann der Algorithmus das Objekt vom Typ `Block` an das Objekt vom Typ `System` der TGG-Regel aus Abbildung 4.15 aus den bereits beschriebenen Gründen binden. Der Synthesealgorithmus entfernt daher alle ++ Markierungen der gebundenen Objekte und Links. Aufgrund der Tatsache, dass in diesem Fall aber alle Objekte und Links gebunden werden konnten, bleiben keine Objekte mit ++ Markierungen übrig. Damit existieren auch keine Objekte, die mit neu erzeugten Objekten direkt verbunden sind. Somit werden alle Objekte gelöscht und damit diese TGG-Regel eliminiert. Die automatisch synthetisierte TGG-Regeln stimmen damit weitestgehend mit den von Hand spezifizierten TGG-Regeln überein.

Zusammenfassend kann festgehalten werden, dass durch die Erweiterung unseres Synthesealgorithmus es nicht mehr nötig ist, die Beispielzuordnungen Schritt für Schritt in einer bestimmten Reihenfolge anzugeben – der Benutzer kann einfach eine Menge von Beispielzuordnungen als Eingabe an den Synthesealgorithmus übergeben und erhält eine Menge von TGG-Regeln als Ausgabe. Darüber hinaus ist der erweiterte Synthesealgorithmus in der Lage, mehrdeutige TGG-Regeln zu erkennen und sie zu eliminieren, um eine

möglichst kleine und optimale Regelmenge zu konstruieren.

4.2.5 Abschließende Betrachtungen zur Regelsynthese

In diesem Abschnitt haben wir einen Algorithmus präsentiert, der TGG-Regeln aus zueinander korrespondierenden Beispielzuordnungen synthetisiert. Die Beispielzuordnungen werden in der konkreten Syntax der beteiligten Modelle angegeben. Dadurch ist die Spezifikation dieser Beispielzuordnungen benutzerfreundlicher und die genaue Kenntnis der zugrundeliegenden Metamodelle sowie des TGG-Formalismus wird nicht zwingend benötigt.

Der hier an einem Beispiel vorgestellte Synthesealgorithmus funktioniert vollautomatisch, sofern keine Objektattribute betrachtet werden müssen. Der Algorithmus ist unabhängig von der gegebenen Reihenfolge der Beispielzuordnungen. Allerdings reicht eine einzige Beispielzuordnung nicht aus, um eine allgemein gültige Menge von TGG-Regeln zu synthetisieren, die in der Lage ist, unterschiedliche Modellinstanzen zueinander in Beziehung zu setzen. Vielmehr werden viele verschiedene Beispielzuordnungen benötigt. Dabei muss eine Beispielzuordnung eine Übereinstimmung mit mindestens einer anderen Beispielzuordnung besitzen und zusätzlich ein weiteres Konzept der Modellierungssprache einführen. Diese Differenz zwischen den Beispielzuordnungen wird durch den Synthesealgorithmus ausgenutzt, um daraus entsprechende TGG-Regeln zu extrahieren. Falls ein Konzept der Modellierungssprache in keiner Beispielzuordnung auftaucht, wird dieses Konzept auch in keiner der synthetisierten TGG-Regeln berücksichtigt. Auf der anderen Seite können aus Beispielzuordnungen, die mehrere Konzepte auf einmal enthalten nur TGG-Regeln synthetisiert werden, die diese Konzepte in dieser Kombination abdecken. Daher muss bei der Anwendung der Regelsynthese darauf geachtet werden, dass mit einer möglichst kleinen Beispielzuordnung begonnen wird und sowohl die Menge der Beispielzuordnungen als auch die Größe der Beispielzuordnungen selbst Schritt für Schritt erweitert werden.

Die Regelsynthese unterstützt einen iterativen Entwurfsprozess. Der iterative Entwurfsprozess ist in Abbildung 4.30 dargestellt. Der Prozess beginnt mit der Definition der Beispielzuordnungen. Aus diesen Beispielzuordnungen werden TGG-Regeln synthetisiert. Die TGG-Regeln können durch den Benutzer validiert werden, zum Beispiel indem sie auf einer definierten Eingabe ausgeführt und das Ergebnis mit dem erwarteten Ergebnis verglichen wird. Falls das Ergebnis der Validierung mit dem erwarteten Resultat nicht übereinstimmt, können die Beispielzuordnungen solange modifiziert, verfeinert und um neue Beispielzuordnungen ergänzt werden, bis die Validierung

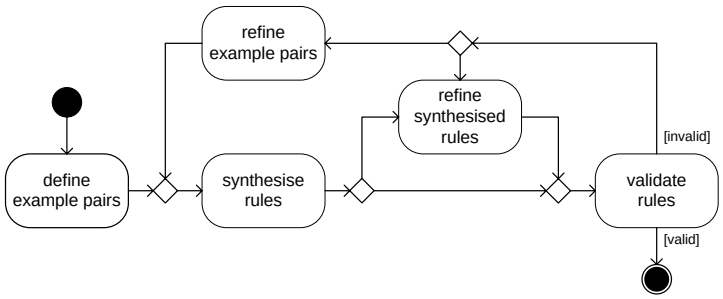


Abbildung 4.30: Überblick zum Prozess

zufriedenstellende Ergebnisse liefert.

Der Regelsynthesealgorithmus funktioniert nur dann vollautomatisch, solange keine Attribute berücksichtig werden müssen. Sind hingegen auch Attribute zu berücksichtigen, arbeitet der Algorithmus interaktiv, das heißt, der Algorithmus macht zwar automatisch Vorschläge für Attributbedingungen, die endgültige Entscheidung, welche Attributbedingungen in der TGG-Regel verwendet werden sollen, muss aber vom Benutzer getroffen werden. Wie schon zuvor erwähnt, kann bei vielen Attributen bzw. Attributen mit einem großen Wertebereich die Spezifikation mit Beispielzuordnungen sehr umständlich werden, da für jeden möglichen Fall eine eigene Beispielzuordnung angegeben werden muss. Enthält ein Modell einer Beispielzuordnung beispielsweise zwei Attribute, wobei das erste Attribut m verschiedene Werte und das zweite Attribut n Werte annehmen kann, so sind insgesamt $m*n$ verschiedene Kombinationen möglich, für die jeweils eine eigene Beispielzuordnung nötig wäre. Der Aufwand hierfür kann so groß werden, dass die Vorteile dieses Ansatzes nicht mehr zu rechtfertigen sind.

Aus diesem Grund erlauben wir in unserem Ansatz, die synthetisierten TGG-Regeln von Hand zu verfeinern. Dadurch können insbesondere auch die Attributbedingungen von Hand angepasst und zum Beispiel durch eine einfache Abbildungsfunktion realisiert werden. Dadurch wird der Ansatz zwar semi-automatisch, allerdings reicht eine einzige Beispielzuordnung in Verbindung mit einer von Hand spezifizierten Abbildungsfunktion, um verschiedene Attributwerte in einer Beispielzuordnung abzudecken. Erlauben wir manuelle Anpassungen und Verfeinerungen der synthetisierten TGG-Regeln, so macht in diesem Fall auch die Validierung auf der Grundlage der gegebenen Beispielzuordnungen wieder Sinn, da jetzt nicht nur die Korrektheit des Synthesealgorithmus geprüft wird, sondern auch die manuell

durchgeführten Änderungen validiert werden.[4]

In unserem Beispiel entsprechen die synthetisierten TGG-Regeln konzeptionell den von Hand spezifizierten TGG-Regeln. Beispielweise entspricht die synthetisierte Regel in Abbildung 4.19 der von Hand spezifizierten TGG-Regel aus Abbildung 3.6. Der einzige Unterschied besteht in dem Typ der eingesetzten Korrespondenzobjekte. In der automatisch synthetisierten TGG-Regel entsprechen alle Korrepondenzobjekte demselben Typ `CorrNode`, während in der von Hand spezifizierten TGG-Regel die Korrespondenzobjekte unterschiedliche Typen aufweisen. Die Erstellung eines angepassten Metamodells für Korrespondenztypen wird im Augenblick von der automatischen Regelsynthese nicht unterstützt. Eine erste Möglichkeit für die Erstellung eines angepassten Korrespondenzmetamodells haben wir in [KW07] vorgestellt. Dabei wurden die eingesetzten Korrespondenztypen während der Regelsynthese durchnummeriert. Damit haben diese Korrespondenztypen die Reihenfolge der synthetisierten TGG-Regeln wiedergegeben. Um automatisch ein Korrespondenzmetamodell ähnlich dem in Abbildung 3.10 zu erhalten, muss dieses Verfahren jedoch noch erweitert und verallgemeinert werden.

Ein weiterer Unterschied fällt auf, wenn wir die synthetisierte TGG-Regel aus Abbildung 4.24 mit der manuell spezifizierten TGG-Regel aus Abbildung 3.8 vergleichen. Der Unterschied besteht in dem zusätzlichen Link zwischen dem Objekt des Typs `ClassDiagram` und der zum Block korrespondierenden Klasse `Class`. Tatsächlich besitzt ein Klassendiagramm immer einen Link zu seinen Klassen. Damit ist die synthetisierte TGG-Regel auch korrekt. In der manuell spezifizierten TGG-Regel fehlt dieser Link. Hier ist der Spezifizierer flexibler und kann einige der Links – sofern sie nicht zwingend notwendig sind – einfach weglassen. In unserem Beispiel muss er nur sicherstellen, dass das Objekt des Typs `ClassDiagram` über einen Link erreichbar ist, das heißt, er muss sicherstellen, das eine zusammenhängende Struktur vorhanden ist. Hierzu kann er beide Links angeben oder aber auch nur einen der Links. Alle diese Varianten führen zu einer gültigen und ausführbaren TGG-Regel.

[4] Mit den Möglichkeiten zur Validierung von TGG-Regeln werden wir uns noch in Kapitel 6 beschäftigen.

4.3 MOF 2.0 Query/View/Transformation

Die in dieser Arbeit vorgestellte Technik zur Modellsynchronisation wurde zeitgleich zum Query/View/Transformation (QVT) Standard der Object Management Group (OMG) entwickelt [QVT08]. Das Hauptanwendungsgebiet dieses erst kürzlich in der finalen Version veröffentlichten QVT-Standards stellen Modelltransformationen dar. Zur Spezifikation einer Modelltransformation definiert der QVT-Standard sowohl einen *delarativen* als auch einen *imperativen* Sprachanteil. Im deklarativen Sprachanteil wird die Transformation durch Beziehungen zwischen einzelnen Mustern der Modelle beschrieben. Der imperative Sprachanteil besteht hingegen aus operationalen Anweisungen, mit denen die Modelltransformation direkt gesteuert wird.

In Abbildung 4.31 ist die QVT-Spracharchitektur zu sehen. Der deklarative Sprachanteil wird durch die Sprachen *QVT-Relations* und *QVT-Core* abgedeckt. Der operationale Sprachanteil setzt sich hingegen aus der Sprache *Operational Mappings Language* und den sogenannten *Black-Box-Implementierungen* zusammen. Die Black-Box-Implementierungen können in beliebigen Sprachen programmiert und zusammen mit der Operational Mappings Language eingesetzt werden, um die beiden deklarativen Sprachen QVT-Relations und QVT-Core um komplexere Transformationsalgorithmen zu ergänzen, die deklarativ gar nicht oder mit nur sehr viel Aufwand beschrieben werden können. Im Rahmen dieser Arbeit konzentrieren wir uns allerdings nur auf die beiden deklarativen Sprachen.

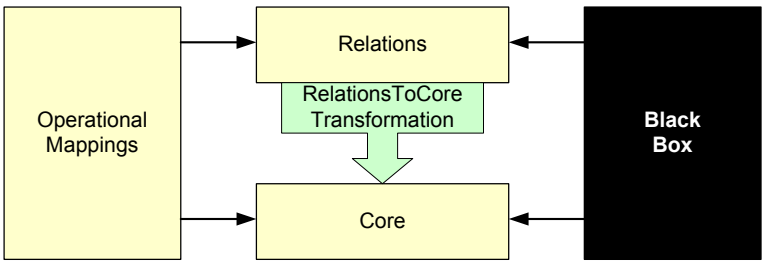


Abbildung 4.31: QVT-Spracharchitektur, entnommen aus [QVT08]

Die Spezifikation mit QVT-Relations kann sowohl in einer textuellen als auch in einer graphischen Syntax erfolgen. In Abbildung 4.32 ist die Spezifikation der TGG-Regel `Block2Class` aus Abschnitt 3.2 in der graphischen Syntax von QVT-Relations dargestellt.

Gegenüber einer TGG-Regel fallen zwei wesentliche Unterschiede auf. Der

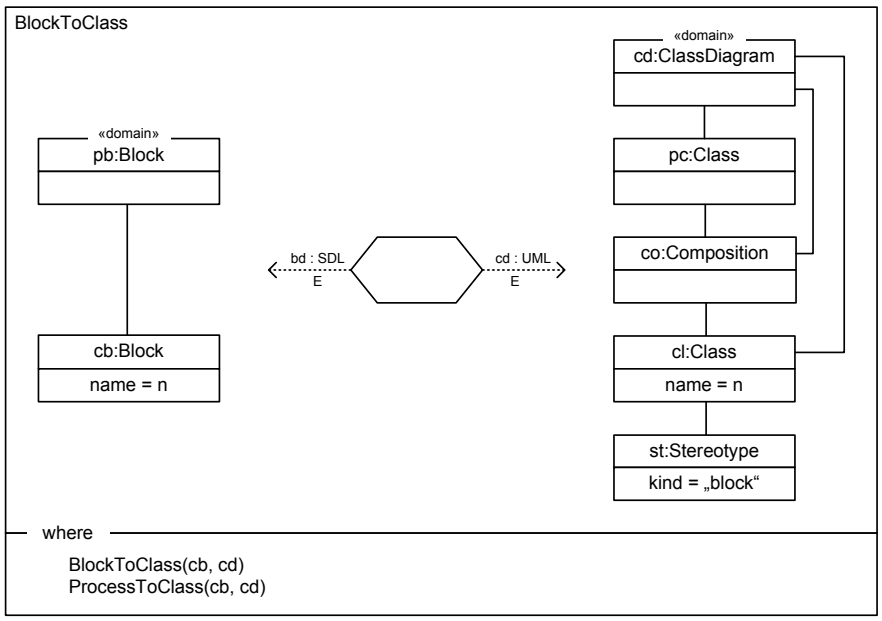


Abbildung 4.32: Beispielregel *BlockToClass* in der graphischen Syntax von QVT-Relations

erste Unterschied ist, dass keine Korrespondenzknoten zwischen den einzelnen Modellelementen spezifiziert werden. Der zweite Unterschied besteht darin, dass in einer QVT-Regel andere QVT-Regeln mit Hilfe einer *where*-Klausel[5] referenziert werden können. Die referenzierten Regeln werden aufgerufen, falls die betrachtete Regel erfolgreich ausgeführt werden konnte. Im Gegensatz zu QVT-Relations sind im TGG-Formalismus die Abhängigkeiten zwischen den TGG-Regeln nur implizit enthalten. Insgesamt sind beide Notationen jedoch sehr ähnlich.

Die Sprache QVT-Core bildet den Kern der QVT-Spezifikation. Zur Spezifikation einer Modelltransformation mit QVT-Core ist allerdings nur eine textuelle Syntax verfügbar. Gegenüber QVT-Relations werden weniger Konstrukte bereitgestellt, so dass die Sprachdefinition von QVT-Core kompakter ausfällt. Auf der einen Seite führt dies dazu, dass die Spezifikationen auf Grundlage von QVT-Core aus vielen einfachen Ausdrücken zusammengesetzt werden müssen. Auf der anderen Seite kann wegen des geringen Umfangs der Sprache die Semantik einfacher definiert werden. Daher wird die Semantik

[5]Darüber hinaus existieren weitere Konzepte, wie z. B. *when*-Klauseln. Die Syntax und Semantik von QVT-Relations ist in der QVT-Spezifikation beschrieben [QVT08].

von QVT-Relations mit einer Transformation definiert. Die hierzu notwendige *Relations2Core-Transformation* (siehe Abbildung 4.31) ist Bestandteil der QVT-Spezifikation.

Das Schema einer Regel in der Sprache QVT-Core haben wir in der Abbildung 4.33 dargestellt. Diese Darstellung haben wir unverändert aus der QVT-Spezifikation übernommen. Bei einem Vergleich zwischen dem Schema einer Regel der Sprache QVT-Core und dem Schema einer TGG-Regel (siehe dazu insbesondere auch Abbildung 5.6 auf Seite 171), fällt die Ähnlichkeit zwischen den Ansätzen besonders deutlich auf.

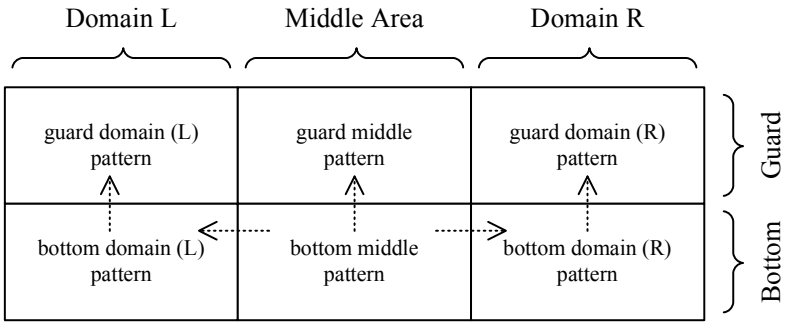


Abbildung 4.33: Schema einer QVT-Core-Regel, entnommen aus [QVT08]

Das in Abbildung 4.33 gezeigte Schema besteht aus drei Spalten, die in der QVT-Spezifikation *Bereiche* (engl. Area) genannt werden. Der linke Bereich (*Domain L*) und der rechte Bereich (*Domain R*) repräsentieren die in Beziehung stehenden Modellierungsdomänen. Dazwischen existiert ein mittlerer Bereich (*Middle Area*), der allerdings erst während einer Modelltransformation hinzugefügt wird. Dieser Bereich enthält die sogenannten *Trace-Klassen*, mit denen Beziehungen zwischen den Elementen der Modellierungsdomänen explizit verwaltet werden. Diese Trace-Klassen entsprechen weitestgehend den Korrespondenzknoten einer TGG-Regel. Weiterhin ist jede der drei Spalten in zwei Muster – das sogenannte *guard pattern* und das *bottom pattern* – unterteilt. Die Muster repräsentieren Objektstrukturen und Bedingungen, wie wir sie bereits aus TGG-Regeln kennen.

Diese Übereinstimmungen wurden ausgenutzt, um eine Transformation von QVT-Core in TGGs zu definieren [Gre06]. Zusammen mit der in der QVT-Spezifikation angegebenen Transformation von QVT-Relations in QVT-Core ist es somit möglich, die Beziehungen für eine Modelltransformation oder Modellsynchronisation in einer der beiden deklarativen QVT-

Sprachen zu spezifizieren und diese Spezifikation mit Hilfe von TGGs auszuführen. In Abbildung 4.34 wird das allgemeine Vorgehen verdeutlicht.

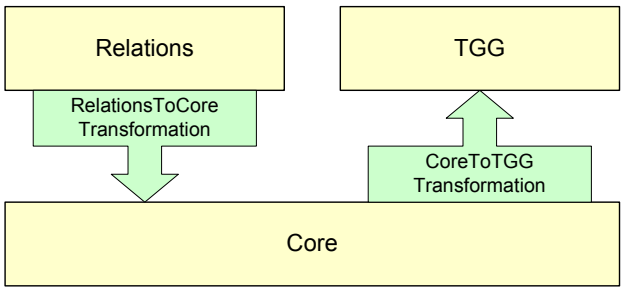


Abbildung 4.34: Abbildung von QVT-Relations auf TGGs

Auf die Transformation von QVT in TGGs wird in dieser Arbeit allerdings nicht weiter eingegangen. Der Grund hierfür ist, dass diese Transformation in der Diplomarbeit von Joel Greenyer [Gre06] dokumentiert ist. Darüber hinaus wurden in der Dissertation von Alexander Königs bereits beide Ansätze miteinander kombiniert, so dass die Akzeptanz für den TGG-Ansatz erhöht und die Technik der TGGs für einen weiten Anwenderkreis zugänglich gemacht wurde [Kön08].

4.4 Zusammenfassung

In diesem Kapitel haben wir uns mit Alternativen zur Spezifikation von Korrespondenzbeziehungen beschäftigt. Zwei dieser Spezifikationsvarianten liegt der Ansatz der Tripel-Graph-Grammatiken zugrunde, so dass beide Alternativen zu einer formalen Spezifikation der Korrespondenzbeziehungen führen. Die dritte Spezifikationsvariante basiert auf dem erst kürzlich verabschiedeten Standard Query/View/Transformation (QVT) der Object Management Group (OMG) [QVT08].

In Abschnitt 4.1 haben wir die erste Spezifikationsvariante vorgestellt. Diese Spezifikationsvariante ist besonders gut geeignet, um Modell-zu-Text Beziehungen, wie sie zum Beispiel bei der Codegenerierung benötigt werden, zu definieren. Auch wenn die Spezifikation von Modell-zu-Text Beziehungen bereits mit dem grundlegenden Ansatz der TGGs möglich ist, so stellt die in diesem Abschnitt vorgestellte Kombination aus TGGs und Textschablonen eine deutliche Vereinfachung für den Entwickler dar. Neben den Vorteilen

sind wir auch auf die Nachteile dieses Ansatzes eingegangen, die mit der zweiten Spezifikationsvariante jedoch behoben werden.

Mit der zweiten Spezifikationsvariante haben wir uns in Abschnitt 4.2 befasst. Bei diesem Ansatz werden die Korrespondenzbeziehungen in der Notation der beteiligten Sprachen durch eine Menge von jeweils zwei zueinander korrespondierenden Beispielen angegeben, die wir als Beispielzuordnungen bezeichnet haben. Aus einer Menge solcher Beispielzuordnungen konnten die entsprechenden TGG-Regeln (semi-) automatisch synthetisiert werden. Obwohl wir die Regelsynthese an unserem durchgängigen Beispiel von Block- und Klassendiagrammen vorgestellt haben, eignet sich dieser Ansatz – wie in der Diplomarbeit von Alexander Geburzi praktisch gezeigt wurde [Geb06] – insbesondere auch zur Spezifikation von Modell-zu-Text Beziehungen, die zur Codegenerierung und Synchronisation genutzt werden können.

In Abschnitt 4.3 sind wir auf die Spezifikation von Korrespondenzbeziehungen mit QVT eingegangen. Der QVT-Standard definiert zwei deklarative Sprachen zur Modelltransformation, für die jedoch lange Zeit keine Werkzeugunterstützung existiert hat. Im Rahmen dieser Arbeit wurden die Konzepte des QVT- und des TGG-Ansatzes untersucht und miteinander verglichen. Dabei wurden sehr viele Übereinstimmungen festgestellt, so dass eine Abbildung zwischen QVT und TGG definiert werden konnte [Gre06]. Auf dieser Grundlage können QVT-Werkzeuge zur Modelltransformation auf Basis von TGGs realisiert werden.

Kapitel 5

Synchronisationsmechanismus

In den vorangegangenen Kapiteln haben wir die Spezifikation von Korrespondenzregeln vorgestellt. In diesem Kapitel gehen wir nun der Frage nach, wie wir auf Grundlage dieser Spezifikation eine inkrementelle Modellsynchronisation durchführen können. Hierzu geben wir zunächst in Abschnitt 5.1 einen Überblick über unseren Synchronisationsmechanismus, der aus einem veränderlichen und einem unveränderlichen Anteil mit einer gemeinsamen Datenstruktur besteht. Der invariante Anteil unseres Synchronisationsmechanismus wird durch einen Algorithmus repräsentiert, der eine Steuerungslogik für die operationalen Graphersetzungsregeln implementiert. Diesen Algorithmus stellen wir zusammen mit der zugrundeliegenden Datenstruktur in Abschnitt 5.2 vor. Der veränderliche Anteil hingegen wird durch operationale Graphersetzungsregeln repräsentiert, die automatisch aus den deklarativ spezifizierten Korrespondenzregeln abgeleitet und zur Parametrisierung des invarianten Anteils eingesetzt werden. Mit der Generierung der operationalen Graphersetzungsregeln beschäftigen wir uns in Abschnitt 5.3. Die Ergebnisse dieses Kapitels fassen wir in Abschnitt 5.4 zusammen.

5.1 Überblick

Der von Andy Schürr in [Sch94] veröffentlichte Algorithmus zur Modelltransformation basiert auf der Idee, ein Modell zunächst zu parsen, um den Ableitungsbaum, d. h., die angewandten Produktionsregeln, durch die dieses Modell erzeugt wurde, zu gewinnen. Anschließend wird durch die Anwendung der dazu korrespondierenden Produktionsregeln, die durch die TGG spezifiziert sind, das Zielmodell erzeugt. Der in [Sch94] veröffentlichte Algorithmus verdeutlicht auf eine sehr elegante Art und Weise das Prinzip, mit dem Modelltransformationen auf der Grundlage von TGGs ausgeführt werden können.

Allerdings wird der Algorithmus in dieser Form in der Praxis nicht eingesetzt. Dies liegt daran, dass es sich bei einer TGG um eine kontextsensitive Grammatik handelt – ein effizienter Algorithmus für das Parsen von Modellen, denen eine kontextsensitive Grammatik zugrunde liegt, existiert bislang nicht. Aus diesem Grund verwenden alle bisher veröffentlichten Algorithmen einen Ansatz, bei dem operationale Graphersetzungsregeln mit einer individuellen Steuerungslogik kombiniert werden [SK08].

Bei diesen Algorithmen traversiert die Steuerungslogik die Elemente eines Quellmodells nach einer zuvor festgelegten Strategie. Dabei wird nach möglichen Anwendungsstellen für die operationalen Graphersetzungsregeln gesucht. Ist eine solche Anwendungsstelle im Quellmodell gefunden, so wird versucht, die operationalen Graphersetzungsregeln anzuwenden. Durch die Anwendung der operationalen Graphersetzungsregeln entsteht ein zum Quellmodell korrespondierendes Zielmodell. Darüber hinaus wird ein Korrespondenzmodell erzeugt, das die Elemente der beiden Modelle mit Hilfe der Korrespondenzknoten zueinander in Beziehung setzt.

Die Strategie zur Traversierung der Modelle ist bei diesen Algorithmen unabhängig von den spezifizierten Regeln. Die Anwendung der operationalen Graphersetzungsregel scheitert daher häufig daran, dass die von einer Regel geforderten Korrespondenzknoten (noch) nicht existieren. Die Anwendung der betrachteten Graphersetzungsregel muss in einem solchen Fall zurückgestellt und die Anwendungsstelle erneut überprüft werden, sobald durch die Anwendung anderer Regeln neue Korrespondenzknoten hinzugekommen sind. Das wiederholte Überprüfen einer Anwendungsstelle wirkt sich negativ auf das Laufzeitverhalten dieser Algorithmen aus.

Der in dieser Arbeit entwickelte Algorithmus basiert ebenfalls auf operationalen Graphersetzungsregeln und einer Steuerungslogik. Im Gegensatz zu den bereits bekannten Algorithmen arbeitet unsere Steuerungslogik allerdings nicht auf dem Quellmodell sondern auf dem Korrespondenzmodell, so dass nur bereits existierende Korrespondenzknoten besucht und die Anwendung der operationalen Graphersetzungsregeln von diesen Korrespondenzknoten ausgehend überprüft wird. Ist eine Regel auf dem untersuchten Korrespondenzknoten nicht anwendbar, so kann dies auf keinen Fall an einem fehlenden Korrespondenzknoten[1] liegen. Eine erneute Überprüfung der betrachteten Regel an dieser Anwendungsstelle ist daher im weiteren Ablauf unseres Algorithmus nicht notwendig. Dieser Umstand führt zu einem günstigeren Laufzeitverhalten unseres Synchronisationsmechanismus.

[1] Eine Ausnahmesituation und die dazugehörige Lösung erläutern wir in Abschnitt 5.3.

Ein weiterer Vorteil unseres Synchronisationsmechanismus besteht darin, dass auf der Grundlage eines erweiterten Korrespondenzmodells eine Modellsynchronisation sowohl batch-artig, d. h., in einem einzigen Schritt, als auch inkrementell, d. h., Schritt für Schritt, durchgeführt werden kann. Die hierzu notwendige Erweiterung am Korrespondenzmodell beruht auf der folgenden Beobachtung (vgl. Abschnitt 3.2): Bei jeder erfolgreichen Regelanwendung wird mindestens ein Korrespondenzknoten gebunden und mindestens ein neuer Korrespondenzknoten erzeugt. Diese Beobachtung nutzen wir in unserem Algorithmus aus, indem wir diese Abhängigkeit zwischen den Korrespondenzknoten explizit durch einen gerichteten Link repräsentieren und während einer Modelltransformation, Modellintegration und Modellsynchronisation ein Korrespondenzmodell aufbauen, das als gerichteter azyklischer Graph (DAG[2]) interpretiert werden kann.

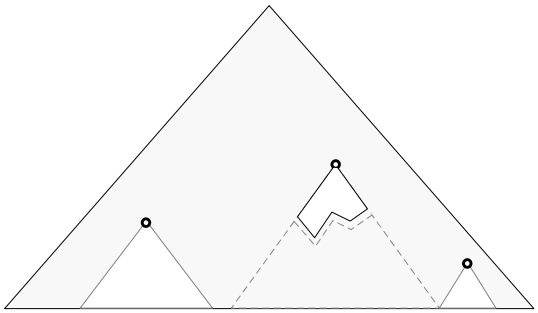


Abbildung 5.1: Prinzip der inkrementellen Modellsynchronisation auf einem Korrespondenzmodell

In Abbildung 5.1 ist ein solches Korrespondenzmodell schematisch dargestellt. Die schematische Darstellung haben wir insofern vereinfacht, als das wir das Korrespondenzmodell nicht durch einen DAG sondern durch eine Baumstruktur repräsentieren. Das Korrespondenzmodell ist allerdings dennoch ein Graph und kein Baum, da jeder Korrespondenzknoten mehrere Korrespondenzknoten als Vorgänger besitzen kann. Der Graph ist azyklisch, weil während der Anwendung einer Regel niemals ein Link zwischen bereits gebundenen Korrespondenzknoten erzeugt wird – es werden immer nur gerichtete Links von den gebundenen zu den neu erzeugten Korrespondenzknoten erstellt.

Bei einer initialen Modellsynchronisation wird das Korrespondenzmodell durch eine Modelltransformation oder – sofern bereits beide Modelle exis-

[2]Abkürzung für 'Directed Acyclic Graph'

tieren – durch eine Modellintegration aufgebaut. Hierbei wird der Algorithmus auf dem Korrespondenzknoten gestartet, der durch das Axiom vorgegeben ist. Dieser Korrespondenzknoten stellt den Einstiegspunkt in den DAG dar. Im Folgenden bezeichnen wir diesen Korrespondenzknoten – in Anlehnung an die in Abbildung 5.1 dargestellte Baumstruktur – als Wurzel unseres Korrespondenzmodells. Ausgehend von der Wurzel werden alle Regeln überprüft und – sofern sie anwendbar sind – ausgeführt. Durch die erfolgreiche Ausführung von Regeln entstehen neue Korrespondenzknoten, auf denen wiederum Regeln ausgeführt werden. Dieser Vorgang wird fortgesetzt, so dass auf diese Art und Weise ein Korrespondenzmodell entsteht, das wir sowohl zu batch-artigen als auch zur inkrementellen Modellsynchronisation nutzen können.

Die batch-artige Modellsynchronisation wird durchgeführt, indem der Synchronisationsalgorithmus auf der Wurzel gestartet wird. In diesem Fall besucht der Algorithmus alle Korrespondenzknoten unseres DAG und versucht – ausgehend von dem aktuell besuchten Korrespondenzknoten – die operationalen Graphersetzungsregeln anzuwenden. Eine inkrementelle Modellsynchronisation hingegen erreichen wir, indem wir zunächst den Korrespondenzknoten ermitteln, der mit einem geänderten Modellelement im Zusammenhang steht. Die Modellsynchronisation beginnt dann nicht auf der Wurzel des Korrespondenzmodells, sondern auf dem zuvor identifizierten Korrespondenzknoten. In diesem Fall werden nur die direkten und indirekten Nachfolger dieses Korrespondenzknotens besucht, was in der schematischen Darstellung der Abbildung 5.1 durch die weiß unterlegten Teilbäume innerhalb der Baumstruktur angedeutet ist. In dem Fall, dass sich Änderungen nur lokal beziehungsweise nur bis zu einer bestimmten Tiefe eines Teilbaums auswirken, kann die Modellsynchronisation – wie im mittleren Teilbaum der Abbildung 5.1 durch die gestrichelten Linien angedeutet wird – sogar früher abgeschlossen werden.

Der hier nur überblicksartig aufgezeigte Synchronisationsmechanismus wird in den nachfolgenden Abschnitten genauer vorgestellt. Hierzu beschäftigen wir uns zunächst in Abschnitt 5.2 mit der zugrundeliegenden Datenstruktur sowie dem darauf basierenden Synchronisationsalgorithmus. Dieser Synchronisationsalgorithmus repräsentiert die Steuerungslogik für operationale Graphersetzungsregeln, die aus TGG-Regeln generiert werden. Mit der Generierung der operationalen Graphersetzungsregeln beschäftigen wir uns erst in dem darauf folgenden Abschnitt 5.3.

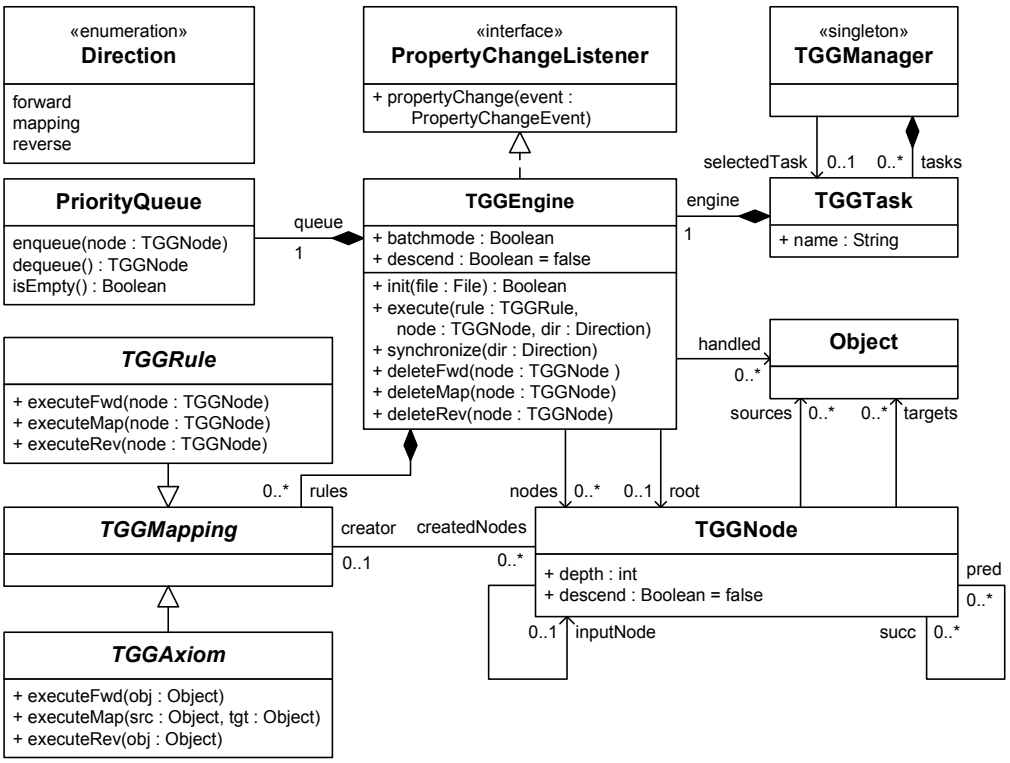


Abbildung 5.2: Datenstruktur

5.2 Datenstruktur und Algorithmus

Die Datenstruktur wird sowohl in den operationalen Graphersetzungsregeln als auch in der Steuerungslogik, d. h., dem invarianten Anteil unseres Synchronisationsmechanismus, eingesetzt.

5.2.1 Datenstruktur

Das Klassendiagramm der Datenstruktur ist in Abbildung 5.2 zu sehen. Ein zentraler Bestandteil dieses Klassendiagramms ist die Klasse `TGGEngine`, die den Synchronisationsalgorithmus repräsentiert. Darüber hinaus enthält das Klassendiagramm Klassen, mit denen unterschiedliche Modelltransformations-, Modellintegrations- sowie Modellsynchronisationsaufgaben verwaltet werden, Klassen zur Repräsentation des Korrespondenzmodells und der operationalen Graphersetzungsregeln sowie Klassen, die zur effizienten Traversierung des Korrespondenzmodells beitragen.

Für die Verwaltung der unterschiedlichen Aufgaben (Modelltransformation, Modellintegration und Modellsynchronisation) sind die Klassen `TGGManager` und `TGGTask` verantwortlich. Die Klasse `TGGManager` implementiert das *Singleton-Entwurfsmuster* [GHJV94], so dass zur Laufzeit immer nur eine Instanz dieser Klasse existiert. Diese Instanz kennt alle vom Benutzer initiierten Aufgaben.

Die Aufgaben werden durch Instanzen der Klasse `TGGTask` repräsentiert. Damit ein Benutzer zwischen den unterschiedlichen Aufgaben besser differenzieren kann, können die Aufgaben benannt werden. Jeder Aufgabe ist eine eigene Instanz der Klasse `TGGEngine` zugeordnet, die durch einen Aufruf der Methode `init` mit den für diese Aufgabe vorgesehenen Graphersetzungsregeln initialisiert wird. Hierzu muss der Methode eine Datei[3] mit den ausführbaren Graphersetzungsregeln übergeben werden.

Nach der Initialisierung können die operationalen Graphersetzungsregeln über die Assoziation `rules` erreicht werden. Die operationalen Graphersetzungsregeln werden in Axiome und Regeln unterteilt. Hierzu erben die Klassen `TGGAxiom` und `TGGRule` von der Klasse `TGGMapping`. Diese Unterteilung ist notwendig, weil im Gegensatz zu den Graphersetzungsregeln einer TGG-Regel den Graphersetzungsregeln eines TGG-Axioms kein Korrespondenzknoten als Parameter übergeben werden kann. Dies liegt daran, dass bei der initialen Zuordnung der Modelle noch gar kein Korrespondenzknoten existiert – ein erster Korrespondenzknoten wird erst durch das Axiom selbst erzeugt. Für den Synchronisationsalgorithmus ist dieser Korrespondenzknoten daher erst nach der Initialisierung über die Assoziation `root` erreichbar.

Bei den Klassen `TGGMapping`, `TGGRule` und `TGGAxiom` handelt es sich um abstrakte Klassen. Die konkreten Klassen, die bei einer Modellsynchronisation zum Einsatz kommen, werden aus der Spezifikation automatisch generiert, kompiliert und in der zuvor erwähnten Datei gespeichert. Dabei müssen die erzeugten Klassen entweder von der Klasse `TGGRule` oder der Klasse `TGGAxiom` erben und die vererbten abstrakten Methoden implementieren.[4] Erst diese Klassen – zusammen mit den darin implementierten Methoden – ermöglichen unserem Synchronisationsalgorithmus den Zugriff auf die operationalen Graphersetzungsregeln, mit deren Hilfe unter anderem das Korrespondenzmodell aufgebaut wird.

[3] Bei der Datei handelt es sich um ein Jar-Archiv mit kompilierten und somit ausführbaren Klassen. Diese Klassen repräsentieren die operationalen Graphersetzungsregeln.

[4] Mit der Generierung dieser Klassen werden wir uns noch in Abschnitt 5.3 genauer beschäftigen.

Die Korrespondenzknoten eines Korrespondenzmodells werden durch Instanzen der Klasse `TGGNode` repräsentiert. Alle Korrespondenzknoten, die im Metamodell für das Korrespondenzmodell spezifiziert werden, müssen von dieser Klasse erben. Die Abhängigkeiten zwischen den Korrespondenzknoten werden über die Assoziation `succ` realisiert. Bei dieser Assoziation handelt es sich um eine bidirektionale Assoziation, so dass jeder Korrespondenzknoten seine Vorgänger über `pred` erreichen kann. Ein Beispiel für eine TGG-Regel, in der ein Korrespondenzknoten mehrere Korrespondenzknoten als Vorgänger besitzt, haben wir in Abbildung 3.8 kennen gelernt (siehe Seite 66).

Jeder Korrespondenzknoten merkt sich zusätzlich über die Referenz `inputNode`, welcher Korrespondenzknoten der Graphersetzungsregel als Parameter übergeben wurde, d. h., auf welchem Korrespondenzknoten die Anwendung der operationalen Graphersetzungsregel gestartet worden war. Die Graphersetzungsregel, die den Korrespondenzknoten erzeugt hat, ist hingegen über die Assoziation `creator` erreichbar. Da es sich hierbei ebenfalls um eine bidirektionale Assoziation handelt, kann jede Graphersetzungsregel über `createdNodes` alle Korrespondenzknoten erreichen, die sie erzeugt hat.

Der Zugriff auf das Korrespondenzmodell ist über die Assoziation `root` der Klasse `TGGEngine` möglich. Zusätzlich werden alle neu erzeugten Korrespondenzknoten durch die Assoziation `nodes` mit der Instanz der Klasse `TGGEngine` verlinkt. Diese Assoziation wird für einen schnelleren Zugriff auf die Korrespondenzknoten des Korrespondenzmodells verwendet. Zur Modellsynchronisation verwendet jede Instanz der Klasse `TGGEngine` eine Prioritätswarteschlange, die durch die Klasse `PriorityQueue` repräsentiert wird. Die Prioritätswarteschlange ist durch die Assoziation `queue` erreichbar. Sie verwaltet alle Korrespondenzknoten, die nach Modelländerungen wieder überprüft werden müssen.

Bei der Anwendung einer Graphersetzungsregel muss sichergestellt werden, dass es sich bei den neu zu bindenden Modellelementen um Elemente handelt, die noch nicht an einer Beziehung beteiligt sind (siehe auch Abschnitt 5.3). Dies wird über die Assoziation `handled` sichergestellt. Während der Anwendung einer Graphersetzungsregel wird mit Hilfe dieser Assoziation überprüft, ob das neu gebundene Objekt bereits verlinkt und damit verbraucht ist. In dem Fall, dass das Objekt noch nicht verbraucht ist, wird die Ausführung der Graphersetzungsregel fortgesetzt und das Objekt anschließend über die Assoziation `handled` als verbraucht markiert. Ansonsten wird die Ausführung der betrachteten Graphersetzungsregel mit dem bereits verbrauchten Objekt nicht weiter verfolgt.

Um die Objekte der beteiligten Modelle zueinander in Beziehung setzen zu können, besitzt ein Korrespondenzknoten die Assoziationen `sources` und `targets`, die Instanzen vom Typ `Object` referenzieren. Daher müssen alle Metaklassen in den beteiligten Metamodellen – mit Ausnahme des Korrespondenzmetamodells – explizit oder implizit von dieser Klasse erben.[5]

5.2.2 Algorithmus

Der Algorithmus zur Modellsynchronisation ist in der Methode `synchronize` der Klasse `TGGEngine` implementiert, die in der Abbildung 5.3 in Pseudocode-Syntax zu sehen ist. Der Algorithmus ist für alle Regeln gleich, so dass weder eine zusätzliche Spezifikation der Ausführungsreihenfolge noch eine Programmierung der Steuerungslogik in Abhängigkeit der spezifizierten Regeln notwendig ist.

```
 1: TGGEngine::synchronize(dir:Direction)
 2:   let node : TGGNode := null;
 3:   while (not self.queue->isEmpty()) do
 4:     node := self.queue->dequeue();
 5:     for each (rule : TGGRule in self.rules) do
 6:       self->execute(rule, node, dir);
 7:     if (self.batchmode = true) then
 8:       for each (child : TGGNode in node.succ) do
 9:         self.queue->enqueue(child);
10:     else if (self.descend = true) then
11:       for each (child : TGGNode in node.succ) do
12:         if (child.descend = true) then do
13:           self.queue->enqueue(child);
14:           child.descend := false;
15:       self.descend := false;
```

Abbildung 5.3: Die Methode `synchronize` der Klasse `TGGEngine`

[5]Die prototypische Realisierung basiert auf der Programmiersprache Java. Darin erben alle Klassen implizit von der Klasse `Object`. Bei einer Umsetzung in einer anderen Programmiersprache muss an dieser Stelle eine Anpassung an die dort verfügbaren Konzepte erfolgen. Um Referenzen auf beliebige Objekte beispielsweise in der Programmiersprache C++ zu realisieren, könnte dort auf anonyme Zeiger (void* ptr;) zurückgegriffen werden.

Beim Aufruf der Methode wird die Richtung der Synchronisation der Methode als Parameter (Zeile 1) übergeben. Als Werte können hier `forward`, `mapping` oder `reverse` der Aufzählung `Direction` verwendet werden. Damit eine Modellsynchronisation durchgeführt wird, muss vor einem Aufruf die Prioritätswarteschlange mindestens einen Korrespondenzknoten enthalten, so dass die nachfolgende Schleife (Zeile 3) mindestens einmal durchlaufen wird. Ist die Warteschlange beim Aufruf der Methode hingegen leer, terminiert die Modellsynchronisation ohne die Modelle miteinander zu synchronisieren.

Für eine batch-artige Modellsynchronisation muss die Warteschlange mit der Wurzel des Korrespondenzmodells initialisiert und das Attribut `batchmode` auf den Wert `true` gesetzt werden. Im Falle einer inkrementellen Modellsynchronisation ist das Attribut `batchmode` auf den Wert `false` zu setzen. Darüber hinaus müssen für eine inkrementelle Modellsynchronisation in der Warteschlange alle Korrespondenzknoten gespeichert sein, die mit den geänderten Modellelementen in Beziehung stehen und erneut überprüft werden sollen.[6]

Innerhalb der Schleife wird zunächst ein Korrespondenzknoten aus der Prioritätswarteschlange entnommen (Zeile 4). Anschließend wird für jede Regel (Zeile 5) die Methode `execute` aufgerufen, die neben der Regel den Korrespondenzknoten und die Synchronisationsrichtung als Parameter erhält (Zeile 6). In dieser Methode wird – abhängig von der Synchronisationsrichtung – eine entsprechende Graphersetzungsregel ausgeführt (siehe Abbildung 5.4; vgl. auch Abschnitt 5.3). Sind alle Regeln behandelt worden, so wird die Schleife (Zeilen 5 und 6) verlassen. Anschließend wird überprüft, ob ein weiterer Abstieg in das Korrespondenzmodell notwendig ist (Zeilen 7 und 10).

Bei einer batch-artigen Modellsynchronisation (Zeile 7) müssen alle Korrespondenzknoten des Korrespondenzmodells besucht werden, so dass alle direkten Nachfolger des gerade betrachteten Korrespondenzknotens zu der Warteschlange hinzugefügt werden (Zeilen 8 und 9). Bei einer inkrementellen Modellsynchronisation hingegen steuert das Attribut `descend` der Klasse `TGGEngine` den Abstieg in das Korrespondenzmodell (Zeile 10). Dazu setzt jede Graphersetzungsregel dieses Attribut auf den Wert `true`, sobald sie Änderungen an den Modellen vorgenommen hat.

[6] Eine inkrementelle Modellsynchronisation, die initial ausgeführt wird – also einer Modelltransformation oder Modellintegration entspricht – kann nur auf der Wurzel des Korrespondenzmodells gestartet werden. In diesem Fall verhält sich der Algorithmus genauso wie bei einer batch-artigen Modellsynchronisation.

```
 1: TGGEngine::execute(rule:TGGRule, node:TGGNode,
                       dir:Direction)
 2:   switch (dir)
 3:     case forward:
 4:       rule->executeFwd(node);
 5:       break;
 6:     case mapping:
 7:       rule->executeMap(node);
 8:       break;
 9:     case reverse:
10:       rule->executeRev(node);
11:       break;
12:     default:
13:       rule->executeMap(node);
```

Abbildung 5.4: Die Methode `execute` der Klasse `TGGEngine`

Damit nicht alle Nachfolger des betrachteten Korrespondenzknotens hinzugefügt werden, markiert jede Graphersetzungsregel zusätzlich die zu überprüfenden Nachfolger, indem sie auch dort das Attribut `descend` auf den Wert `true` setzt. Der Wert dieses Attributs wird innerhalb der Schleife (Zeile 11) überprüft (Zeile 12), so dass nur die tatsächlich von Änderungen betroffenen Nachfolger zur Warteschlange hinzugefügt werden (Zeile 13).

In dem Fall, dass ein Nachfolger des betrachteten Korrespondenzknotens zur Warteschlange hinzugefügt worden ist, wird das Attribut `descend` dieses Nachfolgers wieder auf den Wert `false` gesetzt (Zeile 14). Nachdem alle Nachfolger abgearbeitet worden sind, wird darüber hinaus das Attribut `descend` für den Synchronisationsalgorithmus selbst wieder auf den Wert `false` gesetzt (Zeile 15).[7]

Unabhängig davon, ob eine batch-artige oder eine inkrementelle Modellsynchronisation durchgeführt wird, terminiert der Algorithmus, sobald alle Korrespondenzknoten der Prioritätswarteschlange betrachtet worden sind, d. h., die Prioritätswarteschlange leer ist (Zeile 3).

[7] Natürlich könnte auf dieses Attribut verzichtet werden und immer alle Nachfolger eines Korrespondenzknoten überprüft werden. In den Fällen, in denen gar keine Änderungen durch Graphersetzungsregeln vorgenommen worden sind, wäre dies jedoch ineffizient.

Prioritätswarteschlage

Die Art der in unserem Algorithmus eingesetzten Warteschlange beeinflusst die Reihenfolge, in der die Korrespondenzknoten während einer Modellsynchronisation besucht werden. Beispielsweise könnten wir eine Warteschlange verwenden, die nach dem FIFO[8]-Prinzip arbeitet. Diese Datenstruktur ist zwar sowohl für die batch-artige als auch für die inkrementelle Modellsynchronisation geeignet, allerdings kann es bei einer inkrementellen Modellsynchronisation die Laufzeit des Algorithmus negativ beeinflussen.

Das Problem wird in Abbildung 5.5 anhand der schematischen Baumstruktur unseres Korrespondenzmodells dargestellt. Das Problem taucht auf, sofern sich die zu synchronisierenden Teilbäume überschneiden oder – wie in Abbildung 5.5 gezeigt – ein Teilbaum vollständig innerhalb eines anderen Teilbaums liegt. In der Abbildung sind die beiden Korrespondenzknoten mit `a` und `b` gekennzeichnet. Die dazugehörigen Teilbäume wollen wir mit t_a und t_b benennen. In diesem Beispiel gehen wir davon aus, dass die Änderungen in den Modellen so beschaffen sind, dass auch bei der inkrementellen Modellsynchronisation beide Teilbäume vollständig traversiert werden müssen. Außerdem liegt unserem Beispiel die Annahme zugrunde, dass nicht sofort nach einer Modelländerung synchronisiert wird, sondern die Modelländerungen akkumuliert und erst auf Anforderung durch den Benutzer überprüft werden.

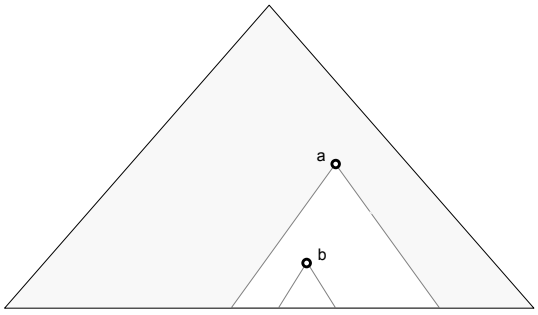


Abbildung 5.5: Zu überprüfende Korrespondenzknoten und ihre Teilbäume im Korrespondenzmodell

Zur inkrementellen Modellsynchronisation wird die Warteschlange mit Korrespondenzknoten gefüllt, die Korrespondenzbeziehungen zwischen geänderten Modellelementen repräsentieren. In unserem Beispiel nehmen

[8]First In – First Out

wir an, dass der Korrespondenzknoten `b` vor dem Korrespondenzknoten `a` in die Warteschlange eingefügt wird und dann der Synchronisationsalgorithmus gestartet wird. In diesem Fall würde im Rahmen der inkrementellen Modellsynchronisation zunächst der Korrespondenzknoten `b` der Warteschlange entnommen, mit Hilfe der Graphersetzungsregeln untersucht und dessen Nachfolger in die Warteschlange eingefügt werden. Anschließend würde der Algorithmus den Korrespondenzknoten `a` aus der Warteschlage entnehmen und damit in gleicher Weise verfahren. Der Algorithmus würde somit abwechselnd Korrespondenzknoten der Teilbäume t_b und t_a überprüfen. Insgesamt würde der Teilbaum t_b jedoch vor dem Teilbaum t_a abgearbeitet werden.[9] Dabei sind zwei Fälle zu unterscheiden:

Fall 1 Die Synchronisation der Vorgänger von `b` im Teilbaum t_a hat keinen Einfluss auf die durch `b` hergestellte Korrespondenzbeziehung. Weil diese Korrespondenzbeziehung bereits bei der Synchronisation des Teilbaums t_b überprüft und gegebenenfalls wieder hergestellt wurde, würden im Rahmen der Synchronisation von Teilbaum t_a beim Erreichen von `b` dessen Nachfolger nicht mehr in die Warteschlange eingefügt werden. Somit würde in diesem Fall der Teilbaum t_b nicht noch einmal überprüft werden.

Fall 2 Während der Synchronisation des Teilbaums t_a führt die Synchronisation der Vorgänger von `b` dazu, dass die Korrespondenzbeziehung des bereits zuvor synchronisierten Korrespondenzknoten `b` nicht mehr gültig ist. Daher müssten und würden in diesem Fall die Nachfolger von `b` wieder in die Warteschlange eingefügt werden, d. h., der Teilbaum t_b würde ein zweites Mal traversiert und überprüft werden.

Um den doppelten Aufwand, der durch den zweiten Fall entsteht, zu vermeiden, verwenden wir statt einer FIFO-Warteschlange eine Prioritätswarteschlange. In der Prioritätswarteschlange werden alle Korrespondenzknoten beim Einfügen nach ihrer Tiefe im Korrespondenzmodell einsortiert. Hierzu besitzt die Klasse `TGGNode` das Attribut `depth`, dessen Wert sich aus der maximalen Tiefe seiner direkten Vorgänger errechnet und um eins erhöht wird. Die Tiefe der Wurzel, d. h., des Korrespondenzknotens aus dem Axiom, beträgt immer null.

[9] Dies trifft auch dann zu, wenn die Korrespondenzknoten in umgekehrter Reihenfolge in die Warteschlange eingefügt worden wären.

In der Prioritätswarteschlange besitzen Korrespondenzknoten mit einer geringeren Tiefe eine höhere Priorität als tiefer angesiedelte Korrespondenzknoten. Daher beginnt die Synchronisation immer mit den Korrespondenzknoten, die der Wurzel am nächsten sind. Eine zusätzliche Sortierung der Korrespondenzknoten mit derselben Tiefe ist nicht notwendig, weil diese Korrespondenzknoten sich nicht gegenseitig beeinflussen.

Die Korrespondenzknoten werden auch beim mehrmaligen Einfügen nur einmal in der Prioritätswarteschlange gespeichert. In unserem Beispiel würde somit beim Synchronisieren von Teilbaum t_a der Korrespondenzknoten `b` nur einmal in der Warteschlange gespeichert werden. Bei der Ausführung der Synchronisation würde dieser Knoten erst dann überprüft werden, wenn alle Korrespondenzknoten geringerer Tiefe abgearbeitet wurden.

Durch den Einsatz der Prioritätswarteschlange und der damit verbundenen Priorisierung der Korrespondenzknoten erfolgt die Traversierung des Korrespondenzmodells durch eine Breitensuche über die zu synchronisierenden Teilbäume. Diese Vorgehensweise verhindert die mehrmalige Überprüfung von Korrespondenzknoten und funktioniert insbesondere auch dann, wenn die Teilbäume sich nur teilweise überlappen.

Ermittlung der Anwendungsstelle

Für eine batch-artige Modellsynchronisation muss nur die Wurzel des Korrespondenzmodells in die Prioritätswarteschlange eingefügt werden. Der Algorithmus sorgt anschließend dafür, dass alle Korrespondenzknoten des Korrespondenzmodells traversiert und dabei die Modelle miteinander abgeglichen werden. Eine inkrementelle Modellsynchronisation hingegen erreichen wir, indem wir nur die von Änderungen betroffenen Modellelemente inspizieren. Dazu muss die Prioritätswarteschlange mit den zu überprüfenden Korrespondenzknoten initialisiert werden, d. h., dass zunächst die potentiellen Anwendungsstellen identifiziert werden müssen.

Zur Identifizierung dieser Anwendungsstellen wird in unserem Ansatz ein Benachrichtigungsmechanismus auf Grundlage des *Observer-Entwurfsmusters* [GHJV94] eingesetzt. Dieser Benachrichtigungsmechanismus sorgt dafür, dass Änderungen im Modell an einen Beobachter gemeldet werden, der darauf geeignet reagieren kann. Der Beobachter wird durch die Klasse `TGGEngine` realisiert. Diese Klasse implementiert die in der Schnittstelle `PropertyChangeListener` definierte Methode `propertyChange`, die dafür sorgt, dass die mit dem gemeldeten Modellelement in Beziehung stehenden Korrespondenzknoten in die Prioritätswarteschlange eingefügt wer-

den. Hierzu extrahiert die Methode zunächst aus dem ihr übergebenen Parameter `evt` das geänderte Modellelement. Anschließend wird der Korrespondenzknoten identifiziert, der mit diesem Modellelement über eine `source`- oder `targets`-Assoziation verlinkt ist. Da dieser Korrespondenzknoten auf der Grundlage seiner Vorgänger erstellt wurde, werden sowohl seine Vorgänger als auch der Korrespondenzknoten selbst zur Prioritätswarteschlange hinzugefügt.

Wird ein solcher Benachrichtigungsmechanismus von den beteiligten Modellen unterstützt, so kann mit unserem Ansatz eine Modellsynchronisation inkrementell durchgeführt werden. Ist ein solcher Benachrichtigungsmechanismus jedoch nicht vorhanden, so kann die Modellsynchronisation nur batch-artig ausgeführt werden.

5.3 Generierung operationaler Graphersetzungsregeln

Im vorangegangenen Abschnitt haben wir den unveränderlichen Anteil unseres Synchronisationsmechanismus kennen gelernt. In diesem Abschnitt stellen wir den veränderlichen Anteil vor. Dieser Anteil wird durch operationale Graphersetzungsregeln repräsentiert, die automatisch aus den spezifizierten TGG-Regeln generiert werden. Zusammen mit der im vorangegangenen Abschnitt vorgestellten Steuerungslogik bestimmen die operationalen Graphersetzungsregeln letztendlich, wie eine Modelltransformation, Modellintegration oder Modellsynchronisation durchgeführt wird. Die automatische Generierung der operationalen Regeln aus einer deklarativen Spezifikation ermöglicht uns somit eine Parametrisierung unseres Synchronisationsmechanismus.

5.3.1 Prinzip

Die Grundstruktur einer TGG-Regel ist in Abbildung 5.6 dargestellt. Wie anhand der farblich unterlegten Bereiche zu sehen ist, kann eine TGG-Regel grundsätzlich in bereits gebundene und neu zu erzeugende Elemente unterteilt werden. Darüber hinaus kann eine weitere Einteilung in die durch die Regel in Beziehung gesetzten Modelle (Modell A, Modell B und Korrespondenzmodell) erfolgen. Diese Einteilung wird in Abbildung 5.6 durch die vertikalen Linien zwischen den drei Modellen verdeutlicht.

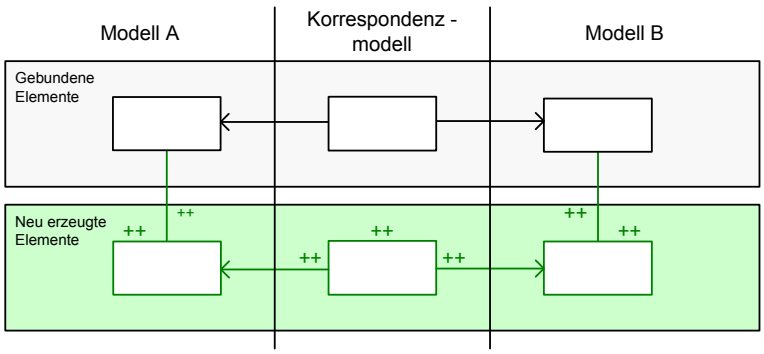
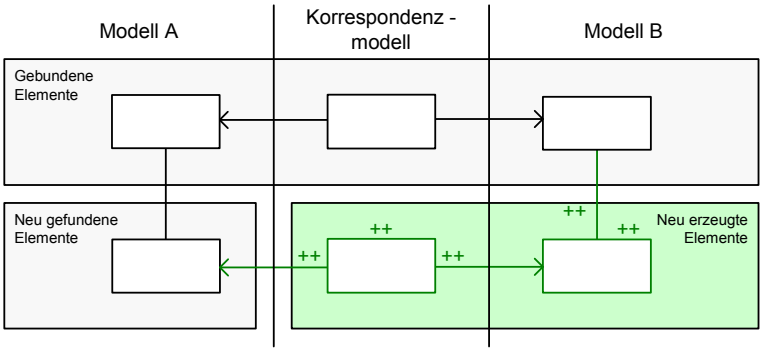


Abbildung 5.6: Grundstruktur einer TGG-Regel

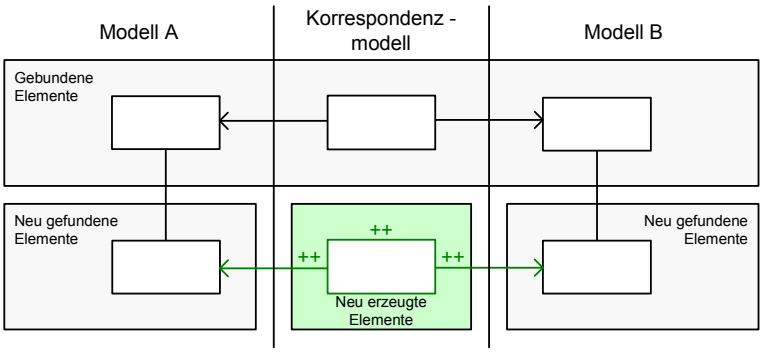
In dem Beitrag von Andy Schürr [Sch94] wurde die Ableitung von insgesamt drei operationalen Graphersetzungsregeln aus einer TGG-Regel vorgeschlagen. Die drei Graphersetzungsregeln sind in Abbildung 5.7 schematisch dargestellt. Mit Hilfe der abgeleiteten Graphersetzungsregeln lässt sich sowohl eine Modelltransformation in Vorwärtsrichtung (Modell A nach Modell B) als auch eine Modelltransformation in Rückwärtsrichtung (Modell B nach Modell A) realisieren. Darüber hinaus kann mit der dritten Graphersetzungsregel eine Modellintegration durchgeführt werden.

Die Regel aus Abbildung 5.7(a) wird zur Transformation eines Modells A in ein Modell B eingesetzt. Bei der Anwendung dieser Regel werden daher zunächst alle Elemente der Regel aus dem oberen, grau schattierten Bereich gebunden. Anschließend wird überprüft, ob ein Element aus dem unteren grau schattierten Bereich ebenfalls gebunden werden kann. Um die Semantik der TGGs nicht zu verletzen, dürfen dabei nur noch nicht transformierte Elemente berücksichtigt werden. In dem Fall, dass ein solches Element gefunden werden konnte, werden neue Elemente im Modell B erzeugt. Darüber hinaus wird ein neuer Korrespondenzknoten erzeugt, der die Elemente der beiden Modelle miteinander über die spezifizierten Links in Beziehung setzt.

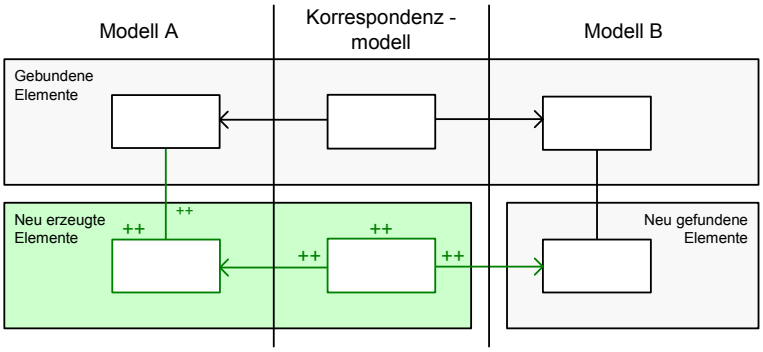
Die Regel aus Abbildung 5.7(b) dient der Modellintegration. Diese Regel funktioniert ähnlich zu der Modelltransformationsregel aus Abbildung 5.7(a). Allerdings werden bei dieser Regel nicht nur Elemente im Modell A sondern auch Elemente im Modell B gesucht. Auch hier dürfen die Elemente noch nicht durch eine Modelltransformation oder eine Modellintegration zueinander in Beziehung gesetzt worden sein. Konnten solche Elemente gefunden werden, so wird in dieser Regel nur noch ein Korrespondenzknoten erzeugt, der diese Elemente über Links miteinander in Beziehung setzt.



(a) Grundstruktur der operationalen Graphersetzungsregel zur Modelltransformation in Vorwärtsrichtung (Modell A nach Modell B)



(b) Grundstruktur der operationalen Graphersetzungsregel zur Modellintegration (Modell A und Modell B)



(c) Grundstruktur der operationalen Graphersetzungsregel zur Modelltransformation in Rückwärtsrichtung (Modell B nach Modell A)

Abbildung 5.7: Grundstruktur der aus einer TGG-Regel abgeleiteten operationalen Graphersetzungsregeln

Die dritte Regel ist in Abbildung 5.7(c) zu sehen. Diese Regel ist eine Umkehrung der in Abbildung 5.7(a) gezeigten Regel und wird zur Transformation eines Modells B in ein Modell A verwendet. Die Ausführung dieser Regel erfolgt analog zur Regel aus Abbildung 5.7(a), nur dass hier bei einer erfolgreichen Anwendung keine Elemente in Modell A sondern in Modell B erzeugt werden.

Der in dieser Arbeit realisierte Modellsynchronisationsmechanismus basiert ebenfalls auf operationalen Graphersetzungsregeln, die vom Prinzip her den in der Abbilung 5.7 gezeigten Regeln entsprechen. Zur inkrementellen Modellsynchronisation werden jedoch komplexere Graphersetzungsschritte benötigt, die durch geeignete Kontrollstrukturen gesteuert werden müssen. Hierzu setzen wir Storydiagramme ein.

5.3.2 Storydiagramme

Bei Storydiagrammen handelt es sich um erweiterte UML-Aktivitätsdiagramme, mit denen das Verhalten einer Methode graphisch spezifiziert werden kann [FNTZ98, Zün01]. Ein Storydiagramm setzt sich aus Aktivitäten zusammen, die über Transitionen miteinander verbunden sind. Die Transitionen steuern den Kontrollfluss innerhalb eines Storydiagramms.

Der Kontrollfluss eines Storydiagramms beginnt bei einer Startaktivität und endet, wenn die Stoppaktivität erreicht wird. Die einzelnen Aktivitäten innerhalb eines Storydiagramms können entweder durch Java-Codefragmente oder durch sogenannte *Story-Patterns* spezifiziert sein. Bei den Story-Patterns handelt es sich um operationale Graphersetzungsregeln. Wie wir schon in Abschnitt 3.1.2 gesehen haben, besteht eine Graphersetzungsregel aus einer linken und einer rechten Regelseite. Während die linke Regelseite eine zu suchende Objektstruktur beschreibt, legt die rechte Regelseite fest, wie diese Objektstruktur modifiziert werden soll, falls sie gefunden wird.

Abbildung 5.8 zeigt ein Beispiel für ein Storydiagramm. Das Storydiagramm besteht aus einer Startaktivität, einer Stoppaktivität, sowie zwei Story-Pattern. Die Nummerierung in der linken oberen Ecke der Aktivität gibt weder eine Ausführungsreihenfolge vor noch ist sie Bestandteil der Spezifikation. Die Nummerierung dient uns in dieser Arbeit lediglich dazu, die Verweise auf einzelne Aktivitäten im Text eindeutig zu machen.

Mit dem Storydiagramm wird das Verhalten der Methode `connect` der Klasse `Block` spezifiziert. Damit kann die Methode auf Instanzen der Klasse `Block` ausgeführt werden. Wie bei Methoden üblich, können Attributwerte und Objekte als Parameter an ein Storydiagramm übergeben werden. Durch

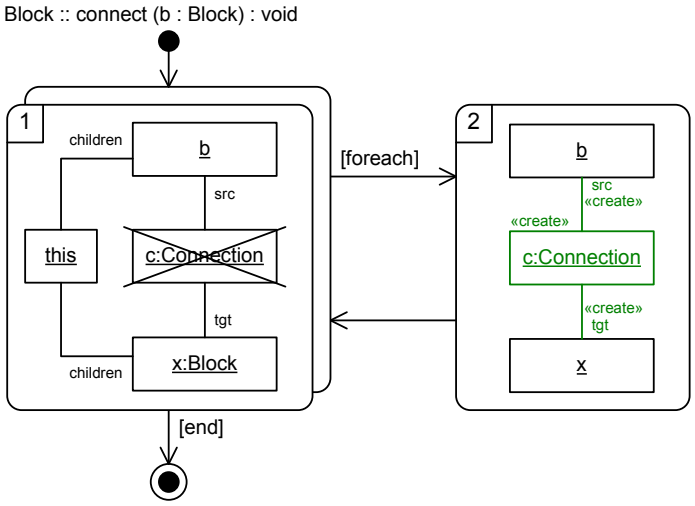


Abbildung 5.8: Beispiel für ein Storydiagramm

die Parameterobjekte werden Teile der Anwendungsstelle für die Graphersetzungsregeln, d. h., die Story-Pattern, vorgegeben. Dies reduziert die Anzahl der möglichen Anwendungsstellen und verringert damit den Aufwand bei der Anwendung der Graphersetzungsregel. In unserem Beispiel wird der Methode ein Parameter `b` vom Typ `Block` übergeben. Ein Rückgabewert wurde nicht spezifiziert.

Die Ausführung der Methode beginnt bei der Startaktivität, die durch einen ausgefüllten Kreis dargestellt ist. Ausgehend von dieser Startaktivität wechselt der Kontrollfluss zum ersten Story-Pattern (Aktivität 1). Dieses Story-Pattern enthält die Objekte `this` und `b`, die beide ohne Typangaben spezifiziert sind. Bei diesen Objekten handelt es sich daher um *bereits gebundene Objekte*, d. h., diese Objekte sind bereits mit konkreten Instanzen belegt. Das Objekt `this` repräsentiert eine Instanz, auf der die Methode aufgerufen wurde, also eine Instanz der Klasse `Block`. Das Objekt `b` repräsentiert den an die Methode übergebenen Parameter.

Ausgehend von den gebundenen Objekten `this` und `b` wird zunächst überprüft, ob ein `children`-Link zwischen diesen beiden Objekten existiert. Anschließend wird versucht, eine Instanz vom Typ `Block` an das Object `x` zu binden. Diese Instanz darf jedoch nicht über ein Objekt `c` vom Typ `Connection` mit dem Objekt `b` verbunden sein, was durch das durchgestrichene Objekt `c:Connection` repräsentiert wird. Bei dieser Bedingung handelt es sich um eine negative Anwendungsbedingung.

Der doppelte Rahmen um die Aktivität 1 bedeutet, dass das darin enthaltene Story-Pattern auf alle Instanzen angewandt wird, die der spezifizierten Graphersetzungsregel entsprechen. Aufgrund der bereits gebundenen Objekte `this` und `b` kann allerdings nur noch das Objekt `x:Block` frei gebunden werden. Für jede gefundene Instanz wird über die mit der Bedingung `foreach` annotierte Transition zur Aktivität 2 gewechselt.

In der Aktivität 2 wird ein Objekt `c:Connection` mit entsprechenden Links zu den Objekten `b` und `x` erzeugt.[10] Anschließend wechselt der Kontrollfluss zurück zur Aktivität 1. Wird kein Objekt `x:Block` mehr gefunden, das die negative Anwendungsbedingung erfüllt, wechselt der Kontrollfluss zur Stoppaktivität, die durch einen doppelten Kreis dargestellt wird. Hier endet die Ausführung der Methode.

Die hier nur in Auszügen dargestellte Syntax und Semantik der Storydiagramme wurde von Albert Zündorf entwickelt, formalisiert und in [Zün01] vorgestellt. Auf der Grundlage dieser Formalisierung kann aus einem Storydiagramm ausführbarer Code generiert werden [FNTZ98]. Diese Eigenschaft nutzen wir in unserem Ansatz aus, um den veränderlichen Anteil unseres Synchronisationsmechanismus automatisch zu generieren.

5.3.3 Generierung

Die Generierung von Storydiagrammen erläutern wir am Beispiel der TGG-Regel *Process2Class*, mit der eine Korrespondenzbeziehung zwischen einem Prozess im Blockdiagramm und einer Klasse im Klassendiagramm beschrieben wird. Diese Regel haben wir bereits in Abbildung 3.7 kennen gelernt (siehe Abschnitt 3.2, Seite 66).

Vor der Generierung der Storydiagramme wird zunächst zu jeder TGG-Regel und zu jedem TGG-Axiom eine eigene Klasse erzeugt. Je nachdem, ob die erzeugte Klasse eine TGG-Regel oder ein TGG-Axiom repräsentiert, erbt die erzeugte Klasse entweder von der Klasse `TGGRule` oder von der Klasse `TGGAxiom` und definiert die dort deklarierten Methoden `executeFwd`, `executeMap` und `executeRev` mit den ensprechenden Methodensignaturen. Aus der dazugehörigen TGG-Regel wird anschließend zu jeder Methode ein

[10] Die Erzeugung dieses Objekts hätte mit gleichbleibender Semantik ebenso in der Aktivität 1 spezifiziert werden können. Die Aufteilung auf zwei Aktivitäten ist in diesem Beispiel erfolgt, um den Kontrollfluss aus einer `foreach`-Aktivität zu demonstrieren. Dieser wird bei der Generierung von Storydiagrammen aus TGG-Regeln an mehreren Stellen eingesetzt.

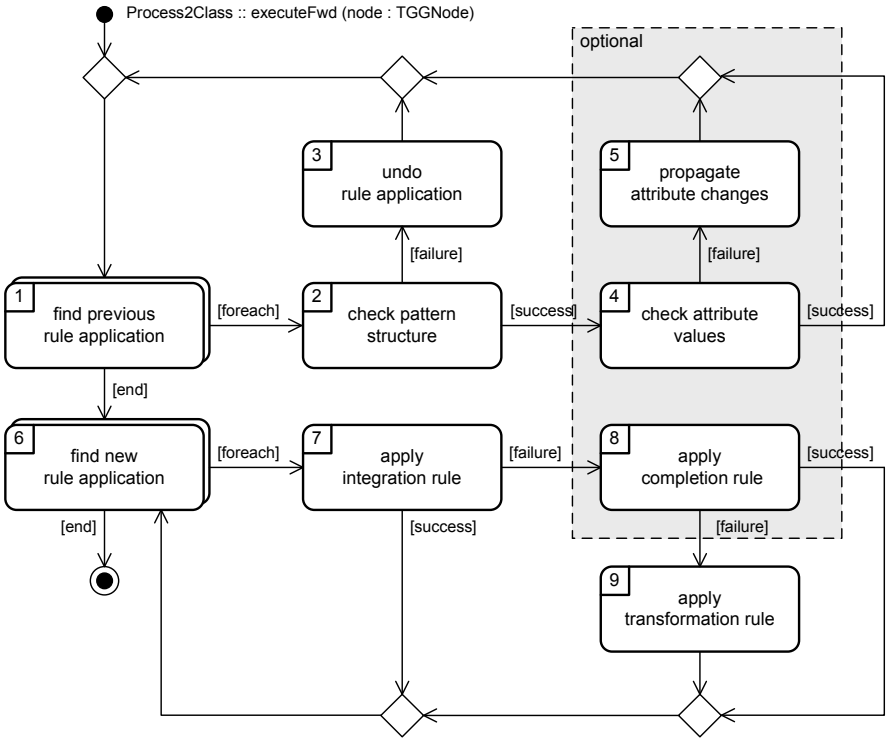


Abbildung 5.9: Grundstruktur der zu generierenden Storydiagramme

Storydiagramm erzeugt. In Abbildung 5.9 ist die Grundstruktur der zu generierenden Storydiagramme dargestellt.

Der Ablauf des in Abbildung 5.9 gezeigten Storydiagramms kann in zwei Phasen unterteilt werden. In der ersten Phase werden bereits etablierter Korrespondenzbeziehungen überprüft und erkannte Inkonsistenzen beseitigt (Aktivitäten 1–5). Die zweite Phase hingegen dient der Überprüfung und Herstellung neuer Korrespondenzbeziehungen (Aktivitäten 6–9). Die einzelnen Aktivitäten werden im Folgenden detailliert vorgestellt.

Konsistenzprüfung

Die Überprüfung der Korrespondenzbeziehungen beginnt mit der Identifikation von bereits angewandten Korrespondenzregeln (Aktivität 1). Das Story-Pattern hierzu ist in Abbildung 5.10 dargestellt. In dem Story-Pattern wird zunächst der als Parameter übergebene Korrespondenzknoten `node` an das Objekt `bc` gebunden. Anschließend wird die aktuelle Regel, die durch das

Objekt `this` repräsentiert wird, sowie eine Instanz der Klasse `TGGEngine` gebunden. Ausgehend von diesen beiden Objekten wird überprüft, ob bereits ein Korrespodenenzknoten vom Typ `Pr2Cl` existiert, der durch diese Regel erzeugt wurde. Hierbei muss der erzeugte Korrespondenzknoten einerseits Nachfolger des Korrespondenzknotens `bc` sein (vgl. Link `succ`). Andererseits muss dieser Korrespondenzknoten als Eingabeknoten für diese Korrespondenzregel fungiert haben (vgl. Link `inputNode`). In dem Fall, dass alle spezifizierten Objekte gebunden und die spezifizierten Links zwischen diesen Objekten erfolgreich überprüft wurden, haben wir eine Regelanwendung gefunden.

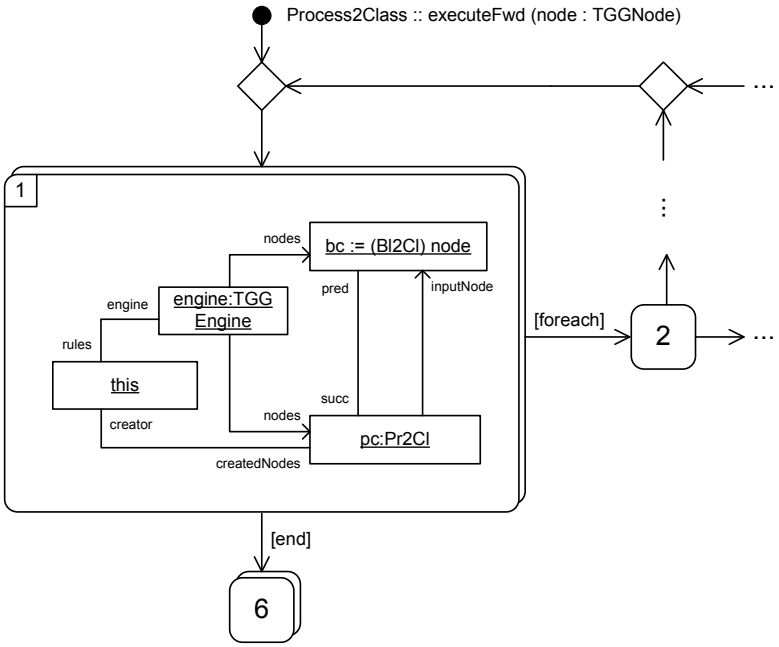


Abbildung 5.10: Hergestellte Korrespondenzbeziehung identifizieren

Für jede gefundene Regelanwendung wird überprüft, ob die bei der Regelanwendung zugrundeliegende Objektstruktur immer noch gegeben ist (Aktivität 2, Abbildung 5.11). Hierzu wird versucht, alle Objekte und Links zu binden, die von den Korrespondenzknoten durch `sources` und targets-Assoziationen erreichbar sind. Dabei handelt es sich um Objekte, die bereits durch die `handled`-Assoziation als verbraucht markiert worden sind. In diesem Zusammenhang wird auch die negative Anwendungsbedingung der TGG-Regel überprüft.

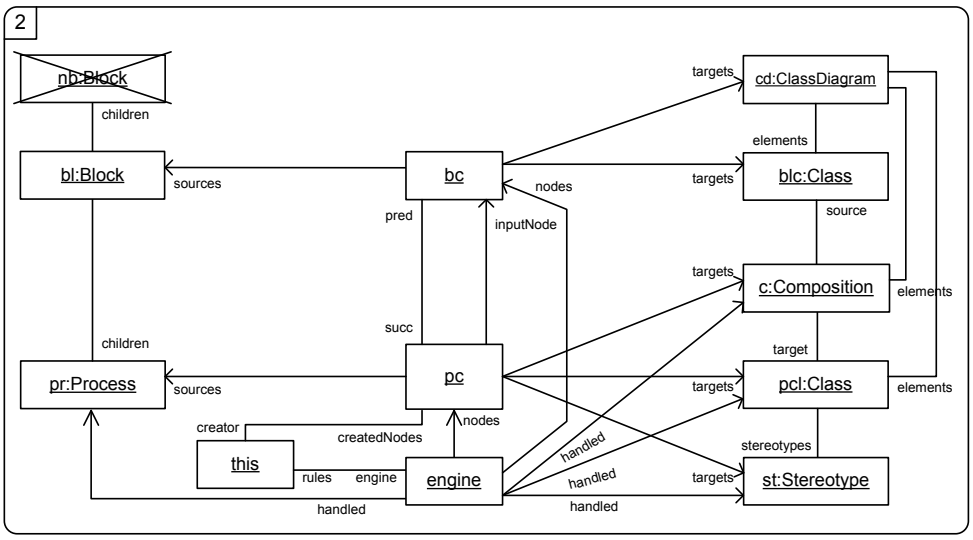


Abbildung 5.11: Objektstruktur der Korrespondenzbeziehung überprüfen

Falls die negative Anwendungsbedingung nicht mehr erfüllt ist, oder falls Objekte und/oder Links gelöscht wurden, so ist die Korrespondenzbeziehung nicht mehr gültig. Sie wird daher wieder rückgängig gemacht (Aktivität 3). In dem Fall, dass die Objektstrukturen dieses Story-Patterns in den Modellen weiterhin gegeben sind, müssen nur noch die Attributwerte überprüft und aktualisiert werden (Akitvitäten 4 und 5).

Inkonsistente Korrespondenzbeziehung auflösen

Eine bestehende Korrespondenzbeziehung wird durch einen Korrespondenzknoten repräsentiert. Somit kann eine Korrespondenzbeziehung am einfachsten aufgelöst werden, indem der Korrespondenzknoten und alle dazu inzidenten Links gelöscht werden. Das hierfür verantwortliche Story-Pattern ist in Abbildung 5.12 dargestellt.

In diesem Story-Pattern wird auf dem Objekt `engine` die Methode `deleteFwd` aufgerufen. Abbildung 5.13 zeigt die Methode in Pseudocode-Syntax. Der Methode wird der zu löschende Korrespondenzknoten als Parameter übergeben. Weil dieser Korrespondenzknoten gelöscht werden soll, werden automatisch alle von diesem Korrespondenzknoten abhängigen Regelanwendungen ebenfalls ungültig – sie müssen daher ebenfalls zurück genommen werden. Daher wird die Methode rekursiv auf allen Nachfolgern dieses Korrespondenzknotens ausgeführt (Zeilen 2 und 3). Falls keine Nach-

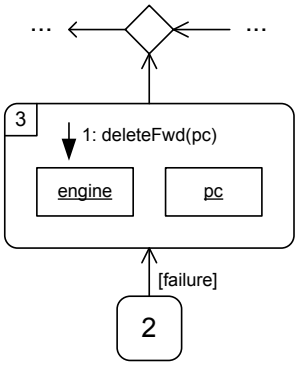


Abbildung 5.12: Inkonsistente Korrespondenzbeziehung auflösen

folger mehr vorhanden sind, werden die `handled`-Links zu allen Objekten im Blockdiagramm entfernt (Zeilen 4 und 5). Anschließend werden – da wir die Modellsynchronisation in Vorwärtsrichtung durchführen – alle referenzierten Objekte im Klassendiagramm gelöscht (Zeilen 6 und 7). Schließlich werden der betrachtete Korrespondenzknoten und alle dazu inzidenten Links gelöscht (Zeile 8).

```
1: TGGEngine::deleteFwd(node : TGGNode)
2:   for each (s : TGGNode in node.succ) do
3:     self->deleteFwd(s);
4:   for each (o : Object in node.sources) do
5:     self.handled->remove(o);
6:   for each (o : Object in node.targets) do
7:     delete o;
8:   delete node;
```

Abbildung 5.13: Die Methode `deleteFwd`

Natürlich sind beim Zurücknehmen einer Regelanwendung auch andere Strategien denkbar. Beispielsweise könnten in beiden Modellen die referenzierten Objekte lediglich als unverbraucht markiert werden. In diesem Fall würde allerdings nach dem Löschen eines Blocks eine Modellsynchronisation in Rückwärtsrichtung dazu führen, dass ein solcher Block wieder erzeugt werden würde. Eine andere Möglichkeit wäre, alle referenzierten Objekte einfach zu löschen. In diesem Fall würde aber beispielsweise beim Entfernen oder Ändern des Stereotyps die Klasse und die Kompositionsbeziehung

im Rahmen der Modellsynchronisation automatisch gelöscht werden. Für den Benutzer wäre ein solches Verhalten – insbesondere bei einer automatischen Synchronisation nach jeder Änderung – nur schwer nachvollziehbar. Aus diesem Grund haben wir uns für die richtungsabhängige Behandlung ungültig gewordener Regelanwendungen in der hier vorgestellten Art und Weise entschieden. Für andere Anwendungsszenarien ist es aber durchaus denkbar, dass eine andere Strategie angemessener ist. Die alternativen Strategien können durch eine andere Implementierung der Methode `deleteFwd` umgesetzt werden.

Attributaktualisierung

Nach einer erfolgreichen Überprüfung der Objektstruktur müssen zusätzlich Attributbedingungen überprüft und gegebenenfalls aktualisiert werden (Aktivitäten 4 und 5). Diese beiden Aktivitäten stellen ein Generierungsmuster dar und werden nur dann generiert, wenn in der TGG-Regel mindestens eine Attributbedingung spezifiziert worden ist. Sind mehrere Attributbedingungen in der TGG-Regel spezifiziert, so werden nach diesem Muster für jede Attributbedingung jeweils zwei separate Aktivitäten erzeugt. Falls jedoch keine Attributbedingung spezifiziert wurde, wechselt der Kontrollfluss direkt zur Aktivität 1. In der Abbildung 5.9 ist die Generierung der Aktivitäten 4 und 5 daher durch das grau schattierte Rechteck als optional gekennzeichnet.

Unsere Beispielregel *Process2Class* enthält zwei Attributbedingungen. Die erste Attributbedingung fordert, dass der Prozess und die dazu korrespondierende Klasse einen identischen Namen besitzen. Die zweite Bedingung besagt, dass es sich bei dem Stereotyp der Klasse um einen Stereotyp der Art „process“ handeln soll. Für diese beiden Attributbedingungen werden daher auf der Grundlage des Generierungsmusters insgesamt vier Aktivitäten erzeugt. Die erzeugten Aktivitäten sind in der Abbildung 5.14 zu sehen.

Die Überprüfung der ersten Bedingung findet in der Aktivität 4a statt. Ist diese Bedingung verletzt, so wird der Name des Prozesses in der Aktivität 5a an die Klasse propagiert. Infolge der durchgeführten Änderung müssen alle Regelanwendungen, die auf dem Korrespondenzknoten `pc` basieren, überprüft werden. Dies wird dem Synchronisationsalgorithmus signalisiert, indem das Attribut `descend` der Objekte `engine` und `pc` auf den Wert `true` gesetzt wird.

Unabhängig davon, ob eine Attributpropagation stattgefunden hat oder nicht, führt der Kontrollfluss zur Aktivität 4b, in der die zweite Bedingung überprüft wird. Ist diese Bedingung verletzt, so wird das Attribut `kind` in

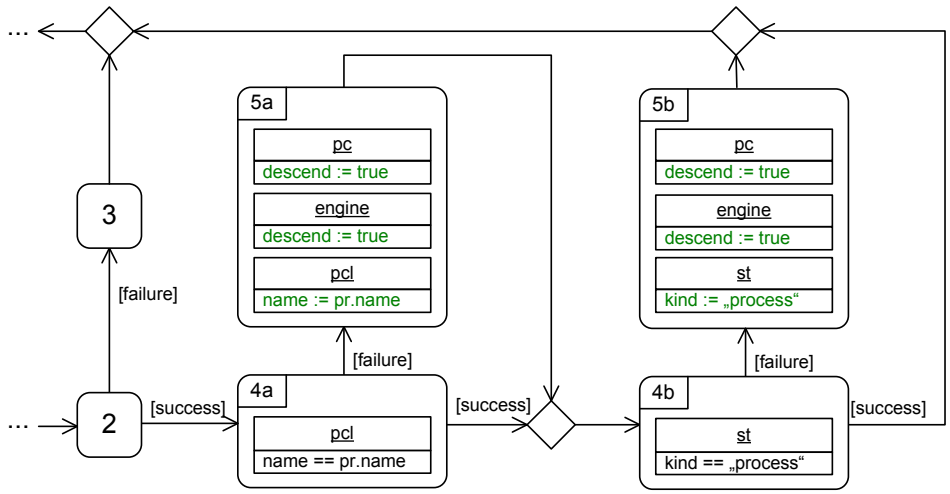


Abbildung 5.14: Attributwerte überprüfen und aktualisieren

der Aktivität 5b aktualisiert. Auch hier wird die Aktualisierung mit Hilfe der `descend`-Attribute dem Synchronisationsalgorithmus mitgeteilt. Da es sich hierbei um die letzte zu überprüfende Attributbedingung handelt, wechselt der Kontrollfluss anschließend wieder zur Aktivität 1.

Neue Anwendungsstelle suchen

Nach der Überprüfung und Aktualisierung aller bereits bestehenden Korrespondenzbeziehungen (Aktivitäten 1–5) wird die Aktivität 1 über die mit `end` annotierte Bedingung verlassen, um nach neuen Korrespondenzbeziehungen zu suchen (Aktivitäten 6–9). Neue Korrespondenzbeziehungen ergeben sich insbesondere dann, wenn der Benutzer neue Objekte und/oder Links zu den Modellen hinzufügt. Ebenso können aber neue Korrespondenzbeziehungen entstehen, wenn Objekte geändert oder aus dem Modell gelöscht werden und dadurch die geforderten (negative) Anwendungsbedingungen erfüllt sind.

In unserem Beispiel könnte ein Benutzer beispielsweise einen neuen Block im Blockdiagramm erstellen. In diesem Fall müsste im Rahmen der Modellsynchronisation im Klassendiagramm eine neue Klasse mit einem Stereotyp und einer Komposition erzeugt werden. Hierzu wird zunächst im Blockdiagramm nach möglichen Anwendungsstellen für die Regeln gesucht. Für unsere Beispielregel bedeutet dies, dass zunächst nur nach neuen Prozessen gesucht wird. Das hierfür aus der TGG-Regel *Process2Class* generierte Story-Pattern ist in Abbildung 5.15 dargestellt.

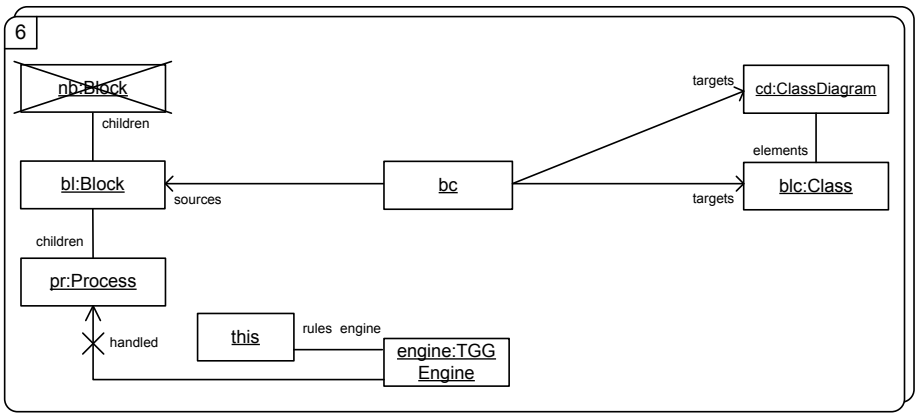


Abbildung 5.15: Neue Anwendungsstelle suchen

Ausgehend von dem betrachteten Korrespondenzknoten `bc` werden zunächst die Objekte `bl`, `cd` und `blc` gebunden sowie die negative Anwendungsbedingung `nb` überprüft. Diese Objekte bilden einen notwendigen Kontext für eine erfolgreiche Regelanwendung. Bei der Ausführung dieses Story-Patterns kann ausserdem nicht davon ausgegangen werden, dass das Objekt `engine` bereits in der vorherigen Aktivtität 1 gebunden worden ist. Daher wird dieses Objekt sicherheitshalber erneut gebunden.

Anschließend wird nach einem neuen Prozess `pr` gesucht. Damit nur unverbrauchte Prozesse berücksichtigt werden, darf der Prozess nicht durch eine `handled`-Assoziation mit dem `engine`-Objekt verlinkt sein. Dies wird durch den durchgestrichenen `handled`-Link spezifiziert, der damit eine negative Anwendungsbedingung darstellt und daher auch als *negativer Link* bezeichnet wird. Für jeden Prozess, der gebunden werden kann und noch nicht verbraucht ist, wird zur Aktivität 7 verzweigt.

Integration

Für jede gefundene Anwendungsstelle wird zunächst überprüft, ob dazu korrespondierende Elemente bereits im Zielmodell existieren (Aktivität 7). Prinzipiell wird hierfür die operationale Graphersetzungsregel zur Modellintegration aus der Abbildung 5.7(b) ausgeführt. Die Anwendung dieser Integrationsregel stellt sicher, dass Objekte im Zielmodell wiederverwendet und nicht neu erzeugt werden.

Das entsprechende Story-Pattern zu unserer Beispielregel ist in Abbildung 5.16 zu sehen. Alle bereits in der Aktivität 6 gebundenen Objekte

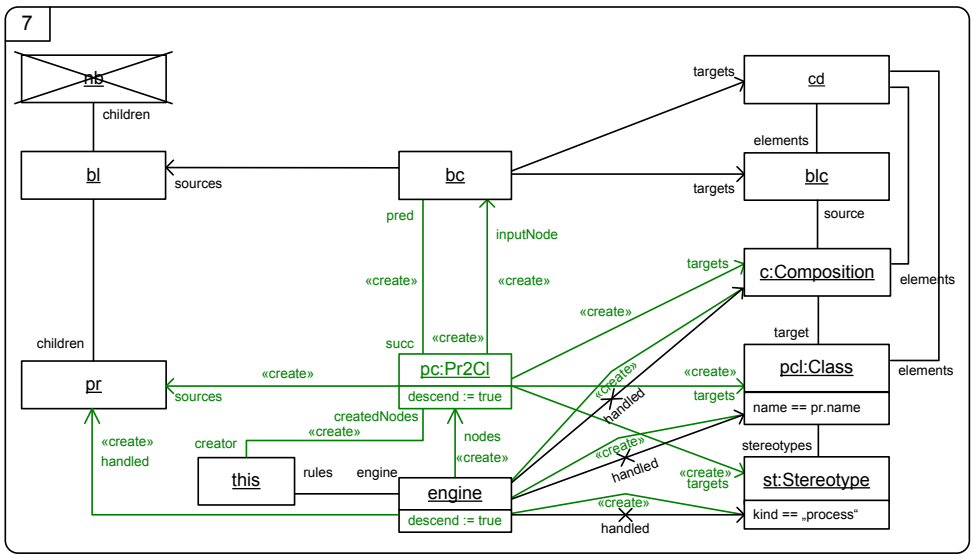


Abbildung 5.16: Integrationsregel anwenden

erscheinen jetzt ohne Typangabe. Im Gegensatz dazu sind die neu zu bindenden Objekte im Klassendiagramm mit einem Typ versehen. Analog zu den Objekten im Blockdiagramm dürfen auch diese Objekte noch nicht verbraucht sein. Dies wird durch die negativen `handled`-Links ausgedrückt. Auf die Überprüfung des negativen `handled`-Links zu dem Objekt `pr` kann verzichtet werden, da diese Überprüfung bereits in der Aktivität 6 stattgefunden hat.

Zur erfolgreichen Anwendung der Integrationsregel müssen die in der TGG-Regel spezifizierten Attributbedingungen erfüllt sein. In unserem Beispiel muss das Objekt `pcl` genauso benannt sein wie der Prozess. Darüber hinaus muss das Attribut `kind` des Objekts `st` den Wert „process“ besitzen. In dem Fall, dass die Objekte diese Bedingungen erfüllen, wird eine Korrespondenzbeziehung zwischen den Objekten hergestellt, indem ein neuer Korrespondenzknoten `pc` erzeugt wird. Der neu erzeugte Korrespondenzknoten stellt die Korrespondenzbeziehung zwischen den Objekten über die `sources`- und `targets`-Links her.

Im Rahmen der Modellsynchronisation werden der Prozess im Blockdiagramm sowie alle neu erstellten Objekte im Klassendiagramm durch die neu erzeugten `handled`-Links als verbraucht markiert. Die für die Erstellung des Korrespondenzknotens verantwortliche Regel wird mit Hilfe des Links `creator` für eventuell nachfolgende Konsistenzprüfungen gespeichert.

Für einen schnelleren Zugriff auf den erzeugten Korrespondenzknoten wird ein `nodes`-Link erstellt. Zusätzlich werden ein `succ` und ein `inputNode`-Link zwischen dem bereits gebundenen und dem neu erzeugten Korrespondenzknoten hinzugefügt, um ein gültiges Korrespondenzmodell aufzubauen. Schließlich wird dem Synchronisationsalgorithmus durch das Setzen des Attributs `descend` auf den Wert `true` in den Objekten `engine` und `pc` signalisiert, dass der neu erzeugte Korrespondenzknoten auf neue Regelanwendungen überprüft werden muss.

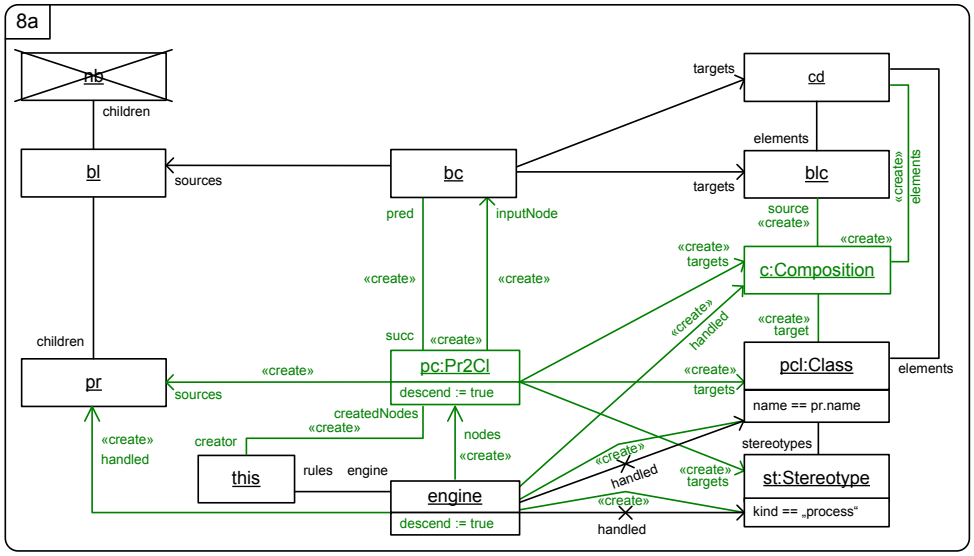
Automatische Vervollständigung

Häufig ist eine Modellintegration nicht möglich, weil keine korrespondierenden Elemente im Zielmodell existieren. In diesen Fällen muss die Modellsynchronisation – wie im nächsten Abschnitt beschrieben wird – durch eine operationale Graphersetzungsregel zur Modelltransformation durchgeführt werden. In einigen Fällen sind aber zumindest einige der geforderten Objekte vorhanden, so dass zur Modellsynchronisation nur die noch fehlenden Objekte erzeugt werden müssen. Diesen Vorgang bezeichnen wir als automatische Vervollständigung.

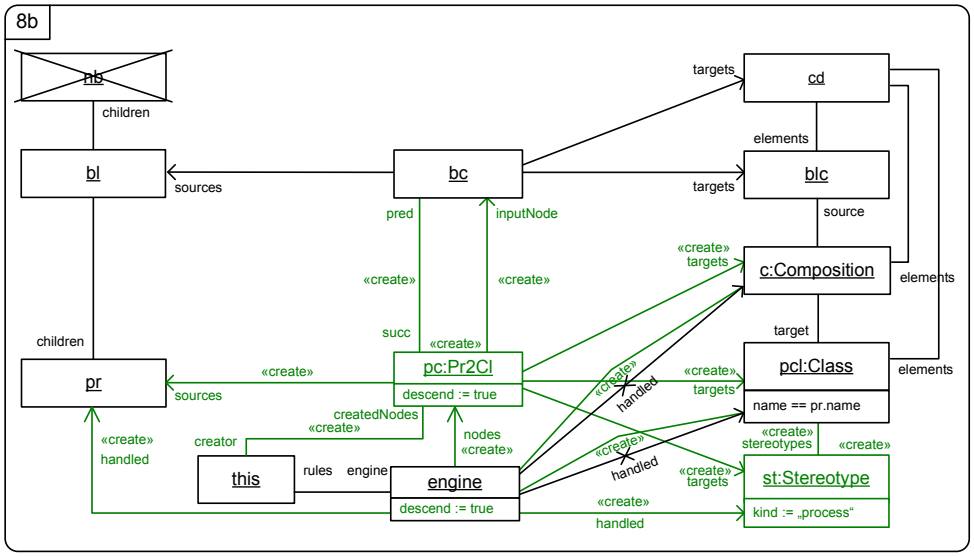
Bei der automatischen Vervollständigung wird zunächst untersucht, ob wenigstens einige Elemente im Zielmodell vorhanden sind, die wiederverwendet werden können. In diesem Fall erfolgt eine automatische Vervollständigung der noch fehlenden Elemente (Aktivität 8). Dabei existieren im Allgemeinen mehrere Situationen, die zu überprüfen sind. In Abbildung 5.17 haben wir für unsere Beispielregel zwei dieser Situationen dargestellt.

Die erste Möglichkeit zur automatischen Vervollständigung ist im Story-Pattern der Abbildung 5.17(a) dargestellt. In diesem Story-Pattern wird überprüft, ob eine Klasse existiert, die denselben Namen besitzt wie der Prozess. Zusätzlich muss diese Klasse mit einem ≪process≫-Stereotypen versehen sein. Ist diese Situation im Klassendiagramm gegeben, so wird die Objektstruktur um das noch fehlende Objekt `c` mit entsprechenden Links zu den Objekten `blc` und `pcl` ergänzt, d. h., eine Komposition zwischen den beiden Klassen erzeugt.

Die zweite Möglichkeit zur automatischen Vervollständigung ist in der Abbildung 5.17(b) zu sehen. Auch in diesem Fall wird eine Klasse vorausgesetzt, die denselben Namen besitzt wie der Prozess. Zusätzlich muss zwischen dieser Klasse und der Klasse, die den Block repräsentiert, eine Kompositionsbeziehung vorhanden sein. Im Gegensatz zum vorherigen Story-Pattern wird ein Stereotyp nicht gefordert.



(a) Eine Möglichkeit der Autovervollständigung



(b) Eine weitere Möglichkeit zur Autovervollständigung

Abbildung 5.17: Automatische Vervollständigung

Natürlich existieren neben diesen beiden Möglichkeiten noch weitere Möglichkeiten für unvollständige Objektstrukturen. Beispielsweise könnte überprüft werden, ob lediglich eine Klasse vorhanden ist, die denselben Namen besitzt wie der Prozess. Für eine gültige Korrespondenzbeziehung müsste dann eine dazugehörige Kompositionsbeziehung und ein Stereotyp erzeugt werden. Andere Möglichkeiten für eine automatische Vervollständigung ergeben sich, wenn einfach nur noch fehlende Links zwischen den geforderten Objekten erzeugt werden müssen. In einigen Fällen könnte es durchaus sinnvoll sein, Objekte wiederzuverwenden, obwohl deren Attributwerte nicht die geforderten Bedingungen erfüllen. Dazu müsste die automatische Vervollständigung die Attributwerte dementsprechend anpassen. Welche dieser Möglichkeiten tatsächlich in Betracht kommen, hängt jedoch häufig von dem Anwendungsszenario und den damit verbundenen TGG-Regeln ab.

Die Erzeugung der Story-Pattern zur automatischen Vervollständigung kann automatisch erfolgen, indem alle möglichen Kombinationen der zu suchenden Objekte erzeugt werden. Wie in der Arbeit von Jörg Baksmeier jedoch gezeigt wurde, führt dies zu automatischen Vervollständigungen, die weder bei der Spezifikation der TGG-Regeln vorgesehen noch vom Benutzer erwünscht waren [Bak06]. Zur Lösung des Problems wurde in der Arbeit daher vorgeschlagen, Objekte und Links zu Gruppen zusammenzufassen, die als eine zusammengehörige Einheit betrachtet und nur gemeinsam gesucht oder vervollständigt werden. Nach der zusätzlichen Spezifikation einer solchen Gruppierung konnten viele problematische Fälle bei der Generierung der Story-Pattern zur automatischen Vervollständigung vermieden werden.

Die automatische Generierung der Story-Pattern führt jedoch selbst unter Einsatz der beschriebenen Gruppierung nicht immer zu sinnvollen Objektstrukturen, weil die Semantik der Objekte weder aus den gegebenen Metamodellen noch aus den spezifizierten TGG-Regeln hervorgeht. Zur automatischen Generierung der Story-Pattern aus unserer Beispielregel müsste jede erdenkliche Kombination der Objekte `c`, `pcl` und `st` berücksichtigt werden. Dabei würde auch ein Story-Pattern entstehen, in dem nur die Kompositionsbeziehung, d. h., das Objekt `c`, überprüft und anschließend um die noch fehlende Klasse mit einem Stereotyp ergänzt wird. Diese Überprüfung und Vervollständigung ist aber wenig sinnvoll, da eine Kompositionsbeziehung immer eine Ursprungs- und Zielklasse besitzen muss.

Damit der Spezifizierer die zu generierenden Story-Pattern zur automatischen Vervollständigung direkt beeinflussen kann, müssen im Rahmen dieser Arbeit die zu vervollständigenden Objektstrukturen zusätzlich vor der Generierung der Story-Pattern angegeben werden. Dazu reicht es aus, die zu

suchende Objektstruktur zu spezifizieren. Bei der Generierung wird eine solche Objektstruktur herangezogen, um die Unterschiede zur Objektstruktur der TGG-Regel zu bestimmen und das entsprechende Story-Pattern zur automatischen Vervollständigung zu erzeugen.

Existieren mehrere Möglichkeiten zur automatischen Vervollständigung, werden die hierzu generierten Story-Pattern aneinandergereiht und durch entsprechende Transitionen miteinander verbunden. Wurde ein Story-Pattern erfolgreich ausgeführt, wird es über eine `success`-Transition verlassen. Der Kontrollfluss wechselt infolge dessen wieder zur Aktivität 6. Kann das Story-Pattern jedoch nicht angewendet werden, so wird über eine `failure`-Transition zum nächsten Story-Pattern gewechselt, das eine weitere Möglichkeit der automatischen Vervollständigung überprüft. In dem Fall, dass gar keine automatische Vervollständigung ausgeführt werden kann, wird die Modellsynchronisation durch eine operationale Graphersetzungsregel zur Modelltransformation ausgeführt.

Transformation

Ist eine automatische Vervollständigung nicht gewünscht oder möglich, so wird eine Modellsynchronisation durch Modelltransformation ausgeführt (Aktivität 9). Das dazu erzeugte Story-Pattern entspricht der operationalen Graphersetzungsregel aus Abbildung 5.7(a). Das aus unserer Beispielregel generierte Story-Pattern zur Modellsynchronisation durch Modelltransformation ist in der Abbildung 5.18 zu sehen.

Die in dem Story-Pattern erzeugten Objektstrukturen entsprechen den bereits beschriebenen Story-Pattern zur Modellintegration und der automatischen Vervollständigung. Der einzige Unterschied besteht darin, dass zur Herstellung der Korrespondenzbeziehung in dem Story-Pattern zur Modelltransformation alle Objekte im Klassendiagramm neu erzeugt werden. Nach der Anwendung dieses Story-Patterns wird wieder das Story-Pattern der Aktivität 6 ausgeführt.

Falls in der Aktivität 6 kein unverbrauchter Prozess mehr gefunden wird, wechselt der Kontrollfluss zur Stoppaktivität. Damit wird die Ausführung dieser Regel auf dem als Parameter übergebenen Korrespondenzknoten beendet. Der Synchronisationsalgorithmus sorgt anschließend dafür, dass entweder weitere Regeln auf diesem Korrespondenzknoten überprüft werden oder der nächste Korrespondenzknoten untersucht wird.

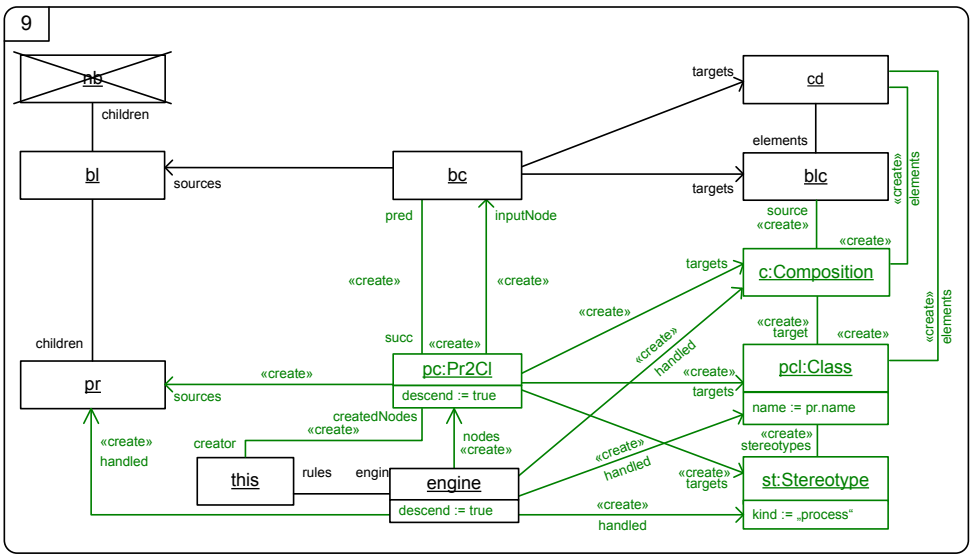


Abbildung 5.18: Modelltransformation ausführen

Sonderfälle

An dieser Stelle betrachten wir einige Sonderfälle, die bei der Generierung von Storydiagrammen zu beachten sind.

Storydiagramme zur Modellintegration Während die Storydiagramme für die Modellsynchronisation in Rückwärtsrichtung analog zu den zuvor beschriebenen Storydiagrammen für die Modellsynchronisation in Vorwärtsrichtung generiert werden, müssen bei der Generierung von Storydiagrammen für die Modellintegration einige Besonderheiten berücksichtigt werden.

Bei der Modellintegration werden korrespondierende Modellelemente identifiziert und zueinander in Beziehung gesetzt. Dabei findet neben der Überprüfung der Objektstruktur auch eine Überprüfung der geforderten Attributbedingungen statt. Allerdings werden bei der Modellintegration keine Attribute aktualisiert, um Korrespondenzbeziehungen aufrecht zu erhalten. Dies geschieht weiterhin durch die Ausführung einer Modellsynchronisation in eine festgelegte Richtung.

Aus diesem Grund entfällt bei der Generierung der Storydiagramme zur Modellintegration die Aktivität 5. Darüber hinaus werden die Aktivitäten 2 und 4 zusammengelegt, so dass die Überprüfung der Attri-

butbedingungen zusammen mit der Überprüfung der Objektstruktur stattfindet. In dem Fall, dass eine Inkonsistenz festgestellt wird, wird in der Aktivität 3 die Methode `deleteMap` aufgerufen. In dieser Methode werden die beteiligten Objekte sowohl im Quell- als auch im Zielmodell als unverbraucht markiert und lediglich der Korrespondenzknoten mit den dazu inzidenten Links gelöscht.

Zur Überprüfung und Herstellung neuer Korrespondenzbeziehungen wird nur die Aktivität 6 benötigt, d. h., die Aktivitäten 7, 8 und 9 entfallen. In die Aktivität 6 wird ein Story-Pattern mit der Integrationsregel eingebettet (vgl. Abbildungen 5.7(b) und 5.16), wobei hier alle Objekte mit Typangaben versehen sein müssen, damit sie gebunden werden. Zusätzlich müssen die Attributbedingungen überprüft und bei einer erfolgreichen Regelanwendung ein Korrespondenzknoten mit all den notwendigen Links erstellt werden.

Storydiagramme für komplexere Regeln In einigen Fällen können TGG-Regeln von der in Abbildung 5.6 gezeigten Grundstruktur abweichen und mehrere Korrespondenzknoten für den zu erzeugenden Korrespondenzknoten voraussetzen. Ein Beispiel für eine solche TGG-Regel haben wir in Abbildung 3.8 gesehen (siehe Seite 66). Die Grundstruktur einer solchen Regel ist in Abbildung 5.19 dargestellt, wobei wir hier nur zwei Korrespondenzknoten voraussetzen. Diese Grundstruktur ist jedoch auch auf mehr als zwei vorausgesetzte Korrespondenzknoten verallgemeinerbar.

Zur Generierung eines Storydiagramms kann das Prinzip angewandt werden, das wir bereits in Abschnitt 5.3.1 kennen gelernt haben. Der in dieser Arbeit vorgestellte Synchronisationsmechanismus übergibt allerdings an jedes Storydiagramm genau einen Korrespondenzknoten als Eingabe, der dann den Ausgangspunkt für die Regelanwendung innerhalb eines Storydiagramms bildet. Bei der Generierung eines Storydiagramms müsste daher einer der vorausgesetzten Korrespondenzknoten als Eingabeknoten festgelegt werden.

Legt man einen der Korrespondenzknoten als Eingabe für die Regelanwendung fest, so kann es während einer Modellsynchronisation vorkommen, dass zu dem Zeitpunkt, an dem die Regel überprüft wird, einer oder sogar mehrere der anderen zur Regelanwendung benötigten Korrespondenzknoten noch nicht existieren. Damit würde die Regelanwendung erfolglos abgebrochen werden und müsste zu einem späteren

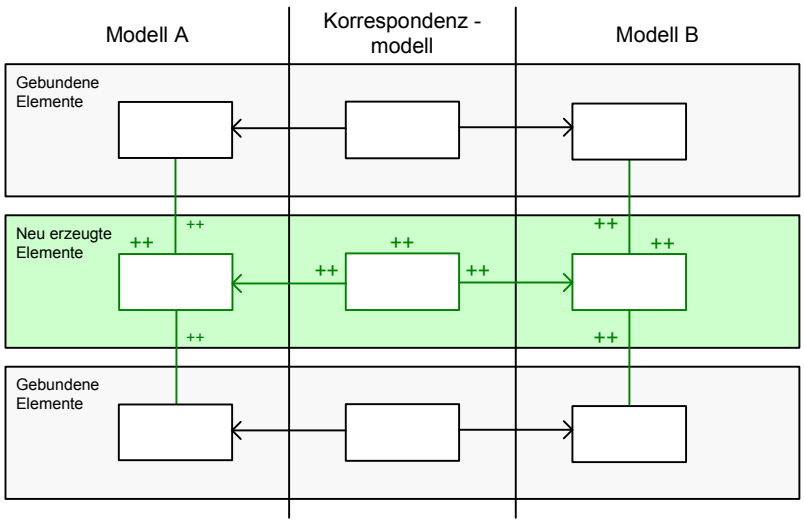


Abbildung 5.19: Grundstruktur einer komplexeren TGG-Regel

Zeitpunkt, d. h., wenn der oder die anderen Korrespondenzknoten erstellt worden sind, erneut ausgeführt werden. Hierfür müsste das Korrespondenzmodell ein zweites Mal durchlaufen werden.

Um die wiederholte Traversierung des Korrespondenzmodells zu vermeiden und die Synchronisation in einem Durchlauf zu ermöglichen, muss die Regelanwendung von jedem der vorausgesetzten Korrespondenzknoten ausführbar sein, d. h., jeder Korrespondenzknoten muss als Eingabeknoten und damit als Ausgangspunkt der Regelanwendung im Storydiagramm akzeptiert werden. Dies wird realisiert, indem für jeden vorausgesetzten Korrespondenzknoten die in Abbildung 5.9 gezeigte Grundstruktur generiert wird und der entsprechende Korrespondenzknoten an den Parameter des Storydiagramms gebunden wird. Zusätzlich werden bei einer erfolgreichen Regelanwendung entsprechende `inputNode`-Links zwischen diesem Korrespondenzknoten und den neu erzeugten Korrespondenzknoten erstellt, so dass auch die Korrespondenzprüfung in den Aktivitäten 1–5 korrekt ausgeführt werden kann. Die mehrfach erzeugten Grundstrukturen werden in einem Storydiagramm hintereinander gereiht.

Storydiagramme für Axiome Ein Axiom repräsentiert eine festgelegte Ausgangssituation und bildet damit eine Grundlage für die Anwendung von Korrespondenzregeln. Daher besitzt ein Axiom einen Sondersta-

tus: der Korrespondenzknoten eines Axioms wird nur dann gelöscht, wenn keine Synchronisation der Modelle mehr gewünscht ist.

Damit ein Benutzer die notwendige Ausgangssituation für eine Modellsynchronisation zwischen zwei Modellen möglichst einfach und automatisch herstellen kann, werden in dieser Arbeit auch aus einem Axiom Storydiagramme generiert. Diese sind jedoch wesentlich einfacher gehalten. Die aus einem Axiom generierten Storydiagramme für die Methoden `executeFwd`, `executeMap` und `executeRev` bestehen jeweils aus einem einzigen Story-Pattern mit dazugehöriger Start- und Stoppaktivität. Die Story-Pattern selbst werden durch die Anwendung des in Abschnitt 5.3.1 vorgestellten Prinzips erzeugt. In Abbildung 5.20 ist das Storydiagramm der Methode `executeFwd` dargestellt, das aus Axiom *System2Class* unseres Beispiels generiert wurde (vgl. Abschnitt 3.2, Abbildung 3.9, Seite 67).

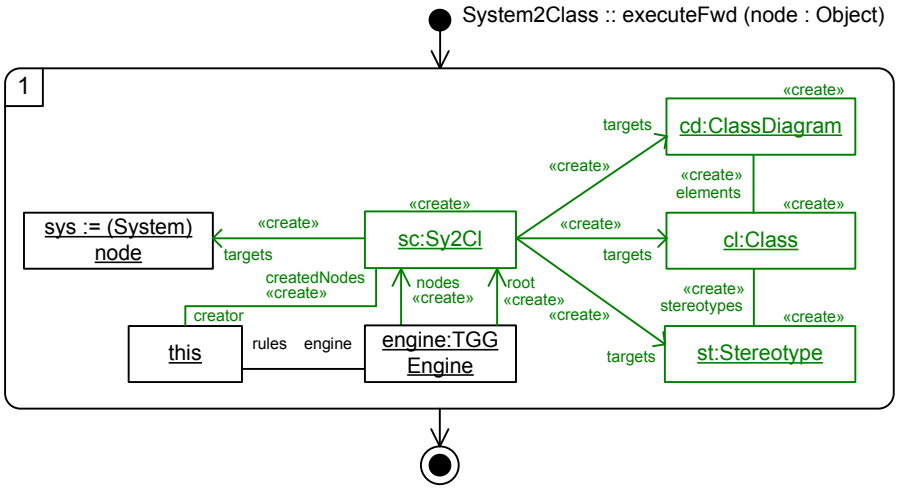


Abbildung 5.20: Storydiagramm zum Axiom *System2Class*

Im Gegensatz zu Storydiagrammen, die aus TGG-Regeln generiert werden, kann der Parameter für Storydiagramme, die aus Axiomen erstellt wurden, nicht an einen Korrespondenzknoten gebunden werden, weil zum Zeitpunkt der Ausführung noch kein Korrespondenzknoten existiert. Daher wird der Parameter an das Objekt `sys` gebunden, d. h., das übergebene Parameterobjekt muss vom Typ `System` sein. Zusätzlich zu der bereits aus den vorangegangenen Abschnitten bekannten Objektstruktur, die aus den Objekten `this` und `engine` mit entspre-

chenden Links besteht, wird in Story-Pattern, die aus einem Axiom generiert werden, der Link `root` erzeugt. Mit Hilfe dieses Links wird der Korrespondenzknoten referenziert, der die Wurzel des Korrespondenzmodells repräsentiert.

5.4 Zusammenfassung

In diesem Kapitel ist ein Synchronisationsmechanismus vorgestellt worden, der auf der Grundlage einer Tripel-Graph-Grammatik eine Modellsynchronisation zwischen zwei Modellen durchführt. Mit dem hier vorgestellten Synchronisationsmechanismus kann eine Synchronisation sowohl batch-artig als auch inkrementell erfolgen. Darüber hinaus kann derselbe Mechanismus sowohl zur Modelltransformation als auch zur Modellintegration eingesetzt werden, da diese Anwendungsszenarien ein grundlegender Bestandteil der in dieser Arbeit vorgestellten Modellsynchronisation sind.

Dem vorgestellten Synchronisationsmechanismus liegt ein invarianter Algorithmus zugrunde, der durch operationale Graphersetzungsregeln parametrisiert wird. Die operationalen Graphersetzungsregeln hingegen sind von einer konkreten Spezifikation der Korrespondenzbeziehungen abhängig und repräsentieren damit den veränderlichen Anteil unseres Modellsynchronisationsmechanismus. Der Algorithmus sowie die dafür notwendige Datenstruktur wurden im ersten Teil dieses Kapitels vorgestellt. Im zweiten Teil haben wir gezeigt, wie operationale Graphersetzungsregeln in Form von Storydiagrammen aus einer Spezifikation generiert werden.

Die in dieser Arbeit vorgestellten Storydiagramme sowie der dazugehörige Steuerungsmechanismus ermöglichen eine partielle Modellsynchronisation. Diese ist jedoch nur möglich, wenn die ursprüngliche TGG-Semantik [Sch94] aufgegeben wird. Die ursprüngliche TGG-Semantik haben wir bereits in Abschnitt 3.2.1 vorgestellt: mit den TGG-Regeln wird beschrieben, wie zwei Modelle simultan, vollständig und konsistent zueinander erzeugt werden können. Dies bedeutet insbesondere, dass zu jedem Modellelement mindestens ein Korrespondenzobjekt existiert, d.h., jedes Modellelement an mindestens einer Korrespondenzbeziehung beteiligt ist. Diese Semantik steht jedoch im Widerspruch zu einer partiellen Modellsynchronisation. Aus diesem Grund wurde die ursprüngliche Semantik der TGG-Regeln in der vorliegenden Arbeit aufgegeben.

Ein weiterer Schwerpunkt dieser Arbeit lag auf der effizienten Ausführung von Modellsynchronisationen. Dabei sollten die Algorithmen zur Modellsyn-

chronisation möglichst so schnell sein, dass auch eine inkrementelle Modellsynchronisation nach jeder Änderung ausgeführt werden kann, ohne dabei den Benutzer bei seiner Arbeit zu stören. Um dieses Ziel zu erreichen, dürfen allerdings keine mehrdeutigen TGG-Regeln verwendet werden, da ansonsten ein aufwändiges Backtracking bei der Regelanwendung zu berücksichtigen ist. Mit dieser Einschränkung ergibt sich allerdings eine weitere Änderung an der ursprünglichen TGG-Semantik.

An dieser Stelle soll daher nicht unerwähnt bleiben, dass der von Andy Schürr geführte Äquivalenzbeweis [Sch94] aufgrund dieser Änderungen nicht mehr gültig ist. In dem Äquivalenzbeweis wurde gezeigt, dass bei der Anwendung der operationalen Graphersetzungsregeln, d.h., in unserem Fall der Storydiagramme, die Reihenfolge der Regelanwendungen unerheblich ist und immer zu gültigen Korrespondenzbeziehungen zwischen den Modellen führt. Dieser Nachweis gilt jedoch nur für die in [Sch94] eingeführten operationale Graphersetzungsregeln. In unserem Fall kann eine geänderte Reihenfolge von Regelanwendungen durchaus zu einem anderen Ergebniss führen.

Die in dieser Arbeit vorgestellte Generierung von operationalen Graphersetzungsregeln, d.h., der Storydiagramme, stellt nur eine Möglichkeit dar, wie eine Modellsynchronisation auf der Grundlage einer Tripel-Graph-Grammatik durchgeführt werden kann. Für andere Anwendungsszenarien können andere Storydiagramme durchaus besser geeignet sein. Gerade in einem solchen Fall zeigt sich ein großer Vorteil dieses Ansatzes: Die neuen Storydiagramme können durch eine Anpassung der hier vorgestellten Generierung erzeugt werden, ohne dass die Spezifikation der TGG-Regeln geändert werden muss.

Kapitel 6

Validierung und Verifikation

Die Modelltransformation ist ein Spezialfall der in dieser Arbeit vorgestellten inkrementellen Modellsynchronisation. Damit eignet sich der hier vorgestellte Ansatz auch zur Spezifikation von Modelltransformationen. Die Spezifikation komplexer Modelltransformationen ist fehleranfällig. Daher ist im Rahmen der Qualitätssicherung häufig ein Nachweis der syntaktischen und semantischen Korrektheit einer solchen Modelltransformation notwendig. Die Validierung und Verifikation von Modelltransformationen ist zurzeit noch Gegenstand der Forschung und nicht sehr weit fortgeschritten.

In diesem Kapitel geben wir daher nur einen Überblick zu ersten Ansätzen der Validierung und Verifikation von Modelltransformationen. Dazu stellen wir in Abschnitt 6.1 einige existierende Ansätze zur Validierung der syntaktischen Korrektheit vor und zeigen, wie diese Ansätze auf TGGs übertragbar sind. Im darauf folgenden Abschnitt 6.2 beschäftigen wir uns mit dem formalen Nachweis der semantischen Korrektheit von Modelltransformationen. Neben anderen existierenden Ansätzen stellen wir hierbei auch einen Ansatz vor, der im Rahmen dieser Dissertation entstanden ist. Wir schließen dieses Kapitel in Abschnitt 6.3 mit einer Zusammenfassung.

6.1 Syntaktische Korrektheit

Neben funktionalen Anforderungen wie der *Terminierung*, die für TGGs bereits in [Sch94] gezeigt wurde, sowie der *Konfluenz*, die für Graphtransformationsregeln mit Hilfe der kritischen Paaranalyse nachgewiesen werden kann [KHE03, LEO08], ist die *syntaktische Korrektheit* eine weitere wichtige Anforderung an Modelltransformationen. Hierbei kann zwischen zwei Formen der syntaktischen Korrektheit unterschieden werden [Küs04b].

Eine Form der syntaktischen Korrektheit liegt vor, wenn die Modelltransformation in einer formalen Transformationssprache spezifiziert wurde. In

diesem Fall müssen die Regeln der Modelltransformation syntaktisch korrekt in Bezug auf die vorliegende Transformationssprache sein.

Die syntaktische Korrektheit von Regeln kann statisch analysiert werden, indem überprüft wird, ob die Regeln konform zum Formalismus der Transformationssprache spezifiziert worden sind und die dort verwendeten Elemente konform zu den Metamodellen der Quell- und Zielsprache angegeben wurden. Häufig werden solche Überprüfungen – wie in unserem Ansatz auch – bereits bei der Spezifikation in einem Regeleditor automatisch durchgeführt und syntaktische Fehler dem Benutzer angezeigt bzw. durch Einschränkung der Bearbeitungsmöglichkeiten in dem Regeleditor erst gar nicht zugelassen.

Allerdings ist es nicht ausreichend, die syntaktische Korrektheit der einzelnen Regeln nachzuweisen. Dies liegt daran, dass Transformationsregeln oft nur Modellfragmente enthalten, die zwar konform zu der Transformationssprache sind, aber einzeln betrachtet keine syntaktisch korrekten (Teil-)Modelle bezüglich der Quell- und Zielsprache darstellen. Hier werden insbesondere statische Integritätsbedingungen, die beispielsweise mit der *Object Constraint Language* (OCL) formuliert werden, nur selten erfüllt.

Bei der anderen Form der syntaktischen Korrektheit wird daher verlangt, dass eine Modelltransformation für jedes syntaktisch korrekte Quellmodell ein syntaktisch korrektes Zielmodell bezüglich der Zielsprache erzeugt. Eine probate und in der Industrie weit verbreitete Methode zur Überprüfung eines Softwaresystems ist die Validierung durch Tests. Diese Methode kann sowohl zur Überprüfung der funktionalen und nicht-funktionalen Anforderungen als auch zur Überprüfung der syntaktischen Korrektheit von Modelltransformationen eingesetzt werden [FSB04].

Abbildung 6.1 zeigt einen allgemeinen Überblick zur Validierung von Modelltransformationen durch Tests. Hierbei werden zunächst Testfälle definiert. Ein Testfall besteht aus einem Quellmodell und einem erwarteten Zielmodell. Anschließend wird die Modelltransformation auf dem Quellmodell ausgeführt. Das Ergebnis dieser Modelltransformation ist ein Ausgabemodell. Dieses Ausgabemodell wird mit dem erwarteten Zielmodell des Testfalls verglichen. Weicht das Ausgabemodell von dem erwarteten Zielmodell ab, so kann daraus geschlussfolgert werden, dass die Modelltransformation fehlerhaft ist. Hierbei ist zu unterscheiden, ob die Implementierung oder die Spezifikation einen Fehler enthält.

In unserem Ansatz wird die Implementierung der Modelltransformation automatisch aus der Spezifikation der TGG-Regeln abgeleitet. Für den Fall, dass die Algorithmen zur automatischen Ableitung dieser Implementierung hinreichend genau getestet wurden, können wir daher im Fehlerfall anneh-

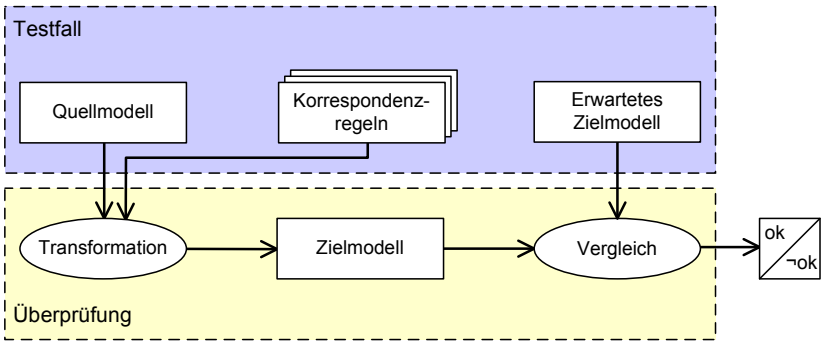


Abbildung 6.1: Überblick zur Validierung durch Tests

men, dass der Fehler in der Spezifikation der TGG-Regeln zu finden ist. Solange die Korrektheit der Algorithmen zur automatischen Ableitung der Implementierung nicht formal bewiesen wurde, kann ein Fehler in der Implementierung jedoch nicht gänzlich ausgeschlossen werden.

Die Methode der Validierung durch Tests kann – sofern eine ausführbare Modelltransformation vorliegt – manuell durchgeführt werden. Hierzu muss der Benutzer die Modelltransformation auf einem Quellmodell ausführen und das Ergebnis der Modelltransformation mit dem erwarteten Zielmodell manuell vergleichen. Ein solcher Vergleich lässt sich allerdings auch automatisiert durchführen. Eine mögliche Automatisierung wurde von Jeff Gray et al. vorgestellt [LZG05]. Für die Modellvergleiche können mittlerweile generische Frameworks und Werkzeuge eingesetzt werden, wie z. B. das SiDiff-Framework [TBWK07] oder EMF-Compare aus dem Eclipse Modeling Framework Technology Project [BGMT].

Im Fall der TGGs könnte eine solche Überprüfung sogar effizienter durchgeführt werden, indem auf der Grundlage der spezifizierten TGG-Regeln eine Modellintegration zwischen dem Quell- und dem erwarteten Zielmodell ausgeführt wird. Anschließend muss überprüft werden, ob alle Modellelemente durch ein Korrespondenzobjekt referenziert werden, d. h., eine vollständige Überdeckung erreicht wurde. Falls das nicht der Fall ist, so ist davon auszugehen, dass auch eine Modelltransformation diese Elemente unberücksichtigt lässt und damit fehlerhaft ist.

Diese Methode kann jedoch nicht angewendet werden, falls eine partielle Modelltransformation getestet werden soll. In einem solchen Transformationsszenario sollen nur Teilmodelle übersetzt werden. Daher müssen einige Modellelemente unberücksichtigt bleiben. Das oben beschriebene Kriterium

der Überdeckung ist somit nicht anwendbar. Darüber hinaus könnte durch die Integration nicht festgestellt werden, ob eine Modelltransformation Modellelemente im Quellmodell nicht doch in das Zielmodell übersetzt.

Um solche Szenarien dennoch automatisch testen zu können, ist eine andere Form der Überprüfung denkbar. Hierzu müsste eine identische Abbildung zwischen Modellen der Zielsprache definiert werden. Liegt eine solche Abbildung vor, so müsste zunächst eine Transformation des Quellmodells ausgeführt werden. Eine anschließende Integration zwischen dem resultierenden Ausgabemodell und dem erwarteten Zielmodell, die auf der Grundlage der identischen Abbildung durchgeführt werden kann, würde darüber Aufschluss geben, ob die Modelle in der geforderten Art und Weise zueinander korrespondieren.

Für eine vollständige Automatisierung müsste allerdings noch untersucht werden, ob die identische Abbildung zwischen den Modellen der Zielsprache automatisch erzeugt werden kann. Eine erste Idee hierzu ist, die identische Abbildung aus den spezifizierten TGG-Regeln zu synthetisieren. Aufgrund der Tatsache, dass eine TGG-Spezifikation verschiedene Produktionen enthält, mit denen der simultane Aufbau eines Quell- und eines Zielmodells beschrieben wird, ist es denkbar, die Produktionen des Zielmodells aus der TGG-Spezifikation zu extrahieren. Daraus kann eine neue TGG-Spezifikation erstellt werden, die eine identische Abbildung zwischen Modellen der Zielsprache beschreibt. Diese TGG-Spezifikation könnte dann für die notwendige Integration zwischen dem resultierenden Ausgabemodell und dem erwarteten Zielmodell eingesetzt werden.

Neben dem Problem der automatischen Vergleiche zwischen einem erwarteten und einem resultierenden Modell einer Modelltransformation besteht ein weiteres Problem darin, geeignete Eingabemodelle für Tests zu finden [BDTM+06]. Im Allgemeinen können zu einem Metamodell unendlich viele Instanzen dieses Metamodells, d. h., Modelle, existieren. Damit ist es unmöglich, alle Modelle zu testen. Einige Arbeiten beschäftigen sich daher damit, relevante und kritische Grenzfälle einer Modelltransformation zu identifizieren und entsprechende Modelle automatisch zu erzeugen [FSB04, BDTM+06, KA06, EKT08].

Mit den automatisch erzeugten Eingabemodellen entsteht allerdings ein weiteres Problem, das für vollständig automatisierte Tests noch zu lösen ist. Dieses Problem hängt damit zusammen, dass automatisch generierte Eingabemodelle nur schwer durch eine Testperson zu interpretieren sind. Wenn eine Testperson ein solches Eingabemodell jedoch nicht vollständig versteht, kann sie auch nicht das erwartete Zielmodell zu dem Eingabemodell defi-

nieren. Baudry et al. schlagen daher vor, durch den Benutzer vorgegebene Modelle auf solche Grenzfälle zu analysieren, um die Qualität der gegebenen Testfälle zu bestimmen. Die Ergebnisse der Analyse können auch verwendet werden, um dem Benutzer Vorschläge für mögliche Erweiterungen der Eingabemodelle zu unterbreiten und dadurch relevante Testfälle zu erstellen [FBMT08]. In anderen Ansätzen wird auf die Spezifikation eines konkreten Zielmodells ganz verzichtet und die resultierenden Zielmodelle lediglich auf bestimmte Eigenschaften überprüft [BDTM+06, NK08b].

Zusammenfassend kann man feststellen, dass zwar bereits einige Ansätze zur Validierung von Modelltransformationen durch Tests existieren, aber nicht alle Probleme zufriedenstellend gelöst sind. Trotz der hohen Relevanz der Validierung durch Tests darf man allerdings nicht vergessen, dass mit Tests zwar die Anwesenheit von Fehlern überprüft, aber nie die Abwesenheit von Fehlern nachgewiesen werden kann. Um die syntaktische Korrektheit einer Modelltransformation zu beweisen, müssen andere, statische Analysetechniken, wie z. B. Model Checking oder Theorembeweiser, auf ihre Eignung zum Nachweis der syntaktischen Korrektheit überprüft werden.

6.2 Semantische Korrektheit

Bei einer Modelltransformation ist es oft wichtig, dass die Transformation eines Modells in eine andere Darstellung bestimmte Eigenschaften dieses Modells erhält. Eine solche Modelltransformation wird als *semantikerhaltend* bzw. *semantisch korrekt* bezeichnet, wenn das Quell- und das Zielmodell bezüglich ihrer Semantik und eines festgelegten Äquivalenzbegriffs zueinander äquivalent sind. Die semantische Korrektheit einer Modelltransformation hängt somit von der Definition einer Äquivalenzrelation und der definierten Semantik des Quell- und Zielmodells ab.

Zur Überprüfung der semantischen Korrektheit einer Modelltransformation existieren zwei grundsätzlich unterschiedliche Ansätze. Der erste Ansatz wird als *Checker-Ansatz* bezeichnet. Bei dem zweiten Ansatz handelt es sich um einen *regelbasierten Ansatz*. Die beiden Ansätze werden in den folgenden beiden Abschnitten kurz vorgestellt.

6.2.1 Checker-Ansatz

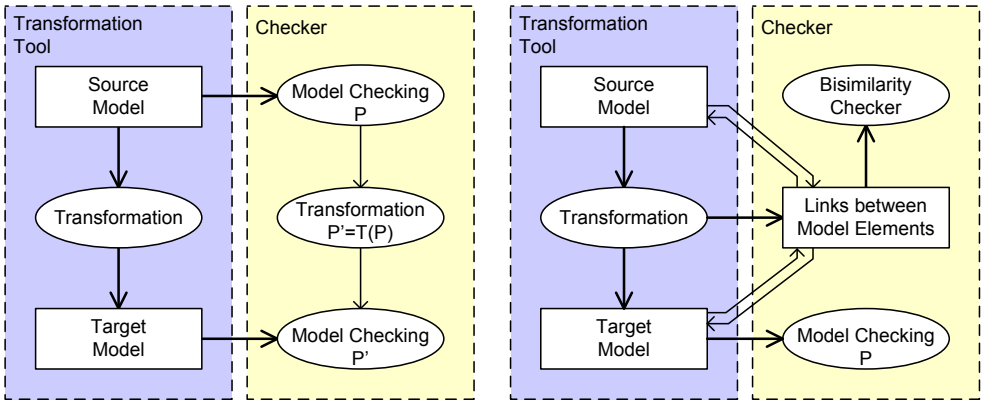
Der Checker-Ansatz kann auf die Arbeit von Pnueli et al. zurückgeführt werden. In dieser Arbeit wird ein Ansatz zur automatischen Validierung

von Übersetzern wie z. B. Compilern und Codegeneratoren vorgestellt. Dieser Ansatz ist auch als *Translation Validation* bekannt geworden [PSS98]. Bei diesem Ansatz werden für jede Ein- und Ausgabe des Übersetzers vorher festgelegte Kriterien automatisch überprüft. Sind die Kriterien erfüllt, kann geschlussfolgert werden, dass die Übersetzung bezüglich der festgelegten Kriterien korrekt ist. Auf der Grundlage dieses Ansatzes wurden für Modelltransformationen zwei Verfahren entwickelt, die in der Abbildung 6.2 zu sehen sind.

Bei dem ersten Verfahren, das in Abbildung 6.2(a) konzeptionell dargestellt ist, wird die Korrektheit mit Hilfe eines *Model Checkers* nachgewiesen [VP03]. Dabei wird nicht die tatsächliche semantische Äquivalenz nachgewiesen, sondern bestimmte, durch den Benutzer festgelegte Korrektheitseigenschaften. Dazu wird ein Modell M der Quellsprache mit Hilfe einer Modelltransformation T in ein Modell M'=T(M) einer Zielsprache automatisch übersetzt. Anschließend wird die Semantik der beiden Modelle in Form von Zustandsübergangssystemen berechnet. Sie dient als Eingabe für einen Model Checker. Der Model Checker überprüft die Gültigkeit einer Korrektheitseigenschaft P auf dem Zustandsübergangssystem von Modell M sowie die Gültigkeit der transformierten Korrektheitseigenschaft P'=T(P) auf dem Zustandsübergangssystem des Zielmodells M'=T(M). In dem Fall, dass beide Korrektheitseigenschaften gültig sind, wird geschlussfolgert, dass auch die Transformation T bezüglich des Prädikats P korrekt ist.

Ein wesentlicher Vorteil dieses Ansatzes liegt darin, dass die festgelegten Kriterien automatisch überprüfbar sind, da sowohl die Berechnung der Zustandsübergangssysteme als auch die Transformation des Prädikats P'=T(P) automatisch erfolgt. Allerdings muss in diesem Ansatz sichergestellt werden, dass die Transformation T(P) korrekt ist, da bei fehlerhaften Transformationsregeln auch die Transformation des Prädikats fehlerhaft verlaufen könnte. Dies muss manuell durch einen Experten überprüft werden. Darüber hinaus müssen geeignete Korrektheitsbedingungen für die Modelle gefunden werden, was – wie die Autoren in ihrem Beitrag anmerken – durchaus keine leichte Aufgabe darstellt.

Bei dem zweiten Ansatz, der in Abbildung 6.2(b) zu sehen ist, wird die Korrektheit einer durchgeführten Modelltransformation mit Hilfe der *Bisimulation* überprüft [NK08a]. Hierzu werden während einer Modelltransformation zwischen den Modellelementen des Quell- und des Zielmodells Verbindungen erstellt, um aus den Modellelementen des Quellmodells hervorgegangene Modellelemente des Zielmodells explizit in Beziehung zu setzen. Diese Verbindungen werden anschließend dazu verwendet, um auf der Grundlage



(a) Ansatz nach Varró et al. [VP03] (b) Ansatz nach Karsai et al. [NK08a]

Abbildung 6.2: Zwei Checker-Ansätze zum Beweis der semantischen Korrektheit von Transformationen

der Bisimulation nachzuweisen, dass sich das Quellmodell bezüglich einer festgelegten Eigenschaft genauso verhält wie das Zielmodell.

In dem Beitrag wird der Ansatz mit dem Ziel verfolgt, ein Entwurfsmodell in ein Analysemodell zu übersetzen, das dann bezüglich einer festgelegten Eigenschaft verifiziert wird. Damit das Ergebnis der Verifikation auf das Quellmodell übertragen werden kann, wird anschließend überprüft, ob die beiden Modelle zueinander bisimular sind. Diese Überprüfung findet automatisch statt und ist leichter, als der Nachweis der tatsächlichen semantischen Äquivalenz. Allerdings muss – wie auch schon im vorherigen Ansatz – die Überprüfung für jede konkrete Instanz der Ein- und Ausgabe einer Modelltransformation stattfinden. Eine allgemeine Überprüfung der Modelltransformationsregeln findet nicht statt.

6.2.2 Regelbasierter Ansatz

Im Gegensatz zum Checker-Ansatz liegen dem regelbasierten Ansatz die spezifizierten Transformationsregeln zugrunde, so dass allgemein bewiesen wird, dass die Transformationsregeln für *jede gültige Eingabe* eine semantisch äquivalente Ausgabe erzeugen. Der formale Beweis der semantischen Äquivalenz muss daher nur einmalig für eine Menge von Transformationsregeln statt finden und nicht – wie im Checker-Ansatz – auf jeder Ein- und Ausgabe.

In Kooperation mit dem Fachgebiet *Programmierung eingebetteter Sys-*

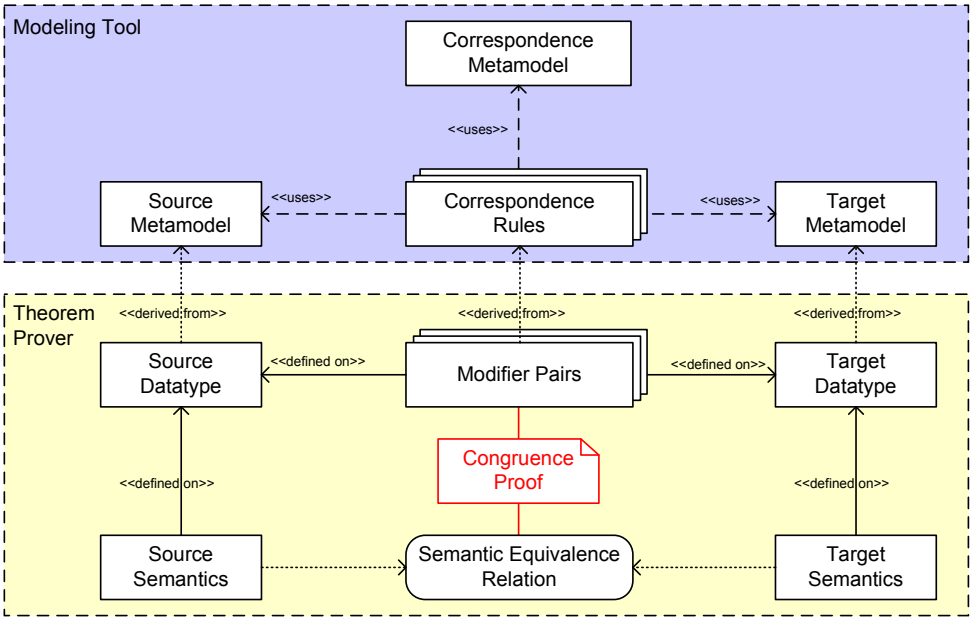


Abbildung 6.3: Überblick zur formalen Verifikation der semantischen Äquivalenz mit einem Theorembeweiser

teme von Prof. Dr. Sabine Glesner wurden im Rahmen dieser Dissertation erste Untersuchungen zu einem solchen Ansatz für TGGs durchgeführt [GGL+06]. Dabei wurde auf der Grundlage des Theorembeweisers Isabelle/HOL[1] ein generisches Beweisschema entwickelt, mit dem die semantische Korrektheit einer Modelltransformation formal nachgewiesen werden kann [Lei06]. Dieses Beweisschema wurde angewandt, um formal zu beweisen, dass eine zuvor spezifizierte Transformation von Automaten in SPS-Code semantikerhaltend ist.

In Abbildung 6.3 ist das generische Beweisschema dargestellt. Im Vergleich zu der Abbildung 2.7 (siehe Seite 47) fehlen hier die Instanzen der Metamodelle, d. h., die Modelle, sowie das Werkzeug zur Ausführung der Modelltransformation, Modellintegration und Modellsynchronisation. Dies liegt darin begründet, dass in diesem Ansatz die semantische Korrektheit der zur Modelltransformation eingesetzten Korrespondenzregeln allgemein bewiesen wird, so dass diese konkreten Modelle für den Beweis irrelevant sind.

[1] Die *Higher Order Logik* (HOL) ist eine typisierte Prädikatenlogik höherer Ordnung.

Um die semantische Korrektheit der TGG-Regeln in Isabelle/HOL nachzuweisen, müssen zunächst die Metamodelle in eine für diesen Theorembeweiser geeignete Darstellung überführt werden. Aufgrund der Tatsache, dass Metamodelle die Menge aller möglichen Instanzen dieses Metamodells und damit eine Syntaxdefinition für gültige Modelle beschreiben, können die Elemente eines Metamodells als Typen und Modelle als Elemente dieser Typen definiert werden.

Bei den in Isabelle/HOL unterstützten Typen handelt es sich um *algebraische Datentypen*, die eine Baumstruktur besitzen. Die durch Metamodelle beschriebenen Modelle hingegen sind im Allgemeinen echte Graphen. Zur Formalisierung von Metamodellen in Isabelle/HOL müssen die Metamodelle daher zunächst auf ein leicht modifiziertes Metamodell abgebildet werden, das eine Baumstruktur besitzt.

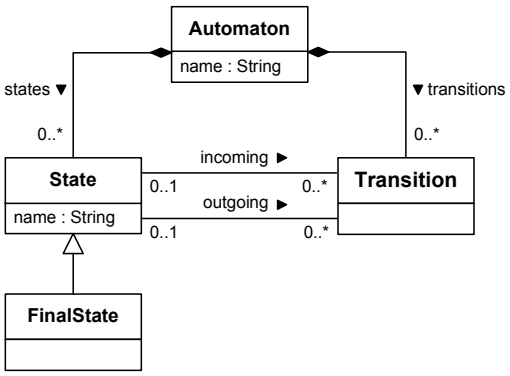
Zur Veranschaulichung ist in Abbildung 6.4(a) ein Ausschnitt aus dem Metamodell für I/O-Automaten dargestellt. Das modifizierte Metamodell ist in Abbildung 6.4(b) zu sehen. Das modifizierte Metamodell erhalten wir, indem Kompositionsbeziehungen und Assoziationen, die zu Zyklen führen können, als Referenzattribute dargestellt werden. Allerdings besteht keine Notwendigkeit, das derartig modifizierte Metamodell tatsächlich zu erstellen, da die Anpassungen direkt auf die notwendigen HOL-Datentypen abgebildet werden können. Die Darstellung dient lediglich dazu, die Abbildung nach Isabelle/HOL leichter nachvollziehbar zu machen.

Die aus dem modifizierten Metamodell entstandenen HOL-Datentypen sind in der Abbildung 6.4(c) dargestellt. Bei den in Isabelle/HOL verwendeten Datentypen kann es sich um zusammengesetzte Datentypen (record), Listen (list) oder andere primitive Datentypen (wie z. B. bool) handeln. Die Abbildung auf die Datentypen in Isabelle/HOL ist zum Teil generisch möglich und wird in [Lei06] genauer vorgestellt.

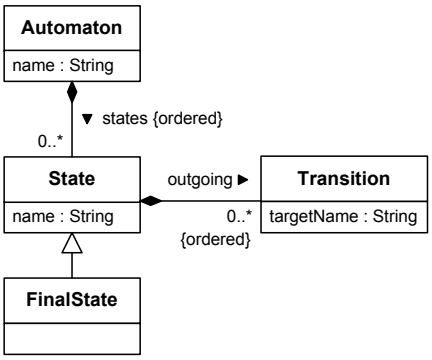
Für den Beweis der Korrektheit einer Modelltransformation müssen die HOL-Datentypen für die Quell- und die Zielsprache mit einer formalen Semantik belegt und eine Äquivalenzrelation definiert werden, mit welcher die Semantik der Modelle verglichen werden kann. Bei der Definition der hierzu benötigten formalen Semantik hat sich gezeigt, dass der übliche Äquivalenzbegriff der *Strukturell Operationalen Semantik*[2] (SOS) nicht ausreichend ist. Daher wurde ein stärkerer semantischer Äquivalenzbegriff eingeführt, der auch die Äquivalenz unerreichbarer Zustände berücksichtigt [Lei06].

Neben der Formalisierung der Metamodelle ist es notwendig, auch die An-

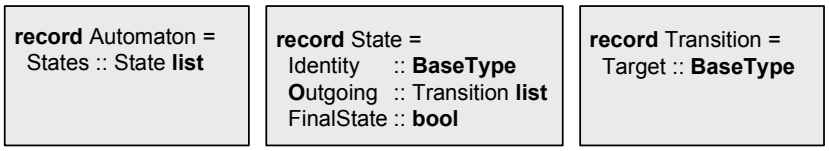
[2] auch als *small-step operational semantics* bekannt



(a) Ausschnitt aus dem Metamodell für I/O-Automaten



(b) Modifiziertes Metamodell



(c) Abbildung auf Datentypen in Isabelle/HOL

Abbildung 6.4: Formalisierung von Metamodellen als induktive Datentypen

wendung der TGG-Regeln zu formalisieren. Aufgrund der Tatsache, dass die TGG-Regeln zur Modelltransformation als einfache Graphersetzungsregeln aufgefasst werden können, wurde zunächst eine Formalisierung der Regelanwendungen als Graphersetzung auf einem zusammenhängenden Graphen durchgeführt. Dabei hat sich jedoch gezeigt, dass eine solche Formalisierung nicht praktikabel ist [Lei06]. Daher wird zur Formalisierung eine TGG-Regel nicht als Transformation auf einem Graphen interpretiert, sondern als Paar zusammengehöriger Produktionen, mit denen zwei Modelle simultan erzeugt werden. Diese Interpretation der TGG-Regeln stimmt mit der in Abschnitt 3.2.1 vorgestellten Semantik überein.

Zur Formalisierung der Produktionen wird für jede Produktion einer TGG-Regel ein Operator – der sogenannte *Modifikator* – auf dem Quell- und Zielmodell definiert und die Anwendung der TGG-Regeln als simultane Anwendung von zueinander korrespondierenden Modifikatoren formalisiert. Für die in Abbildung 6.5 gezeigte TGG-Regel, in der ein Zustand zum Automaten bzw. eine CASE-Anweisung zu einem SPS-Programm hinzugefügt wird, können die Modifikatoren wie folgt definiert werden[3]:

[3]Der Isabelle/HOL-Operator (⟦... := ...⟧) wird verwendet, um ein Attribut eines zusam-

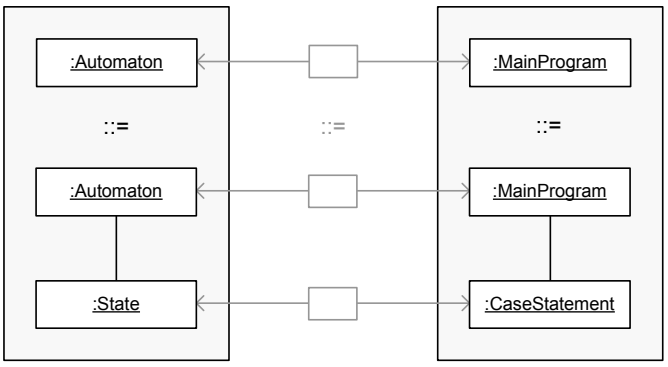


Abbildung 6.5: Interpretation einer TGG-Regel als zusammengehöriges Paar zweier Produktionen

$$A \oplus s \equiv A(\!| \text{States} := (\text{States } A) \cdot s |\!) \tag{6.1}$$
$$P \oplus c \equiv P(\!| \text{MainProgram} := c \cdot (\text{MainProgram } P) |\!) \tag{6.2}$$

Für den Nachweis der Korrektheit einer TGG-Regel genügt es zu beweisen, dass die paarweise Anwendung der Modifikatoren die semantische Äquivalenz der Modelle nicht zerstört. Für die Produktionen der TGG-Regel aus Abbildung 6.5 beispielsweise muss also gezeigt werden, dass beim Hinzufügen eines Zustands zu einem Automaten und einer CASE-Anweisung zu einem SPS-Programm die semantische Äquivalenz zwischen dem Automaten und dem SPS-Programm erhalten bleibt:

$$A \approx P \implies (\ A \oplus s\ ) \approx (\ P \oplus \texttt{State2Case}(s)\ ) \tag{6.3}$$

Bei dem angewandtem Beweisprinzip handelt es sich um einen Induktionsbeweis, der mit Hilfe des Theorembeweisers Isabelle/HOL interaktiv vom Benutzer durchgeführt wird. Eine wichtige Grundvoraussetzung ist die Abbildung der Problemstellung in die formale Sprache von Isabelle/HOL. Der Umfang dieser Formalisierung hängt von der Größe der gegebenen Metamodelle, der zugrundeliegenden Semantik sowie der Anzahl der spezifizierten TGG-Regeln ab. Zur Formalisierung der I/O-Automaten, des SPS-Codes und der TGG-Regeln wurden zusammen mit der anschließenden Beweisführung ca. 1.500 Codezeilen in der Isabelle/HOL-Notation benötigt.

mengesetzten Record-Datentyps zu aktualisieren. Der Operator · fügt Elemente zu einer Liste hinzu.

Der Nachteil dieses Ansatzes gegenüber dem automatischen Checker-Ansatz liegt aufgrund des hohen Beweisaufwands auf der Hand. Allerdings wird im Gegensatz zum Checker-Ansatz die *allgemeine semantische Korrektheit* von Transformationsregeln *bewiesen*. Wird ein solcher Beweis für eine Menge von Transformationsregeln einmal durchgeführt, so gilt er für *jedes* gültige Modell, das übersetzt wird. Für die Übersetzung eines I/O-Automaten in SPS-Code beispielsweise bedeutet dies, dass durch die spezifizierte Modelltransformation tatsächlich immer semantisch äquivalenter SPS-Code erzeugt wird, der die spezifizierten Automaten implementiert. Ein solcher Beweis ist insbesondere in sicherheitskritischen Anwendungen von einem sehr hohen Nutzen, da gewährleistet werden muss, dass der generierte Code korrekt ist und die im Modell überprüften Eigenschaften auch im Code eingehalten werden.

6.3 Zusammenfassung

Die Validierung und Verifikation von Modelltransformationen ist erst mit dem Aufkommen und der Verfügbarkeit verschiedener Techniken stärker in den Mittelpunkt der Forschung gerückt. Daher sind die hierzu notwendigen Methoden derzeit noch Gegenstand der Forschung und nicht sehr weit fortgeschritten. Wir haben uns auf die Modelltransformation beschränkt, da die Modelltransformation ein wichtiger Bestandteil bzw. die Grundlage der in dieser Arbeit vorgestellten Technik zur Modellsynchronisation ist.

Der erste Teil dieses Kaptitels war der Validierung gewidmet. Hierbei haben wir existierende Ansätze zur Überprüfung der syntaktischen Korrektheit von Modelltransformationen vorgestellt. Dabei haben wir argumentiert, dass sowohl die syntkatische Korrektheit einzelner Regeln als auch deren Zusammenspiel überprüft werden müssen. Während die syntaktische Korrektheit bzw. einige hierzu notwendigen Kriterien häufig durch statische Analysen der einzelnen Regeln überprüft werden können, kann das Zusammenspiel der Regeln durch die Ausführung einer Modelltransformation validiert werden, indem das Resultat der Modelltransformation mit dem erwarteten Ergebniss verglichen wird. Ein solcher Test kann automatisiert durchgeführt werden. Hierzu haben wir existierende Ansätze vorgestellt und diskutiert, wie diese auf TGGs übertragen werden können.

Im zweiten Teil dieses Kapitels haben wir einige Ansätze zur formalen Verifikation der semantischen Korrektheit von Modelltransformationen vorgestellt. Zusätzlich haben wir einen Ansatz präsentiert, mit dem allgemein

bewiesen werden kann, dass die spezifizierten Transformationsregeln für jede gültige Eingabe eine semantisch äquivalente Ausgabe erzeugen. Die formale Beweistechnik wurde im Rahmen dieser Dissertation auf TGGs übertragen und zum Nachweis der Korrektheit eines Codegenerators eingesetzt.

Kapitel 7

Werkzeugunterstützung

In diesem Kapitel wird die im Rahmen dieser Arbeit umgesetzte Werkzeugunterstützung vorgestellt. Mit Hilfe dieser Werkzeugunterstützung können Werkzeuge zur Modelltransformation, Modellintegration und Modellsynchronisation gemäß der zuvor vorgestellten Konzepte modellbasiert entwickelt werden. Die Werkzeugunterstützung wurde auf Basis der Entwicklungsumgebungen Eclipse[1] und Fujaba[2] realisiert. Die Festlegung auf Fujaba erfolgte aufgrund des dort bereits implementierten und bewährten Graphersetzungssystems. Dabei handelt es sich um einen generativen Ansatz, bei dem aus erweiterten Graphersetzungsregeln, den sogenannten Storydiagrammen, Java-Code erzeugt wird. Die Umsetzung in Eclipse hat sich insbesondere aufgrund der dort verfügbaren Entwicklungswerkzeuge für Java angeboten. Durch die Integration von Fujaba und Eclipse zu dem Werkzeug Fujaba4Eclipse kann der generierte Java-Code direkt in Eclipse übersetzt und ausgeführt werden.

7.1 Architektur

In diesem Abschnitt verschaffen wir uns zunächst einen Überblick über die Architektur der entstandenen Werkzeugunterstützung. Die Werkzeugunterstützung wurde auf verschiedene Komponenten aufgeteilt und durch Plug-ins realisiert. Einen Überblick über die wichtigsten Komponenten und ihre Abhängigkeiten untereinander zeigt die Abbildung 7.1. Dabei kann zwischen Komponenten zur Spezifikation (`tggeditor`, `tggeditor4eclipse` und `tgggeneration`) und Komponenten zur Ausführung von TGG-Regeln (`mote`, `morten` und `morten4eclipse`) unterschieden werden.

[1] http://www.eclipse.org
[2] http://www.fujaba.de

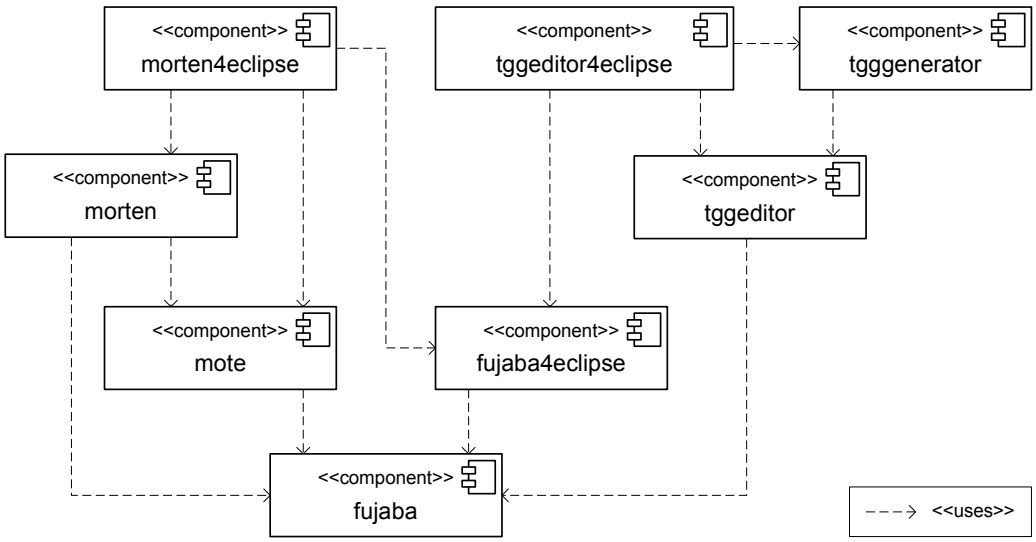


Abbildung 7.1: Komponenten der Werkzeugunterstützung

Die Basis der realisierten Werkzeugunterstützung bilden die beiden Komponenten `fujaba` und `fujaba4eclipse`. Die Komponente `fujaba` stellt die zur Spezifikation von Metamodellen benötigten Klassendiagramme bereit. Zusätzlich enthält sie die notwendigen Editoren zur Modellierung von Storydiagrammen sowie einen Codegenerator, der aus Klassen- und Storydiagrammen ausführbaren Java-Code erzeugt [FNT98, FNTZ98, NNZ00]. Die Komponente `fujaba4eclipse` integriert die Funktionalität von Fujaba in die Eclipse-Entwicklungsumgebung, indem sie entsprechende Benutzerschnittstellen (Menüs, Werkzeugleisten, Sichten, Editoren, etc.) bereitstellt.

Der Editor zur Spezifikation von TGG-Regeln wurde in der Komponente `tggeditor` realisiert. Die Integration des Editors in Fujaba4Eclipse mit den dafür notwendigen Benutzerschnittstellen erfolgt in der Komponente `tggeditor4eclipse`. Die Generierung von Storydiagrammen aus TGG-Regeln ist in der Komponente `tgggenerator` implementiert. Nach der automatischen Generierung der Storydiagramme können diese weiter verfeinert und beispielsweise um zusätzliche Abfragen und Aktionen erweitert werden. Darüber hinaus ist die Komponente so aufgebaut, dass die Generierung von Storydiagrammen austauschbar ist bzw. durch neue Generierungsvarianten erweitert werden kann. Beim Ausführen der Entwicklungsumgebung erkennt die Komponente neu hinzugefügte Generierungsvarianten und bietet diese dem Benutzer zur Auswahl an.

Die Komponente `mote`[3] enthält das Rahmenwerk mit den Algorithmen zur Ausführung einer Modelltransformation, Modellintegration und Modellsynchronisation. Dieses Rahmenwerk wird durch die Komponente `morten`[4] in FUJABA integriert. Die Anbindung an FUJABA4ECLIPSE erfolgt hingegen in der Komponente `morten4eclipse`, die eine entsprechende Benutzerschnittstelle zur Verfügung stellt. Die Benutzerschnittstelle der dadurch entstandenen Entwicklungsumgebung für Modellsynchronisationen wird im nachfolgenden Abschnitt genauer vorgestellt.

7.2 Entwicklungsumgebung

Abbildung 7.2 zeigt die FUJABA4ECLIPSE-Entwicklungsumgebung. Im Aufbau der Benutzeroberfläche ähnelt sie sehr vielen anderen herkömmlichen Entwicklungsumgebungen. So enthält die Benutzeroberfläche im linken, oberen Bereich einen Projektmanager und im rechten Teil des Anwendungsfensters einen Arbeitsbereich zur Einbettung graphischer und textueller Editoren. Darunter befindet sich auf der linken Seite eine Übersicht (*Outline-View*) und auf der rechten Seite verschiedene andere Sichten, die über Karteireiter (engl. Tabs) aktiviert und z. B. zur Anzeige von Warnung und Fehlermeldungen (*Problems-View*) oder Eigenschaften der im Editor ausgewählten Elemente (*Properties-View*) eingesetzt werden.

Die einzelnen Sichten und Editoren lassen sich innerhalb der Entwicklungsumgebung frei anordnen und zu sogenannten *Perspektiven* zusammenfassen. Dadurch kann die Entwicklungsumgebung an die individuellen Bedürfnisse eines Benutzers angepasst bzw. auf die Durchführung einer bestimmten Aufgabe optimiert werden. In Abbildung 7.2 ist die Perspektive von FUJABA4ECLIPSE dargestellt.

7.2.1 Spezifikation

Um mit den in dieser Arbeit vorgestellten Konzepten ein Werkzeug zur Modellsynchronisation zu entwickeln, müssen zunächst die beteiligten Metamodelle spezifiziert werden. Dies erfolgt in FUJABA4ECLIPSE mit Hilfe von

[3] Abkürzung für **Mo**del **T**ransformation **E**ngine. Der Name ist historisch dadurch entstanden, dass diese Komponente zunächst nur für die Modelltransformation vorgesehen war und erst später um die Modellsynchronisation erweitert wurde.

[4] Abkürzung für **Mo**del **R**ound-**T**rip **En**gineering

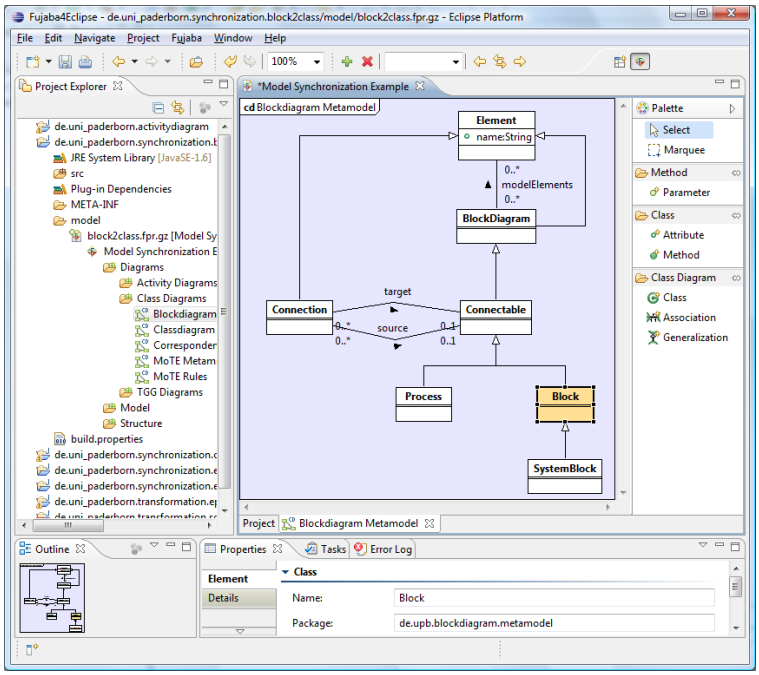


Abbildung 7.2: Die Entwicklungsumgebung Fujaba4Eclipse

UML-Klassendiagrammen. In Abbildung 7.2 ist das in Fujaba verwendete Metamodell für Blockdiagramme aus dem Beispiel dieser Arbeit dargestellt. Neben diesem Metamodell werden – wie zuvor in Kapitel 3 erläutert – noch ein Metamodell für Klassendiagramme und ein Metamodell für das Korrespondenzmodell benötigt. Auch diese Metamodelle werden mit UML-Klassendiagrammen spezifiziert.

Sind die Metamodelle erstellt, so können auf dieser Grundlage Korrespondenzregeln spezifiziert werden. Hierzu wird der graphische Editor für Tripel-Graph-Grammatiken verwendet. Abbildung 7.3 zeigt den Editor mit einer TGG-Regel aus unserem Beispiel zur Synchronisation von Block- und Klassendiagrammen. Mit dem Editor können TGG-Regeln erstellt und bearbeitet werden. Hierzu befindet sich auf der rechten Seite des Editors eine *Werkzeugpalette*, mit der vorhandene Elemente selektiert (*Select* und *Marque*) oder neue Elemente (*Object, Link, Constraint, Assertion*) erzeugt werden können. So können mit Hilfe der Werkzeugpalette zum Beispiel neue Objekte, Verbindungen, Bedingungen und Attributzuweisungen zu einer TGG-Regel hinzugefügt werden. Nach der Erstellung der Elemente können die Eigenschaften eines im Editor selektierten Elements in der Properties-View an-

gezeigt und dort bearbeitet werden. Beispielsweise können in der Properties-View sowohl der Typ als auch der Name eines Objektes modifiziert werden.

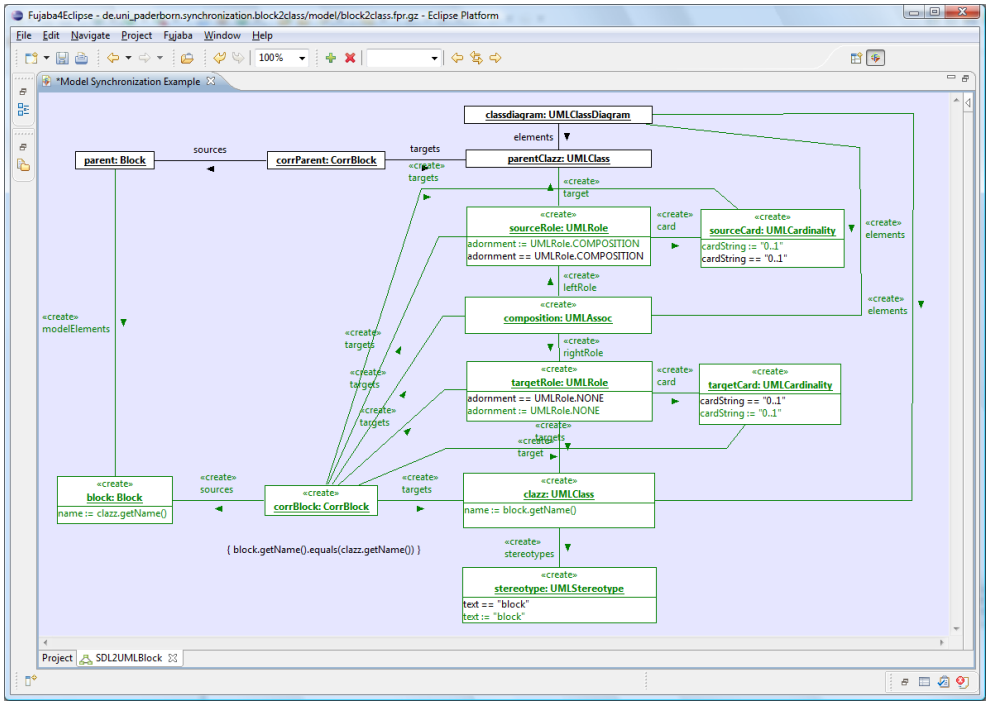


Abbildung 7.3: TGG-Editor

Zusätzlich zur Properties-View kann eine Outline-View eingeblendet werden. Die Outline-View stellt den Inhalt des graphischen Editors verkleinert dar und hilft, den Überblick bei sehr großen TGG-Regeln zu behalten. Um die TGG-Regel in Abbildung 7.3 vollständig darzustellen, wurden sowohl die Properties- als auch die Outline-View ausgeblendet und das Fenster des Editors vergrößert. Die beiden Sichten entsprechen den bereits in der Abbildung 7.2 gezeigten Sichten.

Der Editor ist als Plug-in implementiert und stellt die Konformität der TGG-Regeln zu den zugrunde liegenden Metamodellen sicher. So ist es beispielsweise nicht möglich, den Objekten einen Typ zuzuordnen, der nicht zuvor durch eine Klasse im Metamodell spezifiziert worden ist. Ebenso können nur Verbindungen zwischen Objekten erstellt werden, wenn auch die zugeordneten Objekttypen über eine entsprechende Assoziation miteinander verbunden sind.

Zusätzlich zu dem Editor existiert ein Plug-in, mit dem die in Abschnitt 4.2

beschriebene Regelsynthese durchgeführt werden kann. Die damit synthetisierten TGG-Regeln können mit dem Editor bearbeitet und weiter verfeinert werden. Sind alle TGG-Regeln erstellt, so müssen die TGG-Regeln zu einem Katalog zusammengefasst werden. Der Katalog wird zur Parametrisierung des Rahmenwerks verwendet, um auf dieser Grundlage die spezifizierte Modellsynchronisation auszuführen.

7.2.2 Generierung eines Regelkatalogs

Die Generierung eines Regelkatalogs erfolgt in mehreren Schritten. Zunächst wird zu jeder TGG-Regel eine eigene Java-Klasse erzeugt. Dabei wird der Klasse für jede Richtung der Modellsynchronisation eine eigene Methode hinzugefügt. Zusätzlich generieren wir eine Methode, die lediglich der Überprüfung der Korrespondenzbeziehungen dient und damit zur Modellintegration eingesetzt werden kann. Damit erhalten wir insgesamt drei Methoden: zwei Methoden für die Modelltransformation und Modellsynchronisation (eine Methode vom Quell- zum Zielmodell und eine Methode für die umgekehrte Richtung) sowie eine Methode zur Modellintegration. Das Verhalten dieser Methoden wird festgelegt, indem aus der zugehörige TGG-Regel entsprechende Storydiagramme generiert werden.

Zur Generierung der Storydiagramme muss der Benutzer die TGG-Regeln im Projektmanager selektieren. Auf dieser Auswahl ruft der Benutzer über einen Rechtsklick ein Kontextmenü auf und selektiert dort den Eintrag `Generate Story Diagrams`. In dem sich daraufhin öffnenden Dialog muss der Benutzer eine Generierungsstrategie wählen. Die im Rahmen dieser Arbeit umgesetzte Generierungsstrategie für Modelltransformation, Modellintegration und Modellsynchronisation ist unter dem Menüeintrag `MoTE (Advanced)` zu finden (vgl. Abbildung 7.4). Nach der Generierung der Storydiagramme können diese noch weiter verfeinert und beispielsweise um zusätzliche Seiteneffekte erweitert werden – was aber in den meisten Fällen nicht notwendig ist.

In einem zweiten Schritt muss aus den Storydiagrammen Java-Code generiert werden. Hierzu wird der in Fujaba integrierte Codegenerator verwendet. Der Codegenerator wird in Fujaba4Eclipse ebenfalls über einen Eintrag im Kontextmenü gestartet (vgl. Abbildung 7.5). In dem dazugehörigen Code-Export-Wizard wählt der Benutzer ein Klassendiagramm oder einzelne Klassen, aus denen dann der Java-Code generiert wird. Zusätzlich muss der Benutzer angeben, in welchem Projektverzeichnis der generierte Java-Code gespeichert wird. Nach der Generierung wird der Code durch den Eclipse-

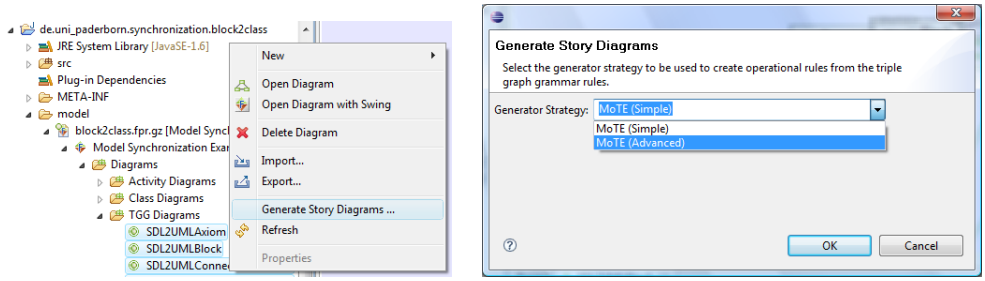


Abbildung 7.4: Generierung der Storydiagramme

Compiler automatisch in ausführbaren Bytecode übersetzt. Die kompilierten Klassen repräsentieren die ausführbaren TGG-Regeln.

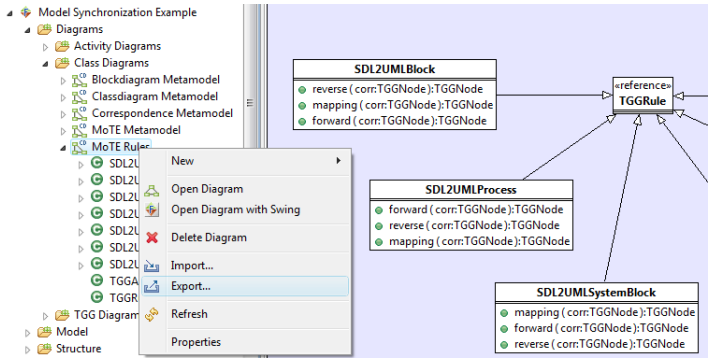


Abbildung 7.5: Start der Codegenerierung

Im letzten Schritt müssen die ausführbaren TGG-Regeln in einem Jar-Archiv zu einem Katalog gebündelt werden. Hierzu wird der in Eclipse integrierte Jar-Packager verwendet (vgl. Abbildung 7.6). Mit Hilfe dieses Wizards kann der Benutzer die Dateien angeben, die in dem Jar-Archiv enthalten sein sollen.

Zusätzlich zu den kompilierten Klassen muss das Jar-Archiv eine Konfigurationsdatei enthalten, in der die verfügbaren TGG-Regeln sowie zusätzlich benötigte Plug-ins und Bibliotheken aufgelistet sind. In Abbildung 7.7 ist ein Ausschnitt einer solchen Konfigurationsdatei für unser Beispiel angegeben.

Bei der Konfigurationsdatei handelt es sich um eine XML-Datei, in der unter anderem beschrieben ist, welche TGG-Regeln verfügbar sind und auf welchen Typen von Korrespondenzknoten die TGG-Regeln ausgeführt werden können. In der zuvor gezeigten Beispielkonfiguration werden unter dem

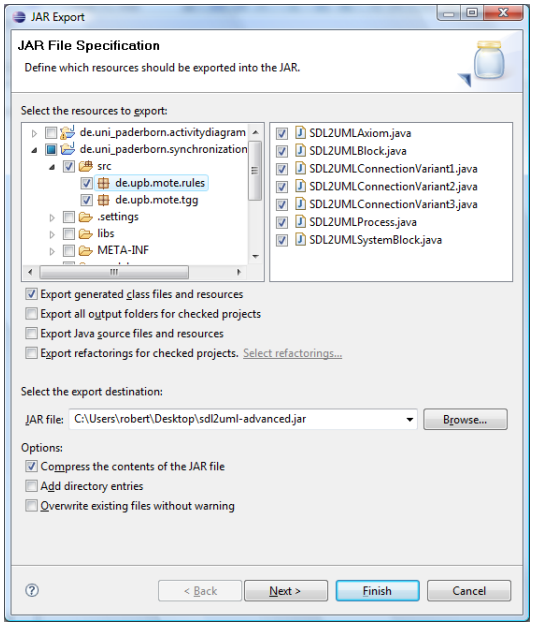


Abbildung 7.6: Wizard zur Erstellung des Jar-Archivs

```
<?xml version="1.0" standalone="yes"?>

<configuration>
  <triggertable>
    <entry  trigger=""
            rule="de.upb.mote.rules.SDL2UMLAxiom"/>
    <entry  trigger="de.upb.mote.tgg.CorrAxiom"
            rule="de.upb.mote.rules.SDL2UMLSystemBlock"/>
    <entry  trigger="de.upb.mote.tgg.CorrSystem"
            rule="de.upb.mote.rules.SDL2UMLBlock"/>
    <entry  trigger="de.upb.mote.tgg.CorrBlock"
            rule="de.upb.mote.rules.SDL2UMLBlock"/>
    ...
  </triggertable>
  <dependencies>
    <plugin id="de.uni_paderborn.example.blockdiagram4eclipse"/>
  </dependencies>
</configuration>
```

Abbildung 7.7: Ausschnitt aus einer Konfigurationsdatei

Element `triggertable` die verfügbaren TGG-Regeln in separaten Einträgen (`entry`) aufgelistet. Das Attribut `trigger` eines solchen Eintrags bezeichnet den Typ des Korrespondenzknotens, auf dem die unter dem Attribut `rule` genannte TGG-Regel potentiell angewendet werden kann. Hier sind auch Mehrfachnennungen möglich, so dass eine TGG-Regel durchaus auf verschiedene Typen von Korrespondenzknoten geprüft werden kann. Ist das `trigger` Attribut hingegen nicht weiter spezifiziert, so handelt es sich bei dem Eintrag um ein Axiom. Die Document Type Definition (DTD) der Konfigurationsdatei ist im Anhang B angegeben.

Das Jar-Archiv repräsentiert einen Katalog mit ausführbaren Regeln einer Tripel-Graph-Grammatik. Ist dieser Katalog erstellt und verfügbar, so kann damit eine Modelltransformation, eine Modellintegration oder eine Modellsynchronisation ausgeführt werden.

7.2.3 Ausführung

Damit eine Modelltransformation, eine Modellintegration oder eine Modellsynchronisation in Fujaba4Eclipse ausgeführt werden kann, müssen die beteiligten Modelle zunächst geladen werden. Dies geschieht automatisch, indem das oder die Projekte, die diese Modelle enthalten, in Fujaba4Eclipse geöffnet werden. Sobald die beteiligten Modelle geladen sind, kann der Benutzer eine neue Synchronisationsaufgabe durch die Auswahl einer hierfür vorgesehenen Schaltfläche der Werkzeugleiste anlegen. Die Werkzeugleiste mit den Erklärungen zu den dort verfügbaren Schaltflächen ist in der Abbildung 7.8 zu sehen.

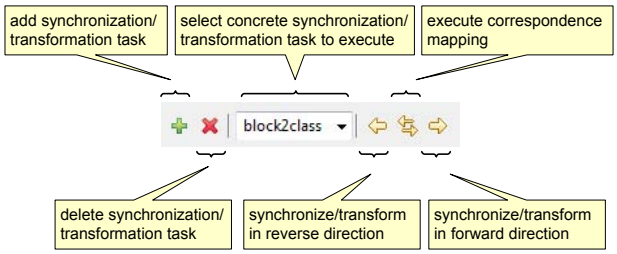


Abbildung 7.8: Werkzeugleiste zur Modellsynchronisation

Beim Anlegen einer neuen Synchronisationsaufgabe wird ein sogenannter *Model-Synchronization-Wizard* gestartet. Der Model-Synchronization-Wizard besteht aus zwei Dialogen, die nacheinander zusätzliche Eingaben

vom Benutzer abfragen, um die Modellsynchronisation erfolgreich initialisieren zu können.

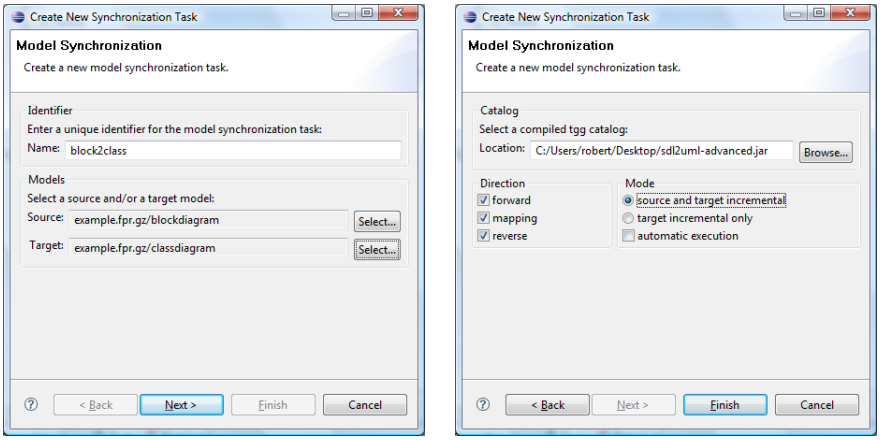


Abbildung 7.9: Synchronisierungs-Wizard

Die beiden Dialoge des Model-Synchronization-Wizards sind in Abbildung 7.9 dargestellt. Im ersten Dialog muss der Benutzer zunächst einen Namen für die Modellsynchronisation eingeben. Damit die hier vorgestellte Werkzeugunterstützung mehrere Synchronisationsaufgaben innerhalb der Entwicklungsumgebung unterstützen kann, sollte dieser Name eindeutig sein, da dieser Name in der Werkzeugleiste aus Abbildung 7.8 dem Benutzer zur Auswahl angeboten wird. Die dort vom Benutzer durchgeführte Auswahl bestimmt, welche Modellsynchronisation ausgeführt wird.

Anschließend muss der Benutzer die zu synchronisierenden Modelle auswählen. Hat der Benutzer zwei Modelle angegeben, so können die Elemente der Modelle durch eine Modellintegration zunächst zueinander in Beziehung gesetzt werden. Ebenso ist es aber auch möglich, die beiden Modelle in eine der beiden Richtungen sofort miteinander zu synchronisieren.

In dem Fall, dass zunächst nur ein Modell vorhanden ist und das zweite Modell durch eine Modelltransformation erzeugt werden soll, reicht es jedoch aus, nur das zu transformierende Modell anzugeben. Ist beispielsweise nur das Quellmodell vorhanden, so wird das Zielmodell durch eine Vorwärtstransformation erzeugt. Steht hingegen nur das Zielmodell zur Verfügung, wird das Quellmodell durch eine Rückwärtstransformation aus dem Zielmodell gewonnen. In beiden Fällen können nach der Transformation die Modelle wie gewohnt miteinander synchronisiert werden.

Bevor eine Modelltransformation, Modellintegration oder Modellsynchronisation ausgeführt werden kann, muss der Benutzer noch den Regelkatalog angeben, d. h., das Jar-Archiv, in dem die ausführbaren TGG-Regeln hinterlegt sind. Darüber hinaus kann der Benutzer die Richtung der Modellsynchronisation einschränken und beispielsweise nur eine Modelltransformation oder Modellsynchronisation in Vorwärtsrichtung erlauben. Zusätzlich kann er einstellen, ob die Modellsynchronisation inkrementell oder batch-orientiert und ob die Modellsynchronisation manuell angestoßen werden muss oder ob sie automatisch nach jeder Modelländerung ausgeführt wird. Diese Einstellungen können in dem zweiten Dialog vorgenommen werden, der auf der rechten Seite der Abbildung 7.9 zu sehen ist.

Hat der Benutzer alle Eingaben vorgenommen, erfolgt eine Initialisierung des Rahmenwerks mit dem Regelkatalog und dem (oder den) Modell(en). Sofern der Benutzer keine automatische Modellsynchronisation eingestellt hat, kann er die Modelltransformation, Modellintegration oder Modellsynchronisation über die entsprechenden Schaltflächen der Werkzeugleiste (vgl. Abbildung 7.8) manuell auslösen. Hierbei erfolgt eine Modelltransformation nur in dem Fall, dass die Modellsynchronisation zum ersten Mal ausgeführt und noch kein zweites Modell vorhanden ist. Sind beide Modelle gegeben, so wird je nach gewählter Schaltfläche entweder nur eine Modellintegration oder eine Modellsynchronisation durchgeführt (sofern keine automatische Synchronisation gewählt wurde). In Abbildung 7.10 ist das Blockdiagramm sowie das damit synchronisierte Klassendiagramm zu sehen. Jede nachfolgende Ausführung in eine der Richtungen führt zu einer Modellsynchronisation.

Nach einer Modelltransformation, Modellintegration und Modellsynchronisation kann der Benutzer die sogenannte *MoRTEn-View* aktivieren. In der MoRTEn-View wird das Korrespondenzmodell in einer Baumstruktur dargestellt. In Abbildung 7.10 ist diese Sicht unter den Diagrammen zu sehen. Jeder Eintrag in dieser Ansicht zeigt neben der angewendeten TGG-Regel den bei der Anwendung erzeugten Korrespondenzknotentyp sowie seine Höhe in der Hierarchie. Links und rechts von dem Korrespondenzmodell werden bei einem selektierten Eintrag die mit dem Eintrag assoziierten Modellelemente angezeigt. Diese Ansicht hat keine weitere Funktionalität und ist lediglich beim Testen der Modellsynchronisation hilfreich, indem sie zueinander korrespondierende Modellelemente darstellt und damit eine Nachverfolgbarkeit einer Modelltransformation, Modellintegration und Modellsynchronisation – zumindest rudimentär – möglich macht. Um die Nachverfolgbarkeit einer Modelltransformation, Modellintegration und Modellsynchronisation zu verbessern, könnte dieser Ansicht weitere Funktionalität hinzugefügt werden.

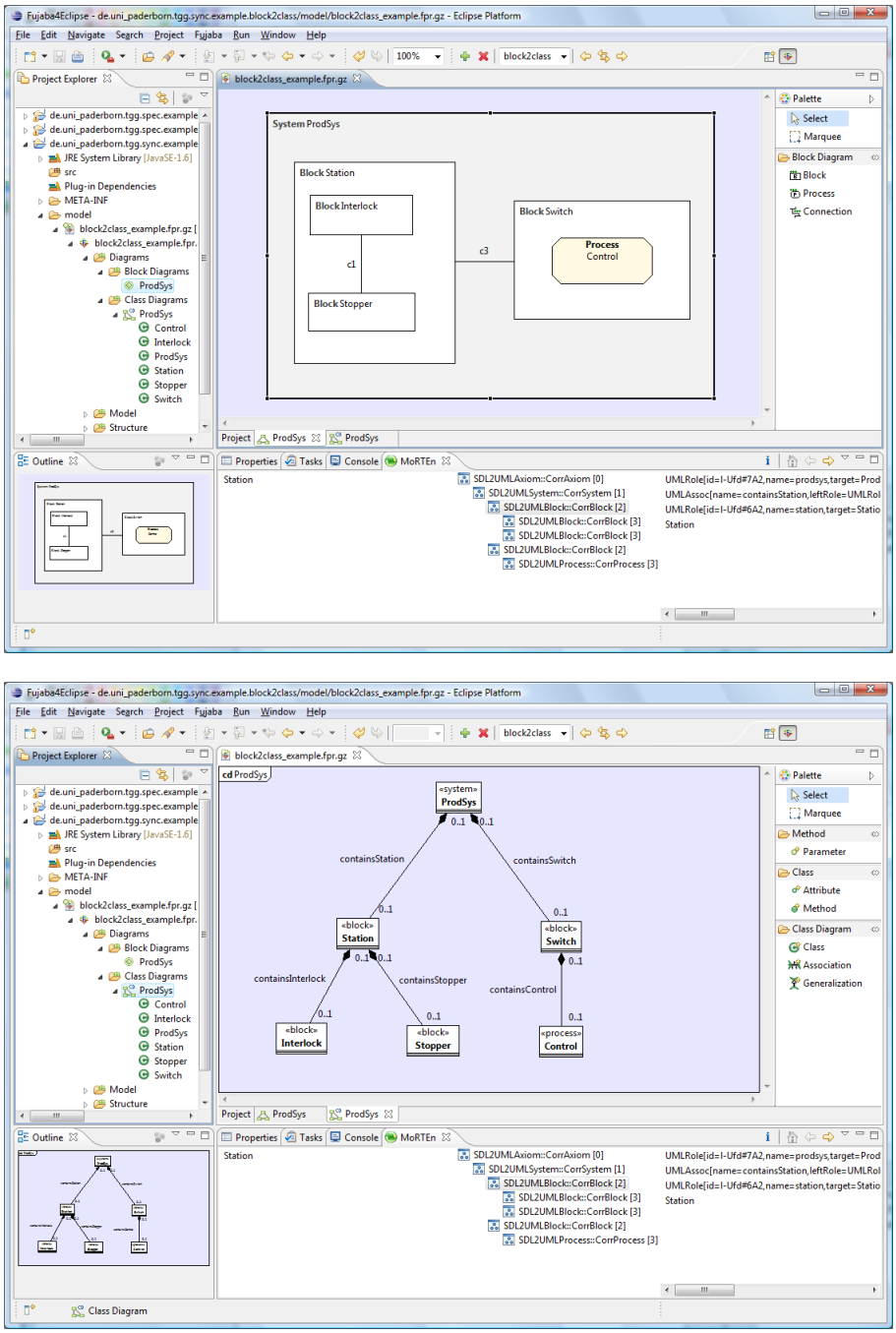


Abbildung 7.10: Modellsynchronisation zwischen einem Block- und einem Klassendiagramm

Dies stand jedoch nicht im Fokus dieser Arbeit.

Die in dieser Arbeit umgesetzte Werkzeugunterstützung kann einerseits dazu verwendet werden, um eine Modellsynchronisation zu spezifizieren und sie direkt in Fujaba4Eclipse zu verwenden. Andererseits kann eine Spezifikation mit Fujaba4Eclipse erstellt, getestet und der daraus generierte Regelkatalog zusammen mit dem Rahmenwerk zur Modellsynchronisation in ein beliebiges Java-basiertes Werkzeug integriert werden. Eine solche Integration wurde zum Beispiel im Rahmen des Projekts MATE durchgeführt [GMW06]. In diesem Projekt wurden Matlab/Simulink-Modelle automatisch in sogenannte Musterspezifikationen [NSW+02] mit der hier vorgestellten Werkzeugunterstützung übersetzt. Aufgrund der Tatsache, dass die wenigsten Werkzeuge tatsächlich ein Fujaba-konformes Metamodell besitzen, mussten hierzu geeignete Werkzeug- bzw. Modelladapter erstellt werden. Die Werkzeug- und Modelladapter zur Integration in andere Werkzeuge werden im nachfolgenden Abschnitt behandelt.

7.3 Werkzeug- und Modelladapter

Damit der in dieser Arbeit vorgestellte Ansatz korrekt funktionieren und kompilierbarer Code aus den Storydiagrammen mit Fujaba erzeugt werden kann, müssen die verwendeten Metamodelle Fujaba-konform implementiert sein, das heißt, dass die Implementierung dieser Metamodelle sich an einige von Fujaba vorgegebene Implementierungsregeln halten muss. Die Einhaltung dieser Vorgaben ist wichtig, um zwischen Modellelementen in einem Modell navigieren zu können, auf Modellelemente zugreifen und sie verändern zu können, als auch um neue Modellelemente erstellen zu können. Diese Operationen sind fundamental, damit die implementierten Algorithmen zur Modelltransformation, Modellintegration und Modellsynchronisation korrekt ausgeführt werden.

In den Fällen, in denen das Metamodell mit Fujaba spezifiziert und die Implementierung automatisch generiert wurde, sind die Anforderungen automatisch erfüllt. Die meisten Modellierungswerkzeuge erfüllen diese Anforderungen allerdings nicht und bieten kein Fujaba-konformes Metamodell an. Leider ist es nicht möglich, die Metamodelle der Werkzeuge beziehungsweise ihre Implementierungen einfach auszutauschen. In den meisten Fällen ist es auch nicht möglich, die Codegenerierung aus den Storydiagrammen daran anzupassen, da sich die Implementierungen der Metamodelle oft an keinen Standard halten. Darüber hinaus sind die Metamodelle der Werkzeuge gar

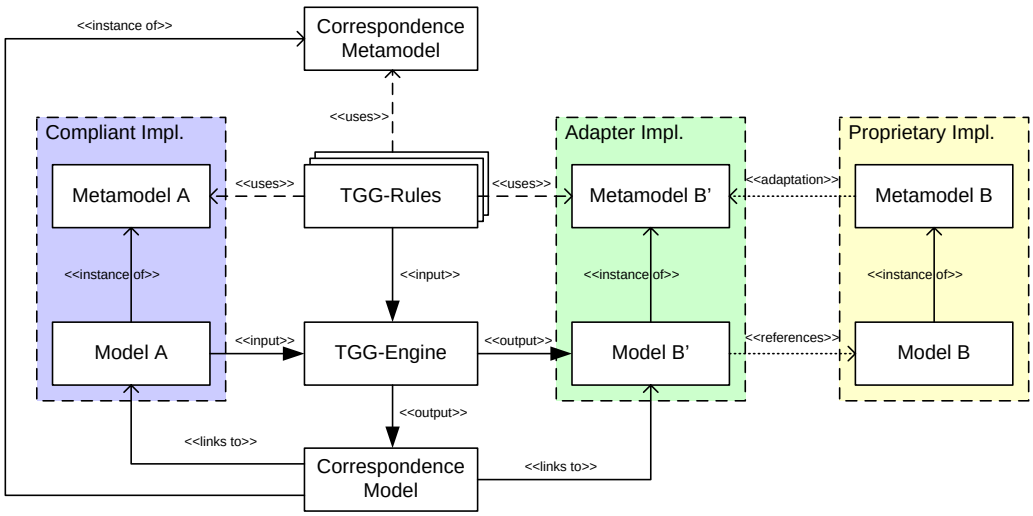


Abbildung 7.11: Überblick zu Werkzeug- und Modelladaptern

nicht oder nur schlecht dokumentiert. In Einzelfällen wird lediglich eine einfache Schnittstelle (engl. Application Programming Interface, API) zur Verfügung gestellt. Im schlimmsten Fall, das heißt, wenn die Metamodelle nicht verfügbar sind, muss ein konzeptionelles Metamodell aus den verfügbaren Artefakten, wie zum Beispiel der API-Schnittstelle, manuell zurück gewonnen werden.

Damit die in dieser Arbeit vorgestellten Anwendungen trotzdem in anderen Werkzeugumgebungen durchgeführt werden können, verwenden wir das sogenannten Adapter-Entwurfsmuster [GHJV94]. Dieses Muster erlaubt es, eine bereits vorhandene Schnittstelle einer Klasse an eine andere Schnittstelle anzupassen.

Abbildung 7.11 zeigt einen Überblick zu dem hier vorgeschlagenen Adapteransatz. In der oberen Hälfte der Abbildung ist die Beziehung zwischen den spezifizierten TGG-Regeln und den beteiligten Metamodellen zu sehen. Dabei wird in den Regeln jedoch nicht direkt das Metamodell *B* referenziert, sondern ein dazwischen geschaltetes Metamodell *B'*. In der unteren Hälfte wird daher eine Modelltransformation gezeigt, in der die TGG-Engine das Modell *A* in das Modell *B* nur indirekt transformiert, indem sie das Modell *B'* erzeugt.

In diesem Szenario besitzt das Metamodell *A* eine zu Fujaba konforme Implementierung. Daher kann die TGG-Engine auf die Elemente des Modells *A* direkt zugreifen. Im Gegensatz dazu entspricht die Implementierung

des Metamodells *B* nicht den Anforderungen. Um dennoch einen Zugriff der TGG-Engine auf die Modellelemente zu ermöglichen, wurde ein Metamodelladapter *B'* implementiert. Diese Implementierung ist konform zu Fujaba und ermöglicht einen Zugriff auf die Modellelemente des Modells *B*.

Zur Implementierung eines Fujaba-konformen Modelladapters kann Fujaba herangezogen werden. In einem ersten Schritt wird das Metamodell als Klassendiagramm in Fujaba spezifiziert. Aus dieser Spezifikation kann die Implementierung des Metamodells automatisch durch die in Fujaba verfügbare Codegenerierung erzeugt werden. Durch die automatische Generierung besitzt die Implementierung bereits die nötigen Schnittstellen, um mit der TGG-Engine zusammen arbeiten zu können. Allerdings wird noch nicht auf das proprietäre Modell des Werkzeugs zugegriffen.

Um einen Zugriff auf das Werkzeugmodell zu ermöglichen, muss der generierte Code manuell erweitert werden. Bei dieser Erweiterung müssen die generierten Attribute aus dem Code entfernt und die Zugriffsoperationen für diese Attribute so angepasst werden, dass sie die API-Schnittstelle des proprietären Modells benutzen. Dadurch verändern wir zwar die Methodenimplementierungen, aber nicht die Methodensignaturen. Damit bleibt die Schnittstelle des Modelladapters weiterhin zu Fujaba konform und kann durch die TGG-Engine genutzt werden. Die Zugriffe auf den Modelladapter werden nun an das proprietäre Modell des Werkzeugs weiterdelegiert.

Der hier beschriebene Ansatz wurde in verschiedenen Projekten erfolgreich angewendet, zum Beispiel zur Adaption eines Metamodells für Zustandsautomaten oder des Metamodells für Matlab/Simulink [GMW06]. Die Implementierungen der Modelladapter arbeiten zustandslos und mit einer verzögerten Initialisierung, das heißt, die einzelnen Adapterobjekte werden erst bei Bedarf erzeugt. Darüber hinaus werden einmal erzeugte Adapterobjekte in einer Liste verwaltet, so dass sie wiederverwendet werden können, sobald ein erneuter Zugriff auf das adaptierte Modellelement nötig ist. Dadurch wird einerseits ein sehr schneller Zugriff auf die adaptierten Modellelemente möglich, andererseits wird ein Modellelement immer nur durch ein und dasselbe Adapterobjekt repräsentiert. Dies ist insbesondere für die TGGs von Vorteil, da die Korrespondenzobjekte immer auch eine Referenz auf die Modellelemente besitzen. Bei einem adaptierten Modell werden hier die Modelladapter referenziert. Würden die Adapterobjekte ständig verworfen werden, müsste zusätzlicher Aufwand betrieben werden, um diese Referenzen immer aktuell zu halten.

Die Implementierung von Modelladaptern ist eine komfortable Möglichkeit zur Überbrückung unterschiedlicher Techniken in den verschiedenen Werk-

zeugen und ihrer Metamodelle. Selbstverständlich kann die hier vorgestellte Adapterimplementierung weiter optimiert werden, indem beispielsweise für jeden Modellelementtyp nur ein einziges Adapterobjekt verwendet wird statt einem Adapterobjekt pro Modellelement. Weiterhin könnte die Implementierung eines Adapters automatisiert werden, falls das zu adaptierende Metamodell nach einem Standard oder einer dokumentierten Richtlinie implementiert wurde. Beispielsweise könnten Adapter für Metamodelle, die auf dem Java Metadata Interface (JMI) basieren, vollautomatisch aus der Spezifikation des Metamodells generiert werden. Allerdings könnte in solchen Fällen ebenfalls die Codegenerierung aus den Storydiagrammen angepasst werden, wie dies zum Beispiel für das Eclipse Modeling Framework (EMF) geschehen ist.

7.4 Evaluation

Ein wichtiger Bestandteil der Evaluation ist die prototypische Implementierung unseres Ansatzes. Die durch die prototypische Implementierung realisierte Werkzeugunterstützung haben wir bereits im vorherigen Abschnitt kennen gelernt. Die verfügbare Werkzeugunterstützung eröffnet jedoch weitere Möglichkeiten der praktischen Erprobung. So konnte auf der Grundlage der prototypischen Implementierung gezeigt werden, dass die Spezifikation von Regeln zur Modelltransformation, Modellintegration und Modellsynchronisation mit unserem Ansatz praktikabel und die Ausführung dieser Regeln effizient ist. Mit diesem Teil der Evaluation beschäftigt sich dieser Abschnitt. Hierbei werden zunächst überblicksartig verschiedene Spezifikationen und anschließend die wichtigsten Ergebnisse der durchgeführten Leistungsmessungen vorgestellt.

7.4.1 Spezifizierte Korrespondenzregeln

In dieser Arbeit haben wir den Ansatz zur Spezifikation von Korrespondenzbeziehungen am Beispiel von Block- und Klassendiagrammen dargestellt. Dieses Beispiel haben wir auch mit Hilfe der entwickelten Werkzeugunterstützung umgesetzt, um Modellsynchronisationen zwischen den beiden Modellen durchzuführen. Neben diesem Beispiel wurden weitere Fallstudien unterschiedlichen Umfangs durchgeführt:

- In der Diplomarbeit von Jörg Baksmeier wurde eine Modellsynchronisation zwischen UML-Klassendiagrammen und Java-Code realisiert

[Bak06]. Zur Synchronisation wurden insgesamt 31 TGG-Regeln eingesetzt. Diese TGG-Regeln sind zuvor im Rahmen der Diplomarbeit von Alexander Geburzi aus Beispielzuordnungen mit der in Abschnitt 4.2 vorgestellten Technik automatisch synthetisiert worden [Geb06]. Die eingesetzten TGG-Regeln berücksichtigen Klassen, Attribute, Methoden mit dazugehöriger Methodensignatur sowie unidirektionale Assoziationen. Nicht berücksichtigt wurden hingegen Methodenrümpfe der Zugriffsmethoden sowie bidirektionale Assoziationen. Zur Repräsentation des Java-Codes wurde dabei auf den abstrakten Syntaxbaum aus dem JDT-Projekt[5] von Eclipse zurückgegriffen, der über die in der Projektgruppe Reclipse[6] entwickelten Werkzeug- und Modelladapter an Fujaba4Eclipse – wie in Abschnitt 7.3 beschrieben – angebunden wurde. Anhand dieser Fallstudie konnte erfolgreich gezeigt werden, dass mit dem Lösungsansatz eine Synchronisation von Modell und Code durchgeführt werden kann.

- Im Rahmen des ISILEIT-Projekts (vgl. Abschnitt 2.1.2) wurden zwei auf TGGs basierende Spezifikationsvarianten zur SPS-Codegenerierung untersucht. Dabei wurde in beiden Fällen aus einem I/O-Automaten SPS-Code in der Sprache *Strukturierter Text* (ST) erzeugt. Bei der ersten Spezifikationsvariante wurden TGG-Regeln auf der Grundlage des Metamodells des I/O-Automaten und der abstrakten Syntax der Sprache ST erstellt. Diese TGG-Regeln wurden zur formalen Verifikation der semantischen Äquivalenz der beteiligten Sprachen eingesetzt (vgl. Abschnitt 6.2). Die zweite Spezifikation erfolgte mit TGGs und Textschablonen und hatte den Zweck, den praktischen Nutzen einer solchen Kombination zu überprüfen. Wie bereits in Abschnitt 4.1.2 beschrieben, eignet sich der kombinierte Ansatz aus TGGs und Textschablonen jedoch lediglich zur Modelltransformation, oder genauer gesagt, zur Codegenerierung. Eine bidirektionale Synchronisation zwischen einem Modell und Code auf der Grundlage von Textschablonen ist damit bisher nicht möglich.

- Eine weitere Evaluation des TGG-Ansatzes wurde an der Hochschule Darmstadt im Rahmen der Masterarbeit von Arpad Vasarhelyi durchgeführt [Vas06]. Die Evaluation erfolgte an einer Fallstudie, bei der plattformspezifische mit plattformunabhängigen Modellen synchroni-

[5] Java Development Tools, siehe auch `http://www.eclipse.org/jdt/`
[6] A Reverse Engineering Framework for Eclipse

siert wurden. Die plattformunabhängigen Modelle wurden durch UML-Klassendiagramme repräsentiert. Die plattformspezifischen Modelle hingegen durch EJB[7]-Komponentenmodelle, denen ebenfalls UML-Klassendiagramme zugrunde liegen. Zur Spezifikation der benötigten Korrespondenzbeziehungen reichten bereits 5 TGG-Regeln. Die zentrale Frage dieser Fallstudie, ob bei der Modellsynchronisation auch Änderungen berücksichtigt werden können ohne dabei Verfeinerungen im Zielmodell zu überschreiben, konnte positiv beantwortet werden.

- An der Technischen Universität Wien erfolgte im Rahmen der Magisterarbeit von Güzide Selin Altan eine weitere Evaluation anhand einer Fallstudie [Alt08]. Bei der betrachteten Fallstudie wurden TGGs zur Modelltransformation und -synchronisation von Geschäftsprozessmodellen eingesetzt. Dabei wurden *Ereignisgesteuerte Prozessketten* (EPKs) in UML-Aktivitätsdiagramme transformiert und nach durchgeführten Modelländerungen wieder miteinander synchronisiert. Die Korrespondenzbeziehungen wurden mit insgesamt 21 TGG-Regeln spezifiziert, wobei viele dieser TGG-Regeln lediglich verschiedene Varianten einer TGG-Regel repräsentieren. Hier hat sich gezeigt, dass eine Möglichkeit zur Wiederverwendung von TGG-Regeln mit einem dazugehörigen Verfeinerungskonzept durchaus sinnvoll und hilfreich wäre.

- Der in dieser Arbeit vorgestelle Lösungsansatz wurde in der Studienarbeit von Oliver Rohe eingesetzt, um eine Modelltransformation zu spezifizieren und auszuführen [Roh06]. Die dabei spezifizierten TGG-Regeln dienen der Übersetzung von TGG-Regeln selbst. Hintergrund der Übersetzung von TGG-Regeln ist, dass an der Universität Paderborn neben dem hier vorgestellten, generativen Ansatz zur Ausführung von TGG-Regeln ein interpretativer Ansatz, der sogenannte TGG-Interpreter, entwickelt wurde [KRW04]. Dieser TGG-Interpreter basiert auf denselben Konzepten wie wir sie in dieser Arbeit vorgestellt haben. Allerdings werden diese Konzepte im TGG-Interpreter durch ein abweichendes Metamodell repräsentiert. Um die mit dem TGG-Editor des generativen Ansatzes spezifizierten TGG-Regeln im TGG-Interpreter nutzen zu können, müssen die TGG-Regeln in den Formalismus des TGG-Interpreters übersetzt werden. Die Spezifikation der zur Modelltransformation benötigten Korrespondenzbeziehungen besteht aus insgesamt 20 TGG-Regeln, die in [Roh06] angegeben sind.

[7] Enterprise JavaBeans, siehe auch `http://java.sun.com/products/ejb/`

- Weitere Korrespondenzregeln wurden im Rahmen des MATE-Projekts spezifiziert [GMW06]. Bei diesem Projekt werden Matlab/Simulink-Modelle erstellt, die im Rahmen einer Analyse als Muster dienen. Diese Muster beschreiben Situationen, die in einem Matlab/Simulink-Modell vermieden werden sollen. Im Rahmen einer automatischen Analyse wird nach diesen Mustern gesucht und erkannte Muster dem Benutzer gemeldet. Damit das in dem Projekt eingesetzte Analysewerkzeug nach diesen Mustern suchen kann, müssen die in Matlab/Simulink spezifizierten Muster in die Repräsentation des Analysewerkzeugs übersetzt werden. Hierfür wurde der in dieser Arbeit beschriebene Ansatz mit der dazugehörigen Werkzeugunterstützung eingesetzt.

- In der Studienarbeit von Yascha Cebeci wurden TGGs eingesetzt, um Java-Codebeispiele in Objektstrukturen von Graphtransformationsregeln zu übersetzen [Ceb07]. Aus jeweils zwei solchen Objektstrukturen konnte anschließend mit den in der Studienarbeit entwickelten Algorithmen eine Graphtransformationsregel automatisch synthetisiert werden. Dadurch wurde – ähnlich zu der in Abschnitt 4.2 vorgestellten Spezifikation von Korrespondenzregeln durch Beispielzuordnungen – eine Spezifikation von Codetransformationen anhand konkreter Beispiele realisiert. Die Spezifikation der Modelltransformation besteht aus 7 TGG-Regeln.

Bei den hier erwähnten Fallstudien ist zu berücksichtigen, dass sie zum Teil unterschiedliche Ziele verfolgten. In einigen Fallstudien wurden die Korrespondenzbeziehungen spezifiziert, um mit der in dieser Arbeit entwickelten Werkzeugunterstützung eine Modelltransformation durchzuführen. In einigen anderen Fällen wurde jedoch auch die Modellsynchronisation untersucht. Wichtig ist, dass anhand dieser Fallstudien gezeigt werden konnte, dass der Ansatz der TGGs durchaus geeignet ist, um Korrespondenzbeziehungen zwischen unterschiedlichen Modellen zu spezifizieren. Natürlich heißt das nicht, dass die Technik der TGGs und die hier vorgestellte Modellsynchronisation für alle Arten von Modellen gleich gut geeignet sind. Für eine vollständige Evaluation sind viele weitere Beispiele aus der Praxis notwendig, die aufgrund des dafür erforderlichen zeitlichen Aufwands im Rahmen dieser Arbeit aber nicht mehr durchgeführt werden konnten.

7.4.2 Leistungsmessungen

In diesem Abschnitt präsentieren wir Leistungsmessungen, die wir durchgeführt haben, um eine Einschätzung der Geschwindigkeit unseres Ansatzes im Vergleich zu anderen Ansätzen zu erhalten. Leider existieren nur sehr wenige Werkzeuge, die eine Modellsynchronisation, wie wir sie hier kennengelernt haben, unterstützen (siehe Abschnitt 8.2). Diese Werkzeuge sind zumeist auf spezielle Modelle festgelegt und zudem nicht frei zugänglich. Dadurch konnte ein Leistungsvergleich mit diesen Werkzeugen nicht durchgeführt werden.

Um dennoch eine Einschätzung des Laufzeitverhaltens zu erhalten, haben wir das in dieser Arbeit erstellte Werkzeug mit Werkzeugen zur Modelltransformation verglichen. Die Modelltransformation stellt in unserem Ansatz einen Spezialfall der Modellsynchronisation dar. Hier sind in den letzten Jahren – aufgrund der Aktualität und Relevanz dieses Themas – sehr viele Ansätze und Werkzeuge entwickelt worden. Die Werkzeuge unterscheiden sich hauptsächlich durch die Art, wie die Transformationsregeln spezifiziert und ausgeführt werden. Für unsere Leistungsmessungen haben wir uns auf zwei Vertreter dieser Werkzeuge festgelegt. Bei dem ersten Werkzeug handelt es sich um *medini QVT* der Firma *ikv++ technologies ag* [IKV], das die Sprache QVT-Relations der OMG unterstützt. Das zweite Werkzeug gehört zum *M2M-Projekt* der *Eclipse Foundation* [M2M] und implementiert die Sprache QVT-Operational.

Durchführung

Die Leistungsmessungen werden auf einem Rechner mit einem Intel® Core™ 2 Duo E6300 Prozessor mit 1,86 GHz und 2048 MB Arbeitsspeicher durchgeführt. Als Betriebssystem wird Microsoft® Windows Vista – Home Premium mit installiertem Service Pack 1 eingesetzt. Bei der Java-Laufzeitumgebung handelt es sich um die Java™ SE Runtime Environment in der Version 1.6.0_07, auf deren Grundlage auch das eingesetzte Eclipse 3.4.1 (Ganymede) ausgeführt wird. Die Modelltransformationswerkzeuge sind als Plug-ins für Eclipse realisiert. Das Werkzeug medini QVT wird in der Version 1.6.0.25263 verwendet. Das Eclipse Operational QVT liegt in der Version 1.0.1 vor. Für die Übersetzung verwenden wir die aus dieser Arbeit bereits bekannten Korrespondenzbeziehungen zwischen Block- und Klassendiagrammen, die wir in der jeweiligen Beschreibungssprache des zugrundeliegenden Werkzeugs spezifiziert haben.

Für die Leistungsmessungen haben wir drei Plug-ins erstellt, wobei jedes Plug-in für die Leistungsmessung eines Werkzeugs verantwortlich ist. Hierzu erstellt ein Plug-in zunächst ein Blockdiagramm, welches als Quellmodell dient. Die Anzahl der Blöcke in einem Blockdiagramm kann vom Benutzer durch einen Parameter festgelegt werden. Darüber hinaus kann über einen weiteren Parameter die Anzahl der Blöcke, die ein Block enthalten soll, definiert werden. Die Anzahl der Blöcke legt die Modellgröße fest. Die Anzahl von Blöcken in einem Block bestimmt den Verzweigungsgrad und damit die hierarchische Struktur des Blockdiagramms. Ein geringer Verzweigungsgrad bewirkt, dass die hierarchische Struktur des Blockdiagramms schmal und tief ist. Im Gegensatz dazu führt ein hoher Verzweigungsgrad zu einer breiten und flachen hierarchischen Struktur. Die hierarchische Struktur eines Blockdiagramms spiegelt sich direkt in dem dazugehörigen Korrespondenzmodell wieder, so dass darüber die Struktur des Korrespondenzmodells beeinflusst und das Laufzeitverhalten unseres Algorithmus für unterschiedliche Ausprägungen des Korrespondenzmodells untersucht werden kann.

Nach der Erstellung eines Blockdiagramms mit den gegebenen Parametern wird das jeweilige Werkzeug zur Modelltransformation initialisiert und dabei die entsprechenden Transformationsregeln geladen. Die Leistungsmessung der Modelltransformation beginnt erst nach der Initialisierung, so dass weder die Initialisierung noch die Erstellung eines Blockdiagramms in die Zeitmessung einfließen. Zur Zeitmessung wird die Systemzeit vor und nach der Modelltransformation ermittelt und daraus die benötigte Zeit für die Modelltransformation berechnet. Über einen weiteren Parameter kann die Anzahl der durchzuführenden Leistungsmessungen für die eingestellte Modellgröße festgelegt werden. Hierbei wird für jede Messung das Transformationswerkzeug neu initialisiert sowie ein neues und damit noch nicht transformiertes Blockdiagramm erstellt. Dieser Vorgang fließt ebenfalls nicht in die Zeiterfassung ein.

Die Leistungsmessungen werden mit Blockdiagrammen unterschiedlicher Größe durchgeführt, wobei wir den Verzweigungsgrad für die hier beschriebene Leistungsmessung auf zwei Blöcke festgelegt haben. Bei den ersten Leistungsmessungen werden Blockdiagramme mit 100–1.000 Blöcken übersetzt. In den nachfolgenden Leistungsmessungen wird die Modellgröße auf 1.000 Blöcke festgelegt und in jeder nachfolgenden Leistungsmessung um jeweils 1.000 weitere Blöcke erhöht. Die letzte Leistungsmessung erfolgt mit einem Blockdiagramm mit 25.000 Blöcken. Für jedes Blockdiagramm einer Größe wurde die Leistungsmessung 10 Mal wiederholt und der Mittelwert über die einzelnen Ergebnisse gebildet.

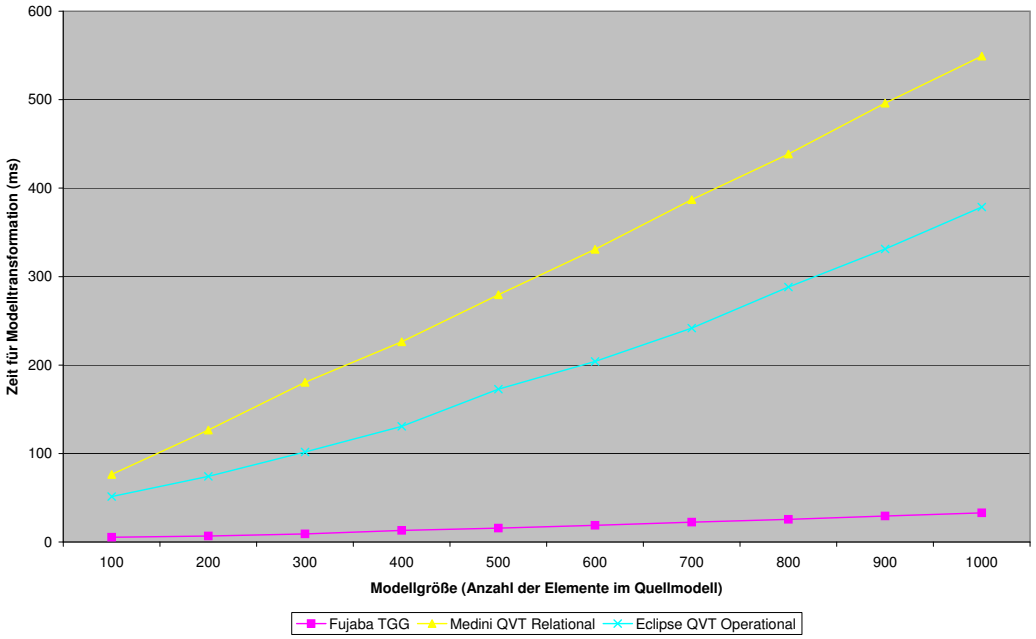


Abbildung 7.12: Leistungsmessung bei der Transformation kleiner Modelle

Ergebnisse

Das Diagramm in Abbildung 7.12 zeigt die ermittelten Zeiten der Leistungsmessungen für kleine Modelle (Blockdiagramme mit bis zu 1.000 Blöcken). Hierbei hat sich gezeigt, dass die Modelltransformation mit medini QVT am längsten dauert. Für ein Blockdiagramm mit 100 Blöcken wurden 74 ms und für ein Blockdiagramm mit 1.000 Blöcken bereits 550 ms benötigt. Die Modelltransformation mit Eclipse Operational QVT ist zwar schneller, kommt jedoch nicht an die Geschwindigkeit der Modelltransformation mit dem TGG-Ansatz von Fujaba heran. Der TGG-Ansatz benötigt für die Modelltransformation eines Blockdiagramms mit 1.000 Blöcken lediglich 33 ms. Die Transformation von 100 Blöcken dauert sogar nur 5 ms.

Für kleine Modelle mit maximal 1.000 Elementen liegen die ermittelten Zeiten zur Modelltransformation bei allen untersuchten Werkzeugen unter 600 ms und sind damit sehr niedrig. Aufgrund der niedrigen Zeiten sind die Unterschiede zwischen den einzelnen Werkzeugen für einen Benutzer kaum wahrnehmbar. Daher ist eine Modellsynchronisation durch Modelltransformation auch nach kleinen Änderungen mit allen diesen Werkzeugen durchaus denkbar.

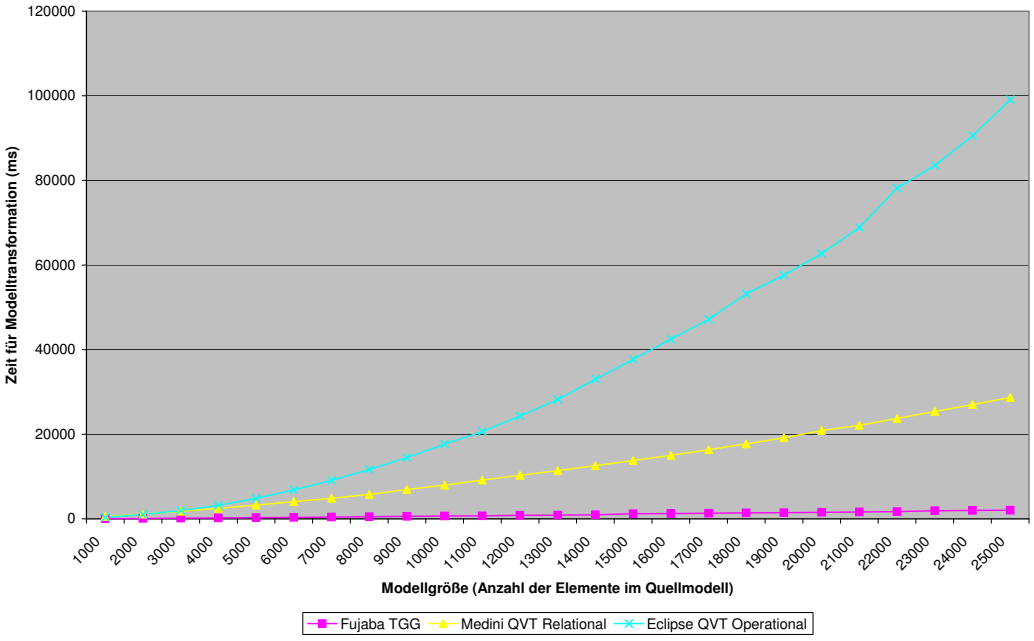


Abbildung 7.13: Leistungsmessung bei der Transformation größerer Modelle

Anders verhält es sich jedoch, wenn große Modelle (Blockdiagramme mit 1.000 - 25.000 Blöcken) transformiert werden. Die Ergebnisse dieser Leistungsmessungen sind in der Abbildung 7.13 dargestellt.

Die Leistungsmessungen bei der Transformation großer Modelle machen die Unterschiede beim Laufzeitverhalten zwischen den Werkzeugen deutlich. Dabei zeigt sich, dass Eclipse Operational QVT bei großen Modellen am langsamsten ist und sehr viel Zeit für die Modelltransformation benötigt. Die ermittelten Zeiten deuten darauf hin, dass hier ein polynomieller Algorithmus zur Modelltransformation verwendet wird. Im Gegensatz dazu verläuft die Modelltransformation mit medini QVT linear mit steigender Anzahl von Blöcken – die Transformation eines Blockdiagramms mit 25.000 Blöcken dauert hier daher nur ca. 29 sec. Am schnellsten ist wieder der TGG-Algorithmus. Dieser benötigt für die Transformation des größten Blockdiagramms lediglich 2050 ms. Bei dem Leistungsvergleich konnten zwei wesentliche Beobachtungen gemacht werden.

Zunächst kann festgestellt werden, dass kleine Modelle mit den beiden untersuchten QVT-Werkzeugen in vertretbarer Zeit transformiert werden können. Dies ist insofern interessant, da es sich hierbei um interpretative Umsetzungen des QVT-Standards handelt. Im Allgemeinen sind Interpreter

bei der Ausführung langsamer als generative Ansätze. Dies hat sich einerseits durch die durchgeführte Leistungsmessung bestätigt. Andererseits hat sich aber auch gezeigt, dass insbesondere für kleinere Modelle dieser Nachteil von nicht allzu großer Bedeutung ist, da die Zeiten bei einer Modellgröße von 1.000 Elementen unter 600 ms liegen.

Die zweite Beobachtung betrifft unseren eigenen Ansatz zur Modelltransformation und Modellsynchronisation. Dieser Ansatz basiert auf Tripel-Graph-Grammatiken, aus denen operationale Graphtransformationsregeln abgeleitet werden. Zur Ausführung einer Graphtransformationsregel muss eine Teilgraphensuche durchgeführt werden, die im Allgemeinen ein NP-vollständiges Problem darstellt. Aus diesem Grund werden in der Praxis häufig Ansätze gemieden, die auf Graphgrammatiken basieren. Mit den durchgeführten Leistungsmessungen konnte jedoch bestätigt werden, dass durch eine geschickte Einschränkung der Anwendungsstelle und die Angabe eines Anwendungskontextes, die in unserem Ansatz auf der Grundlage des Korrespondenzmodells erfolgt, die Regelanwendungen auf polynomielle bzw. sogar lineare Laufzeiten reduziert werden können. Natürlich hängt die Laufzeit für eine Modelltransformation nicht nur von der Größe der Modelle ab, sondern auch von der Komplexität der verwendeten Regeln. Dies gilt jedoch für alle Transformationsansätze, in denen zunächst Muster gesucht werden und erst anschließend in Muster eines anderen Modells übersetzt werden.

Allerdings hat sich gezeigt, dass selbst mit unserem Ansatz eine Modelltransformation eines Blockdiagramms mit 25.000 Blöcken ca. 2 sec dauert. Damit ist eine Modellsynchronisation durch Modelltransformation bei großen Modellen nicht vertretbar – insbesondere dann nicht, wenn eine Synchronisation nach jeder Änderung durchgeführt werden soll. Aus diesem Grund haben wir in dieser Arbeit den Algorithmus erweitert, um auch inkrementelle Modellsynchronisationen ausführen zu können. Zur Überprüfung der erreichten Performanzsteigerungen wurden Leistungsmessungen und Leistungsvergleiche zwischen den verschiedenen Versionen unseres inkrementellen Algorithmus durchgeführt. Hierbei hat sich gezeigt, dass der zusätzliche Aufwand zur Überprüfung der Korrespondenzbeziehungen relativ gering ist und lokale Änderungen in nur wenigen Millisekunden synchronisiert werden können. Die genauen Ergebnisse dieser Leistungsmessungen, bei denen auch unterschiedliche Verzweigungsgrade betrachtet worden sind, wurden bereits publiziert [GW09]. Darauf aufbauende Optimierungen sowie dazugehörige Leistungsmessungen wurden in der Diplomarbeit von Stephan Hildebrandt dokumentiert [Hil07] und in [GH08] veröffentlicht, so dass auf eine detaillierte Darstellung an dieser Stelle verzichtet wird.

7.5 Zusammenfassung

In diesem Kapitel haben wir die im Rahmen dieser Arbeit entstandene Werkzeugunterstützung vorgestellt, die zur Modelltransformation, Modellintegration und Modellsynchronisation eingesetzt werden kann. Die Werkzeuge wurden in die Entwicklungsumgebung Fujaba4Eclipse integriert und unterstützen die in Abschnitt 2.4 vorgestellte Methode. Durch den Einsatz von Werkzeug- und Modelladaptern ist die entwickelte Werkzeugunterstützung nicht auf Fujaba4Eclipse eingeschränkt – die entwickelten Werkzeuge können leicht in andere Java-basierte Entwicklungsumgebungen integriert werden. Mit den durchgeführten Leistungsmessungen konnte darüber hinaus gezeigt werden, dass der hier vorgestellte Ansatz sehr schnell arbeitet und daher auch zur Synchronisation großer Modelle sehr gut geeignet ist. Die Anforderungen aus Abschnitt 2.4 konnten somit umgesetzt werden.

Bei der entwickelten Werkzeugunterstützung handelt es sich natürlich nur um einen Forschungsprototypen – für einen Einsatz in der Praxis wäre es sinnvoll, die Benutzerschnittstelle weiter zu verbessern. Bei der prototypischen Implementierung sind wir pragmatisch vorgegangen und haben das in Fujaba4Eclipse verfügbare Graphersetzungssystem mit der dazugehörigen Werkzeugunterstützung verwendet. Dieser Umstand hat die Entwicklungszeit reduziert und die Umsetzung erst möglicht gemacht. Allerdings sind nicht alle Konzepte aus Kapitel 3 implementiert worden. So können in der aktuellen Implementierung beispielsweise Bedingungen nicht mit OCL spezifiziert werden. Aufgrund der Tatsache, dass eine Integration von OCL in Fujaba4Eclipse sehr aufwändig wäre, haben wir stattdessen die in Fujaba4Eclipse verfügbaren Ausdrücke für Bedingungen verwendet. Trotz dieser Einschränkung konnte durch die prototypische Implementierung die Umsetzbarkeit unseres Ansatzes erfolgreich gezeigt werden.

Kapitel 8

Verwandte Arbeiten

In den vorangegangenen Kapiteln haben wir die Konzepte zur automatischen Modellsynchronisation sowie deren Realisierung kennen gelernt. In diesem Kapitel vergleichen wir unseren Ansatz mit verwandten Arbeiten. Hierbei werden wir insbesondere die Unterschiede zu unserer Arbeit hervorheben und zeigen, dass derzeit kein durchgängiger Ansatz zur Spezifikation von Modellbeziehungen existiert, auf dessen Grundlage eine Modelltransformation, eine Modellintegration sowie eine Modellsynchronisation ausgeführt werden kann. Sowohl die Modelltransformation als auch die Modellintegration sind Spezialfälle der in dieser Arbeit vorgestellten Modellsynchronisation. Daher stellen wir zunächst in Abschnitt 8.1 einige Arbeiten zur Modelltransformation und Modellintegration vor. Anschließen behandeln wir in Abschnitt 8.2 existierende Ansätze zur Modellsynchronisation. Im darauf folgenden Abschnitt 8.3 diskutieren wir einige wenige vorhandene Ansätze zur Vereinfachung der Spezifikation von Modelltransformationen. Wir schließen das Kapitel mit einer Zusammenfassung in Abschnitt 8.4.

8.1 Modelltransformation und Modellintegration

In dieser Arbeit haben wir eine Technik zur Modellsynchronisation auf der Grundlage von Tripel-Graph-Grammatiken vorgestellt. TGGs sind nicht neu. Sie wurden bereits in einigen Arbeiten zur Modelltransformation und Modellintegration eingesetzt. In diesem Abschnitt stellen wir daher zunächst diese Arbeiten vor. Aufgrund der Tatsache, dass wir zur Modellsynchronisation einen Ansatz zur Modelltransformation und Modellintegration verwenden, betrachten wir anschließend andere Arbeiten in diesem Themenbereich und untersuchen dabei, inwieweit diese Ansätze zur Modellsynchronisation geeignet sind.

8.1.1 Tripel-Graph-Grammatiken

Tripel-Graph-Grammatiken sind bereits 1994 von Andy Schürr [Sch94] als Erweiterung von T. W. Pratts Pair-Graph-Grammatiken [Pra71] eingeführt und von Martin Lefering im Rahmen des IPSEN-Projekts zur Entwicklung von Integrationswerkzeugen in einer integrierten Entwicklungsumgebung eingesetzt worden [Lef95, LS96].

In einer solchen Entwicklungsumgebung hat ein Integrationswerkzeug die Aufgabe, zueinander in Beziehung stehende Elemente zweier Modelle zu identifizieren, um diese Korrespondenzbeziehungen über den gesamten Lebenszyklus der Modelle zu überwachen und auf Konsistenz zu prüfen. Hierzu werden im Rahmen einer Modelltransformation oder Modellintegration Korrespondenzobjekte erstellt, die dann auf Anforderung durch den Benutzer einer Konsistenzprüfung unterzogen werden können.

Die in der Arbeit von Martin Lefering entwickelten Integrationswerkzeuge arbeiten in *einer zuvor festgelegten* Hauptintegrationsrichtung. Für eine bidirektionale Modelltransformation oder Modellintegration müssen daher *zwei separate* Integrationswerkzeuge erstellt werden, wohingegen mit unserer Technik nur ein Werkzeug für beide Richtungen benötigt wird. Darüber hinaus müssen in dem Ansatz von Martin Lefering die für die Integrationswerkzeuge notwendigen operationalen Graphersetzungsregeln noch von Hand aus einer TGG abgeleitet werden. Eine Automatisierung, wie wir sie in dieser Arbeit vorgestellt haben, wurde nicht umgesetzt.

Bei der Konsistenzprüfung wird untersucht, ob die durch die Korrespondenzobjekte annotierten Beziehungen noch gültig sind. Zusätzlich werden während der Konsistenzprüfung neue Korrespondenzbeziehungen identifiziert. Aufgrund der festgelegten *Hauptintegrationsrichtung* werden dabei nur Modellelemente *im Quellmodell* berücksichtigt.

Im Gegensatz zu der in dieser Arbeit vorgestellten Konsistenzerhaltung durch Modellsynchronisation findet in einem Integrationswerkzeug eine automatische Behebung von Inkonsistenzen nur dann statt, falls ein neues Element im Quellmodell *erstellt* oder ein existierendes Modellelement *gelöscht* wurde. Attributwerte werden zwar bei der Modelltransformation, Modellintegration und den nachfolgenden Konsistenzprüfungen berücksichtigt, aber eine automatische Aktualisierung der Attributwerte zur Konsistenzwiederherstellung wird *nicht* unterstützt. Ebenso ist eine automatische Vervollständigung der Modelle, wie wir sie in dieser Arbeit kennen gelernt haben, nicht vorgesehen. Die Behebung von Inkonsistenzen findet daher *überwiegend manuell*, d. h., durch den Benutzer, statt.

Im Ansatz von Martin Lefering muss in jeder TGG-Regel ein Modellelement speziell gekennzeichnet werden. Diese Modellelemente werden *dominante Quellinkremente* genannt. Alle übrigen Modellelemente werden als *Kontextinkremente* bezeichnet. Die dominanten Quellinkremente dienen als Ausgangspunkt der zur Regelanwendung benötigten Mustersuche, so dass zur Ausführung einer Modelltransformation, Modellintegration oder Konsistenzprüfung zunächst alle dominanten Quellinkremente in einem Modell identifiziert und eingesammelt werden müssen.

Hier liegt der Hauptunterschied zu unserem Ansatz. Im Ansatz von Martin Lefering wird eine *inkrementelle Nachintegration*, d. h., eine Überprüfung und Behebung von Inkonsistenzen, *nur im Zielmodell* inkrementell durchgeführt. Dazu werden zunächst *alle* bereits existierenden Korrespondenzobjekte betrachtet. Anschließend muss zur Identifikation der neu hinzugefügten, dominanten Quellinkremente das *gesamte Quellmodell* analysiert werden.[1] Im Gegensatz dazu kann die in dieser Arbeit vorgestellte Modellsynchronisation als *vollständig inkrementell* bezeichnet werden, da die Inkonsistenzen *lokal* ermittelt und behoben werden.

In diesem Zusammenhang besteht ein weiterer Unterschied zwischen den beiden Ansätzen. So muss in dem Ansatz von Martin Lefering bei jeder Konsistenzprüfung ein Abhängigkeitsgraph von anwendbaren Regeln berechnet werden, mit dem eine günstige Reihenfolge für die Regelanwendungen bestimmt wird. Hierzu müssen die Kontextinkremente wiederholt geprüft werden, was zu Effizienzproblemen führt [Bec07]. In unserem Ansatz hingegen wird die Reihenfolge unmittelbar bei der Ausführung durch das Korrespondenzmodell vorgegeben. Hierzu werden die Abhängigkeiten zwischen den Regeln ausgenutzt, die bereits in den Regeln implizit enthalten sind. Eine zusätzliche Spezifikation oder Berechnung dieser Abhängigkeiten ist daher nicht notwendig.

Die Arbeit von Martin Lefering wurde von Simon Becker fortgesetzt [BLW04, Bec07]. Daher gelten die meisten der bereits genannten Eigenschaften auch für die dort erstellten Integrationswerkzeuge. Dies betrifft insbesondere die Algorithmen zur inkrementellen Modelltransformation und Modellintegration. Allerdings wurde der Ansatz dahingehend erweitert, dass nun eine bidirektionale Modelltransformation, Modellintegration und Konsistenzprüfung mit nur einem Integrationswerkzeug möglich ist.

[1] In Ansätzen zur Modelltransformation wird dieses Vorgehen als *„target incremental“* bezeichnet. Modelltransformationen, in denen zur Identifikation der Inkonsistenzen nur ein kleiner Ausschnitt eines Modells betrachtet werden muss, werden hingegen *„source incremental“* genannt [CH06].

In der Arbeit von Simon Becker wurde der Schwerpunkt auf Benutzerinteraktionen gelegt, da in dem betrachteten Anwendungsbereich die Korrespondenzbeziehungen häufig nicht eindeutig sind, so dass prinzipiell mehrere Regeln an einer Anwendungsstelle ausgeführt werden können. Aufgrund der potentiellen Konflikte zwischen Regeln und den zur Auflösung notwendigen Benutzerinteraktionen werden die Regeln aufgeteilt in Regeln zur Mustersuche und Musterersetzung, die dann in zwei Phasen ausgeführt werden. In der ersten Phase werden zunächst auf Grundlage der Mustersuche potentielle Regelanwendungsstellen identifiziert sowie Beziehungen und Konflikte zwischen diesen Anwendungsstellen bestimmt. In der zweiten Phase wird unter Einbeziehung von Benutzerinteraktionen die Musterersetzung und damit die Transformation bzw. Integration durchgeführt.

Bei der Ausführung der Regeln entscheidet der Benutzer, welche der potenziell an einer Anwendungsstelle möglichen Regeln tatsächlich angewendet wird. Dazu unterbricht der Algorithmus die Arbeit und wartet auf eine Benutzereingabe. Für große Modelle kann dies allerdings bedeuten, dass der Benutzer während einer ersten Übersetzung sehr viele Benutzereingaben tätigen muss. Inwieweit dieser Ansatz auch für große Modelle geeignet ist, muss sich daher in der Praxis noch zeigen.

Der Ansatz von Simon Becker ist im Gegensatz zur Arbeit von Martin Lefering stark automatisiert, so dass eine Programmierung der notwendigen Integrationswerkzeuge von Hand nicht notwendig ist. Wie in unserem Ansatz werden auch hier die operationalen Regeln automatisch aus der Spezifikation generiert. Während in unserer Arbeit die Algorithmen auf eine effiziente Modelltransformation und Modellsynchronisation optimiert sind, liegt in der Arbeit von Simon Becker der Schwerpunkt auf der Erkennung und Behandlung von Regeln, die zueinander in einem Konflikt stehen. In diesem Zusammenhang wäre eine Kombination beider Ansätze sehr vorteilhaft [Bec07], um solche Konflikte bei der Modellsynchronisation zu erkennen und durch Benutzerinteraktionen aufzulösen. Derzeit werden Konflikte bei mehreren möglichen Regelanwendungen in unserem Ansatz ignoriert – es wird einfach die erste Regel ausgeführt, die anwendbar ist.[2]

Parallel zu der Arbeit von Simon Becker hat Alexander Königs in seiner Arbeit untersucht, wie der MOF 2.0 QVT-Standard der OMG mit Hilfe der TGGs implementiert werden kann [Kön08]. Durch die Übertragung der

[2] Allerdings kann in unserem Ansatz die Reihenfolge der Regeln durch die in Abschnitt 7.2.2 beschriebene Konfigurationsdatei beeinflusst werden. Dies entspricht weitestgehend einer Priorisierung von Regelanwendungen.

Metamodellierungstechniken des MOF-Standards auf die TGGs konnten die Vorteile beider Spezifikationstechniken in einer einzigen Technik zusammengefasst und die jeweiligen Nachteile beseitigt werden. Eine entsprechende Abbildung wurde bereits in der Diplomarbeit von Joel Greenyer vorgestellt [Gre06], die im Rahmen dieser Dissertation entstanden ist und überblicksartig in Abschnitt 4.3 skizziert wurde.

Zur Implementierung des QVT-Standards hat Alexander Königs einen Transformationsalgorithmus für TGGs entworfen, der von der hierarchischen Struktur der beteiligten Modelle ausgeht [KS06, Kön08]. Der Vorteil ist, dass die Transformation an einer beliebigen Stelle im Modell begonnen werden kann. Die Anwendung eines initialen Axioms ist damit nicht notwendig. Ein Nachteil ist jedoch, dass aus den zugrundeliegenden Metamodellen bereits eine hierarchische Struktur erkennbar sein muss. Diese Voraussetzung erfüllen jedoch beispielsweise nur MOF- oder EMF-Metamodelle. Im Gegensatz dazu wird eine solche Struktur in unserem Ansatz nicht gefordert, so dass der Ansatz nicht auf MOF- oder EMF-Modelle eingeschränkt ist.

Ein anderer Nachteil betrifft die Performanz des entwickelten Algorithmus, da zuerst eine Mustersuche im Modell durchgeführt und erst anschließend überprüft wird, ob eine Regel bereits zu diesem Zeitpunkt angewendet werden kann. Dies wird aber anhand von Korrespondenzobjekten überprüft. Ist ein notwendiges Korrespondenzobjekt nicht vorhanden, so ist diese Bedingung nicht erfüllt und die Regel muss für diese Anwendungsstelle zurückgestellt werden. Die dadurch zusätzlich notwendigen Überprüfungen wirken sich damit negativ auf die Performanz des Algorithmus aus. Eine entsprechende Weiterentwicklung dieses Algorithmus, die diesen Nachteil aufhebt, wurde von Andy Schürr und Felix Klar in dem Beitrag [SK08] vorgestellt. Allerdings unterstützt dieser Algorithmus – wie auch der Algorithmus von Alexander Königs – keine inkrementelle Modellsynchronisation.

Neben den hier vorgestellten Arbeiten existieren weitere Arbeiten, die TGGs zu unterschiedlichen Zwecken einsetzen. Beispielweise verwendet Katja Cremer in ihrem Ansatz Tripel-Graph-Grammatiken zur Spezifikation von Transformationsregeln zwischen einem Strukturdokument und einem Architekturdokument. Jede Regel muss dabei durch den Benutzer explizit angestoßen werden, das heißt, es erfolgt keine automatische Transformation des gesamten Strukturmodells in das Architekturmodell. Stattdessen muss der Benutzer sowohl die Anwendungsstelle als auch die auszuführende Regel von Hand vorgeben. Eine Konsistenzprüfung oder Modellsynchronisation, wie wir sie in dieser Arbeit vorgestellt haben, war nicht das Ziel der Arbeit und wurde daher nicht umgesetzt.

Juan de Lara verwendet TGGs zur Synchronisation eines Modells mit den daraus abgeleiteten Sichten, die in verschiedenen Editoren dargestellt werden [GL04]. Hierbei werden die TGG-Regeln aber nur zur Erzeugung und Verwaltung der Korrespondenzbeziehungen eingesetzt. Die Synchronisation arbeitet ereignis-orientiert, das heißt, eine Änderungsoperation auf dem Modell wird aufgezeichnet, eine äquivalente Änderungsoperation für das Zielmodell berechnet, über die Korrespondenzbeziehung die Anwendungsstelle im Zielmodell identifiziert und dann die berechnete Änderungsoperation im Zielmodell ausgeführt.[3] Der inkrementelle Transformationsalgorithmus wird somit durch verschiedene Bearbeitungsaktionen, wie zum Beispiel Erstellen, Ändern oder Löschen von Modellelementen, ausgelöst. Hierzu müssen alle möglichen Bearbeitungsaktionen und Aktualisierungsaktivitäten sowohl im Quell- als auch im Zielmodell spezifiziert werden. Obwohl die Spezifikation der Bearbeitungsaktionen und Aktualisierungsaktivitäten visuell erfolgt und damit komfortabler als eine ad-hoc Programmierung ist, nimmt der Aufwand mit der Anzahl und Granularität der möglichen Bearbeitungsaktionen zu. Darüber hinaus ist eine vollständige Modelltransformation in einem Schritt mit diesem Ansatz nicht möglich. Im Gegensatz dazu deckt der in dieser Arbeit vorgestellte Ansatz beide Fälle ab.

8.1.2 Andere Ansätze zur Modelltransformation und Modellintegration

In diesem Abschnitt beschäftigen wir uns mit anderen Ansätzen zur Modelltransformation und Modellintegration und diskutieren, ob diese zur Modellsynchronisation geeignet sind.

Modelltransformation

Motiviert durch die Einführung der *Model Driven Architecture* (MDA) und der Ausschreibung zu dem neuen Standard *Query/View/Transformation* (QVT) [RFP03] für Modelltransformation durch die *Object Management Group* (OMG) sind Modelltransformationen in den Fokus vieler Forschungsaktivitäten gerückt. Mittlerweile ist eine finale Version der Spezifikation dieses Standards veröffentlicht [QVT08]. In dieser Spezifikation spielen inkrementelle Modelltransformationen, die auch zur Modellsynchronisation

[3] Die Nachteile und grundsätzlichen Probleme einer ereignis-orientierten Modellsynchronisation werden von Jack Greenfield et al. in [GSCK04] diskutiert (Kapitel 14, Seiten 471–473), so dass wir an dieser Stelle nicht weiter darauf eingehen wollen.

beitragen können, eine wichtige Rolle. Bisher existieren zwei Werkzeuge, die den operationalen Teil dieses Standards implementieren [Bor06, Fra06]. Inkrementelle Modelltransformationen, wie sie zur Modellsynchronisation benötigt werden, sind mit diesen Werkzeugen jedoch nicht möglich. Im Gegensatz dazu unterstützen zwei kommerzielle Werkzeuge [IKV, Mor], die den deklarativen Teil der Sprache verwenden, inkrementelle Modelltransformationen. Allerdings müssen diese Werkzeuge hierzu jedes Mal die Modelle vor der Übersetzung aus der dazugehörigen Dateirepräsentation laden, so dass eine schnelle und sofortige Modellsynchronisation nach durchgeführten Änderungen nicht möglich ist. Im Gegensatz dazu arbeitet der in dieser Arbeit vorgestellte Ansatz direkt auf den Modellrepräsentationen im Speicher, so dass eine Modellsynchronisation sofort nach jeder Änderung vorgenommen werden kann.

Eine prototypische Implementierung, die in einer Vorstudie zum QVT-Standard realisiert wurde, ist das von IBM entwickelte *Model Transformation Framework* (MTF) [Gri04]. Der Prototyp verwendet nicht die im Standard vorgeschlagene deklarative Spezifikationssprache, ist aber in der Lage, hinzugefügte oder gelöschte Modellelemente in einem Quellmodell zu identifizieren und das Zielmodell nachträglich zu aktualisieren. Leider ist es in MTF nicht möglich, zusätzliche Bedingungen an die Modellelemente zu formulieren. Daher ist eine vollständige Modellsynchronisation mit diesem Ansatz nicht möglich [DGS05, KC05b]. Darüber hinaus existieren keine Veröffentlichungen, die den verwendeten Ansatz oder Leistungsmessungen beschreiben.

Neben QVT existieren viele weitere Ansätze zur Modelltransformation – jeder Ansatz ist für einen speziellen Zweck und eine bestimmte Anwendungsdomäne mit verschiedenen Anforderungen unterschiedlich gut geeignet. Im Folgenden stellen wir einige der bekanntesten Modelltransformationsansätze vor. Für eine Klassifikation und Übersicht der Ansätze verweisen wir auf [CH03, CH06].

Ein bekannter Ansatz zur Modelltransformation ist *Extensible Stylesheet Language Transformation* (XSLT) [W3C99]. Die Spezifikation erfolgt auf Basis der *Extensible Markup Language* (XML) beziehungsweise *XML Metadata Interchange* (XMI). Zur Modelltransformation wird das Modell als XML/XMI-Dokument exportiert. Dieses Dokument wird dann in einem Schritt in das Zielmodell übersetzt. Eine inkrementelle Modelltransformation zur Propagation von Änderungen im Quellmodell an das Zielmodell wird nicht unterstützt. Die Spezifikation der Modelltransformationsregeln erfolgt auf Basis von XML und XSLT und ist sehr komplex, unleserlich und damit schwer zu verstehen.

Eine andere Kategorie von Modelltransformationsansätzen umfasst visuelle Transformationssprachen, die auf Graphgrammatiken und Graphtransformationssystemen basieren. Bei diesen Ansätzen werden die Modelle als Graphen interpretiert. Eine Modelltransformation wird durch Techniken der Graphersetzung erreicht. Beispiele für Ansätze, die in diese Kategorie fallen, sind VIATRA [VVP02], GreAT [VAS04], UMLX [Wil03], ATOM3 [LVA04] und BOTL [MB03]. Einige dieser Ansätze werden im Folgenden kurz vorgestellt. Eine vollständige Übersicht enthält [CH06].

VIATRA ist ein Ansatz zur Validierung und Verifikation von Modellen. Dabei werden Modelltransformationen eingesetzt, um ein Modell in einen verifizierbaren Formalismus zu übersetzen. Die Modelltransformation wird visuell in einer UML-ähnlichen Notation spezifiziert. Zur Ausführung der Modelltransformation wird die Spezifikation in Graphtransformationsregeln übersetzt und mit Abstract State Machines (ASM) kombiniert, indem die Graphtransformationsregeln in Kontrollstrukturen der ASMs eingebettet werden. Die Einbettung der Graphersetzungsregeln in Kontrollstrukturen ermöglicht die Ausführungsreihenfolge der Graphtransformationsregeln festzulegen und damit die einzelnen Graphtransformationsregeln zu komplexeren Modelltransformationen zusammen zu setzen.

In einem weiteren Schritt wird eine Implementierung der Modelltransformation generiert. Zur Ausführung der Modelltransformation muss das Modell in einer XMI-Repräsentation vorliegen. Das Ergebnis der Transformation ist wiederum ein XMI-Dokument, welches das erzeugte Zielmodell der Transformation enthält. Eine direkte Transformation von Modellen, wie sie in [KRW04] beschrieben wurde, ist mit diesem Ansatz nicht möglich. Ebenso existiert weder eine Möglichkeit zur Nachverfolgbarkeit der Modelltransformation, noch werden bidirektionale Modelltransformationen unterstützt. Damit ist dieser Ansatz nicht zur Realisierung einer Modellsynchronisation geeignet.

Die *Graph Rewriting and Transformation Language* (GReAT) [VVP02] verfügt über eine operationale Spezifikationstechnik, die ebenfalls auf Graphtransformationen basiert. Durch zusätzliche Konstrukte kann ein Kontrollfluss mit Ein- und Ausgabeparametern spezifiziert und damit die Ausführungsreihenfolge der Graphtransformationsregeln festgelegt werden. Wie in VIATRA, existieren keine weiteren Informationen zur Nachverfolgbarkeit der Modelltransformation. Ebenso kann die Modelltransformation nur unidirektional und in einem Durchlauf ausgeführt werden. Inkrementelle Modelltransformationen nach durchgeführten Änderungen im Quellmodell der Transformation sind damit nicht möglich.

In den bisher vorgestellten Ansätzen muss für jede Transformationsrichtung eine eigene Spezifikation erstellt werden. Für bidirektionale Modelltransformationen sind diese Ansätze daher weniger geeignet. Ein Ansatz, der bidirektionale Modelltransformationen unterstützt ist BOTL [MB03]. Die Spezifikation erfolgt in einer UML-ähnlichen Notation. Die Modelltransformation basiert auf Graphtransformationsregeln, die für die entgegen gesetzte Richtung aus der Spezifikation abgeleitet wird. Hierfür muss die Modelltransformation jedoch bijektiv sein. Diese Voraussetzung ist jedoch nur selten erfüllt [SK04]. Wie bei den zuvor genannten Ansätzen erfolgt die Modelltransformation in einem Schritt und ist nicht inkrementell.

Die bisher diskutierten Ansätze sehen keine Mechanismen zur Nachverfolgbarkeit der Modelltransformation vor. Dies verhindert eine inkrementelle Modelltransformation bzw. Modellsynchronisation, die zum Abgleich der Modelle nach einer erfolgten Modelltransformation und anschließend durchgeführten Änderungen am Quellmodell notwendig ist. Im Gegensatz dazu sind Tripel-Graph-Grammatiken (TGGs) eine spezielle Technik, die eine Spezifikation und Ausführung von bidirektionalen und inkrementellen Modelltransformationen ermöglicht.

Modellintegration

Die Modellintegration kann für unterschiedliche Menschen unterschiedliche Dinge bedeuten. Beispielsweise wird die Modellintegration häufig als ein Prozess angesehen, in dem aus zwei oder mehreren Modellen ein einziges gemeinsames Modell erstellt wird. In der Domäne des Modellmanagements wird diese Art der Modellintegration häufig als Modellverschmelzung (engl. model merging) bezeichnet [BHP00, SZK04]. Hierbei entsteht ein Modell in einem bestimmten Formalismus. Im Gegensatz dazu verstehen wir unter Modellintegration nicht die Verschmelzung mehrerer Modelle zu einem einzigen Modell, sondern die Zuordnung von Modellelementen zweier oder mehrerer Modelle zueinander, die durchaus auch in unterschiedlichen Formalismen gegeben sein können. In der Domäne des Modellmanagements wird diese Art der Modellintegration durch eine sogenannte Matching-Operation definiert: Zu zwei gegebenen Modellen wird eine Abbildung berechnet und als Ausgabe zurückgegeben [BHP00, SZK04].

Ein prominentes Werkzeug zur Modellintegration stellt der *ATLAS Model Weaver* (AMW) [FBV06] dar. Die Idee hinter AMW besteht darin, zwei Modelle miteinander zu „vernetzen“, allerdings nicht indem die Beziehungen a-priori spezifiziert, sondern manuell durch den Benutzer zwischen den

gegebenen Modellen hergestellt werden.[4] Im Gegensatz dazu werden in unserer Arbeit die Beziehungen zwischen Modellelementen explizit spezifiziert, um daraus automatisch Operationen, beispielsweise zur Modellintegration, abzuleiten.

Die Arbeiten zur Werkzeugintegration [Lef95, BLW04, Bec07] entsprechen am ehesten unserem Verständnis von Modellintegration. Dies ist damit zu begründen, dass die dort durchgeführte Werkzeugintegration stark auf der Integration von Dokumenten beruht, die als Modelle gesehen werden können. Die Integration erfolgt dort – wie in unserer Arbeit auch – auf der Basis von TGGs.

8.2 Modellsynchronisation

Eine erste Klassifikation und Kategorisierung der Modellsynchronisation sowie ein Ansatz zur Modellsynchronisation zwischen einem so genannten Feature Modell und seiner Spezialisierung wurde von Kim und Czarnecki in [KC05b] gegeben. In dem dort vorgestellten Ansatz basiert die Modellsynchronisation auf Verknüpfungen (engl. traceability links) zwischen Modellelementen der in Beziehung stehenden Modelle. Diese Verknüpfungen werden erstellt, während das ursprüngliche Modell geklont wird, um ein initiales Modell für die nachfolgende Spezialisierung zu erhalten. Nach der Erstellung der Verknüpfungen dienen die Verknüpfungen dazu, Änderungen an dem ursprünglichen Modell an die dazugehörige Spezialisierung weiter zu propagieren. Die Modellsynchronisation wird in diesem Anwendungsfall in nur eine Richtung durchgeführt. Der Anwendungsfall führt auch dazu, dass nur 1-zu-n Beziehungen erstellt werden. Die Autoren bestätigen, dass diese Art der Beziehungen einfacher zu implementieren und zu verwalten ist als n-zu-m Beziehungen, wie sie in unserem Ansatz eingesetzt werden können. Der Ansatz in [KC05b] ist für die dort gestellten Anforderungen ausreichend. Im allgemeinen Fall, in dem auch n-zu-m Beziehungen benötigt werden, die darüber hinaus in beide Richtungen gepflegt werden müssen, ist der Ansatz jedoch nicht einsetzbar.

Ivkovic and Kontogiannis haben einen Ansatz zur Modellsynchronisation entwickelt, der auf impliziten Nachverfolgbarkeitsrelationen basiert, das heißt, die Relationen zwischen Modellelementen werden nur im Metamodell definiert und implementiert, aber nicht zwischen den in Beziehung stehen-

[4] In einigen darauf aufbauenden Arbeiten wird versucht, die Vernetzung der Modelle mit Hilfe von Heuristiken automatisch durchzuführen [FBV06, RKRS05].

den Modellen, also nicht auf den Metamodellinstanzen [IK04a, IK04b]. In ihrem Ansatz müssen zur Modellsynchronisation spezielle Graphen aus den Metamodellen abgeleitet werden. Zusätzlich müssen atomare Änderungsoperationen auf den Knoten und Kanten dieses Graphen wie z. B. Hinzufügen, Löschen, Ändern, etc. spezifiziert und implementiert werden. Um zwei Modelle zu synchronisieren, werden diese Änderungsoperationen aufgezeichnet und in entsprechende Operationen auf dem Zielmodell transformiert. Anschließend werden diese Transformationen auf dem Zielmodell ausgeführt. In einem finalen Schritt überprüft eine zuvor definierte Äquivalenzrelation, ob die so durchgeführte Modellsynchronisation erfolgreich, in diesem Fall also äquivalenzerhaltend, ausgeführt werden konnte. Zur Erstellung einer Modellsynchronisation zwischen zwei Modellen müssen in dem dort vorgestellten Ansatz insgesamt sieben verschiedene Schritte durchgeführt werden. Soll eine Modellsynchronisation von den Benutzern an spezielle Bedürfnisse einfach anpassbar sein, dann ist dies sicherlich zu komplex. Wie die Autoren selbst anmerken, kann eine implizite Modellsynchronisation in der Praxis nicht alle Synchronisationsszenarien abdecken.

Hearnden et al. erweitern in ihrer Arbeit einen deklarativen, auf Logik basierenden Transformationsansatz, um Änderungen in einem Quellmodell inkrementell an ein Zielmodell weiter zu propagieren und damit eine Modellsynchronisation durchzuführen [HLR06]. Der Ansatz zeichnet die Ausführung einer Modelltransformation auf und speichert diese Aufzeichnung in einem so genannten *Execution-Record*. Änderungen im Quellmodell werden anschließend dem Execution-Record zugeordnet, so dass eine Berechnung von Aktualisierungen im Zielmodell ermöglicht wird. Diese Aktualisierungen synchronisieren das Zielmodell mit dem geänderten Quellmodell. Die vorgestellte Lösung hat den Nachteil, dass der Execution-Record sehr viel Speicherplatz benötigt, so dass zur Synchronisation größerer Modelle weitere Optimierungen des Execution-Records notwendig sind. Darüber hinaus ist nicht ersichtlich, ob zur bidirektionalen Modellsynchronisation ein Execution-Record ausreichend ist, oder ob für jede Synchronisationsrichtung ein eigener Execution-Record benötigt wird.

Ein anderer Zweig der Forschungsaktivitäten beschäftigt sich mit der Synchronisation von Modellen mit dem daraus generierten Code. Es existieren viele Werkzeugumgebungen, die zum Beispiel ein Codegerüst aus Klassendiagrammen generieren, oder die ein Klassendiagramm aus Code wiedergewinnen und anschließend versuchen, beide miteinander synchron zu halten [Bor06, IBM]. In den meisten Werkzeugen ist diese Synchronisation hart codiert. Häufig existiert nur ein gemeinsames Implementierungsmodell und

beide, das heißt, sowohl der Code als auch das eigentliche Modell, sind nur spezielle Sichten auf dieses gemeinsame Implementierungsmodell. Die Synchronisation wird dann häufig auf Grundlage des Model-View-Controller Paradigmas [PMD05] realisiert, in der dann die Sichten aktualisiert werden sobald das Implementierungsmodell geändert wurde. Dieser Ansatz erlaubt jedoch nicht die Anpassung der Abbildung zwischen dem Modell und dem Code an benutzer- oder firmenspezifische Bedürfnisse.

Andere Ansätze zur automatischen Synchronisation von Modellen und Code schlagen eine Kombination von Forward- und Reverse-Engineering-Techniken vor. Diese Vorgehensweise wird auch als Round-Trip-Engineering bezeichnet [Aßm03, HL03]. Der Nachteil dieses Ansatzes ist, dass zu einer Forward-Engineering-Funktion eine inverse Funktion für das Reverse-Engineering berechnet werden muss. Dies ist allerdings nur dann möglich, wenn es sich bei den Modell-zu-Code Abbildungen um bijektive Funktionen handelt. Dies ist jedoch selten der Fall [SK04]. Erschwerend kommt hinzu, dass der Ansatz versagt, falls ein Entwickler die Implementierung verfeinert und beispielsweise ein generiertes Codeskelett vervollständigt. Werden anschließend auch Änderungen am Modell vorgenommen, so führt ein erneutes Forward-Engineering dazu, dass die Änderungen im Code überschrieben werden.

8.3 Ansätze zur Vereinfachung der Spezifikation

In den meisten visuellen Ansätzen erfolgt die Spezifikation in der abstrakten Syntax der beteiligten Modellierungssprachen, die durch entsprechende Metamodelle definiert sind (z. B. [VVP02, VAS04, MB03]). Die graphische Spezifikation ist gegenüber der rein textuellen Spezifikation viel übersichtlicher und leichter zu verstehen, da die korrespondierenden Teile der Modellierungssprachen in einem Diagramm zueinander in Beziehung gesetzt werden. Bei großen und komplexen Metamodellen kann aber auch diese Art der Spezifikation groß und unübersichtlich werden. Dieses Problem wurde lange Zeit ignoriert. Daher existieren bisher nur wenige Lösungsansätze, die das Ziel verfolgen, die Spezifikationstechnik zu vereinfachen und übersichtlicher zu machen. In diesem Abschnitt werden diese Arbeiten kurz vorgestellt.

8.3.1 Kompakte Repräsentation von Modelltransformationen

Um unübersichtliche Spezifikationen zu vermeiden, stellt Bettin einige Ideen für eine kompaktere Repräsentation von Metamodellen vor [Bet03]. Ein Vorschlag ist beispielsweise, eine Komposition zwischen Klassen nicht mit einer Kompositionsassoziation zwischen diesen Klassen darzustellen, sondern die komponierte Klasse direkt eingebettet in der übergeordneten Klasse darzustellen. Auf Grundlage dieser kompakteren und übersichtlicheren[5] Darstellung von Metamodellen schlägt der Autor eine mögliche Notation für Modelltransformationen vor. Die Arbeit stellt allerdings nur einige wenige sehr grobe Ideen vor, die darüber hinaus nur an einem sehr einfachen Beispiel präsentiert werden. Aufgrund der Tatsache, dass die vorgeschlagene Notation nicht an größeren und komplexeren Beispielen evaluiert wurde und daher auch keine praktischen Erfahrungen mit dieser Notation gemacht werden konnten, ist diese Arbeit eher als Weckruf für die bestehende Problematik zu verstehen und weniger als eine praktikable Lösung des vorliegenden Problems.

Baar und Whittle haben untersucht, inwieweit die konkrete Syntax der zugrundeliegenden Modellierungssprachen geeignet ist, um Modelltransformationsregeln zu spezifizieren [BW06]. Dabei sind die Autoren schnell an Grenzen gestoßen, da beispielsweise zu abstrakten Klassen in den meisten Fällen keine graphische Repräsentation in der konkreten Syntax der Sprache existiert, aber Instanzen dieser Klassen in Mustern der Transformationsregeln sehr wohl vorkommen können. Um die Transformationsregeln dennoch in der konkreten Syntax der beteiligten Modellierungssprachen spezifizieren zu können, schlagen die Autoren vor, die Metamodelle zu diesem Zweck anzupassen und eine neue, speziell für die Spezifikation der Transformationsregeln geeignete konkrete Syntax zu entwickeln. Dies ist mit einem sehr hohen Aufwand verbunden und stellt – wie die Autoren anmerken – den Flaschenhals ihres Ansatzes dar.

8.3.2 Spezifikation durch Beispiele

Wimmer et al. haben einen Ansatz vorgeschlagen, bei dem die Korrespondenzbeziehungen zwischen Modellen in der konkreten Syntax der Modelle

[5] Die Entscheidung, ob diese Art der Darstellung tatsächlich kompakter und damit übersichtlicher ist, muss jeder Leser alleine für sich treffen, da dies rein subjektiv und daher nicht bewertbar ist.

mit Hilfe von Beispielen definiert wird. Anschließend werden die Transformationsregeln nahezu automatisch aus den definierten Korrespondenzbeziehungen abgeleitet [WSKK07]. Zu diesem Zweck muss der Entwickler der Modelltransformation zuerst semantisch zueinander korrespondierende Modelle liefern, die alle Konzepte der beteiligten Modellierungssprachen enthalten. In einem zweiten Schritt werden die Korrespondenzbeziehungen zwischen den einzelnen Modellelementen spezifiziert. Aus dieser Spezifikation und einer zuvor gegebenen Abbildung zwischen der abstrakten und konkreten Syntax der Modellierungssprachen werden in einem dritten Schritt die Transformationsregeln synthetisiert.

Der Ansatz unterscheidet sich von unserem Ansatz (vergleiche Abschnitt 4.2) in der Art wie die Beispiele zu entwickeln sind. In dem Ansatz von Wimmer werden (1) ein Beispiel, das alle Konzepte der beteiligten Modellierungssprachen enthält, (2) benutzerdefinierte Korrespondenzbeziehungen zwischen den Modellelementen des Beispiels, sowie (3) eine Abbildung zwischen der abstrakten und der konkreten Syntax der beteiligten Modellierungssprachen benötigt, um die Transformationsregeln zu synthetisieren.

In unserem Ansatz reicht es aus, verschiedene kleinere Beispiele anzugeben, um daraus Transformationsregeln zu synthetisieren. Auch wenn in dem Ansatz von Wimmer ebenso mehrere kleine Beispiele angegeben werden können und dies sogar von den Autoren empfohlen wird, muss der Entwickler weiterhin die Korrespondenzbeziehungen in diesen Beispielen manuell hinzufügen. Im Gegensatz dazu reicht in unserem Ansatz die Angabe der zueinander korrespondierenden Beispiele aus. Außerdem wird in unserem Ansatz keine explizite Zuordnung der konkreten zur abstrakten Syntax benötigt - die Synthese findet ausschließlich auf der abstrakten Repräsentation der Beispiele statt, auch wenn die Beispiel selbst in der konkreten Syntax angegeben werden.

Der in [WSKK07] präsentierte Ansatz funktioniert nur zum Teil automatisch, da bei Mehrdeutigkeiten der Benutzer in den Synthesealgorithmus eingreifen und diese manuell auflösen muss. Darüber hinaus existiert zu dem vorgestellten Ansatz noch keine Implementierung, so dass nicht klar ist, ob die vorgestellte Idee zur Spezifikation mit Beispielen auch auf größere Modelltransformationen anwendbar und praktikabel durchführbar ist.

Ein anderer Ansatz zur Spezifikation von Modelltransformationen mit Beispielen stellt Varró vor [Var06]. Die Zielsetzung des Ansatzes ist vergleichbar zu unserem Ansatz und dem Ansatz von Wimmer et al., allerdings existieren deutliche Unterschiede in den zugrundeliegenden Konzepten. In [Var06] werden die Beispiele weiterhin in der abstrakten Syntax angegeben. In unse-

rem Ansatz wird hierfür die konkrete Syntax verwendet. Zusätzlich muss in [Var06] vom Benutzer eine initiale Abbildung zwischen den Modellelementen erstellt werden. Diese initiale Abbildung beschreibt kritische Situationen bei der Modelltransformation. Eine Schwierigkeit ist, diese kritischen Situationen im Vorfeld zu identifizieren. Darüber hinaus können die Abbildungen nur mit Hilfe von 1-zu-1 Beziehungen definiert werden. Um die Transformationsregeln aus der initialen Abbildung zu generieren, müssen zusätzliche Benutzerinteraktionen durchgeführt werden. Beispielsweise müssen weitere Abbildungen zwischen den Modellelementen vom Benutzer definiert werden, bevor weitere Regeln abgeleitet werden können. Auch muss der Benutzer die generierten Transformationsregeln von Hand verfeinern, um die Anzahl der generierten Regeln zu reduzieren. Dies bedeutet insbesondere, dass die Transformationsregeln in [Var06] iterativ entwickelt werden müssen, während in unserem Ansatz die iterative Spezifikation von Transformationsregeln ermöglicht, aber nicht zwingend vorausgesetzt wird.

8.4 Zusammenfassung

In diesem Kapitel haben wir uns mit den verwandten Arbeiten beschäftigt. Dabei sind wir zunächst auf Ansätze zur Modelltransformation und Modellintegration eingegangen, die ebenfalls auf der Technik der TGGs basieren. Hierbei mussten wir feststellen, dass keiner dieser Ansätze eine vollständig inkrementelle Modelltransformation, Modellintegration oder Modellsynchronisation unterstützt, wie wir sie in dieser Arbeit kennen gelernt haben. Andere Ansätze zur Modelltransformation, die nicht auf der Grundlage von TGGs beruhen, fokussieren auf eine batch-artige Ausführung. Sie sind meistens auf eine einzige Transformationsrichtung festgelegt und unterstützen keine Konzepte, um eine Modelltransformation für den Benutzer nachverfolgbar zu machen. Daher sind sie zur Realisierung einer inkrementellen Modellsynchronisation ungeeignet.

Mit der Modellsynchronisation haben sich bisher nur sehr wenige Arbeiten beschäftigt. Dabei wurden die Modellsynchronisationen meistens nur für ein spezielles Synchronisationsszenario und in nur eine zuvor festgelegte Synchronisationsrichtung betrachtet. Eine allgemeine Methode zur modellbasierten Entwicklung von bidirektionalen Modellsynchronisationen zwischen zwei beliebigen Modellen wurde in diesen Ansätzen nicht berücksichtigt. Insbesondere zur Synchronisation von Modell und Code wird heutzutage immer noch auf von Hand programmierte und damit hart codierte Algorithmen

zurückgegriffen. Eine einfache Parametrisierung und Anpassung einer so umgesetzten Modellsynchronisation ist damit nicht möglich – die Modellsynchronisation kann nur durch eine erneute Programmierung bzw. Anpassung der Algorithmen erfolgen.

Im letzten Teil dieses Kapitels haben wir Ansätze vorgestellt, die sich der Vereinfachung von Spezifikationen zur Modelltransformation widmen. Zwei der vorgestellten Ansätze schlagen hierzu eine neue bzw. angepasste Notation zur kompakten Repräsentation der Modelle vor. Wie die Autoren selbst anmerken, ist diese Vorgehensweise mit einigen Nachteilen verbunden. Die anderen Ansätze basieren auf der Spezifikation von Beispielen, die entweder in der konkreten oder der abstrakten Syntax angegeben werden. Im Gegensatz zu unserem Ansatz sind die Techniken jedoch entweder noch nicht evaluiert worden oder schränken den Benutzer durch viele Restriktionen bei der Entwicklung stark ein.

Kapitel 9

Zusammenfassung und Ausblick

In diesem Kapitel fassen wir die Ergebnisse dieser Arbeit zusammen und bewerten die erreichten Ziele. Anschließend geben wir einen Ausblick über mögliche Erweiterungen des hier vorgestellten Ansatzes.

9.1 Zusammenfassung

In dieser Arbeit wurde eine Technik zur inkrementellen Modellsynchronisation vorgestellt. Die Modellsynchronisation gleicht zwei zueinander in Beziehung stehende Modelle automatischen miteinander ab und löst damit eventuell vorhandene Inkonsistenzen zwischen den Modellen auf. Die vorgestellte Modellsynchronisation kann sowohl inkrementell als auch batch-artig ausgeführt werden. Darüber hinaus eignet sich der Ansatz zur Modellintegration, Modelltransformation und zur Codegenerierung.

Die Grundlage der vorgestellten Technik zur Modellsynchronisation bilden Tripel-Graph-Grammatiken (TGGs). Diese deklarative, formale und visuelle Spezifikationstechnik ist nicht neu – sie wurde bereits im Rahmen einiger anderer Arbeiten eingesetzt, um Modelle zu transformieren oder miteinander zu integrieren. Einige dieser Ansätze können Änderungen an einem Quellmodell inkrementell an ein dazugehöriges Zielmodell propagieren. Im Gegensatz zu dem hier vorgestellten Ansatz bezieht sich die inkrementelle Ausführung allerdings nur auf das Zielmodell. Zur Identifikation der Änderungen muss immer noch das gesamte Quell- sowie Korrespondenzmodell traversiert und analysiert werden. In dieser Arbeit hingegen wird die Analyse des Quellmodells lokal an den von Änderungen betroffenen Stellen durchgeführt. Eine Traversierung und Analyse des gesamten Quell- und Korrespondenzmodells ist nicht notwendig. Daher kann die in dieser Arbeit vorgestellte Technik als vollständig inkrementell bezeichnet werden.

Im Rahmen dieser Arbeit wurden eine Methode und dazugehörige Softwarewerkzeuge zur modellbasierten Entwicklung von Modellsynchronisationen vorgestellt. Hierzu haben wir in einem ersten Schritt das Problem der Modellsynchronisation untersucht und daraus Anforderungen an eine Modellsynchronisation formuliert.

Die wichtigste funktionale Anforderung ist die bidirektionale und inkrementelle Modellsynchronisation. Der entwickelte Algorithmus ist allerdings auch in der Lage, eine Modellintegration und Modelltransformation durchzuführen. Diese Eigenschaften sind hauptsächlich auf die eingesetzte Spezifikationstechnik der TGGs zurückzuführen. Die Modellsynchronisation kann sowohl batch-artig als auch inkrementell erfolgen, was hingegen dem entwickelten Algorithmus zuzurechnen ist. Die batch-artige Arbeitsweise wird hauptsächlich dazu eingesetzt, um eine Modellsynchronisation durch Modelltransformation auszuführen, bei der initial nur ein Modell vorhanden ist, oder um eine Modellintegration zwischen zwei bereits bestehenden aber noch nicht synchronisierten Modellen durchzuführen.

Die Anpassbarkeit der Modellsynchronisation wurde erreicht, indem die Spezifikation der Korrespondenzregeln von dem Ausführungsalgorithmus entkoppelt wurde. Dadurch ist ein Rahmenwerk entstanden, das mit den spezifizierten Korrespondenzregeln parametrisierbar ist. Zur Parametrisierung werden aus einer TGG-Spezifikation operationale Graphersetzungsregeln abgeleitet, aus denen wiederum ausführbarer Code generiert wird. Dieser Code wird dynamisch zum Ausführungsalgorithmus gebunden und sorgt so für eine individuelle Parametrisierung der Modellsynchronisation.

Bei der Evaluierung einiger Beispielspezifikationen hat sich gezeigt, dass bei (Modellierungs-) Sprachen mit komplexen Metamodellen sehr viele und auch sehr umfangreiche Korrespondenzregeln benötigt werden. Die Spezifikation dieser Regeln kann daher sehr aufwändig werden. Aus diesem Grund wurden in dieser Arbeit verschiedene Ansätze erarbeitet, die eine Spezifikation erleichtern.

Bei einem dieser Ansätze werden die Korrespondenzbeziehungen zwischen Modellen mit Hilfe von Beispielpaaren angegeben. Hierzu müssen zunächst Beispiele der zueinander in Beziehung stehenden Modelle entworfen werden. Die entsprechenden Korrespondenzregeln werden anschließend automatisch aus diesen Beispielpaaren synthetisiert. Der Synthesealgorithmus erlaubt jederzeit eine manuelle Anpassung der synthetisierten Regeln durch den Benutzer und berücksichtigt diese Änderungen in den darauf folgenden Schritten. Der in dieser Arbeit vorgestellte Synthesealgorithmus ermöglicht somit eine iterative und inkrementelle Synthese der Korrespondenzregeln.

Im Rahmen der vorliegenden Arbeit wurde zusätzlich der Einsatz von TGGs zur Spezifikation von Modell-zu-Text Beziehungen untersucht. Diese Beziehungen können beispielsweise zur Codegenerierung verwendet werden. Dabei wurden zwei Möglichkeiten in Betracht gezogen. Die erste Variante besteht darin, den Text bzw. den Code als Modell zu interpretieren und die Spezifikation auf der Grundlage eines Metamodells der Sprache durchzuführen. Dieses Metamodell kann aus der Grammatik der Sprache abgeleitet werden. Die zweite Variante kombiniert TGGs mit Textschablonen. Dabei wird die TGG zur Abbildung der Modellstruktur auf einzelne Codefragmente genutzt. Die Codefragmente selbst werden mit Hilfe von Textschablonen beschrieben. Die Spezifikation eines feingranularen Metamodells der Sprache ist damit nicht nötig. Dies trägt signifikant zur Senkung des benötigten Spezifikationsaufwands bei.

Die vorliegende Arbeit wurde zeitgleich zum aufkommenden Standard Query/View/Transformation (QVT) der Object Management Group (OMG) erstellt. Der Standard definiert neben einer operationalen auch eine deklarative Transformationssprache, für die lange Zeit keine Werkzeugunterstützung existiert hat. Im Rahmen dieser Arbeit wurde gezeigt, wie der QVT-Standard mit Hilfe der TGGs formalisiert und realisiert werden kann. Dadurch werden TGGs für einen weiten Anwenderkreis zugänglich, der zunehmend auf die OMG-Standards setzt.

Zur Erprobung der inkrementellen Modellsynchronisation wurden mehrere Werkzeuge für Fujaba4Eclipse prototypisch entwickelt. Die umgesetzte Werkzeugunterstützung wurde anhand einiger Beispiele auf ihre Praktikabilität evaluiert. Dabei konnte gezeigt werden, dass die entwickelte Modellsynchronisation nicht auf Fujaba und Eclipse beschränkt ist, sondern sich leicht in andere Werkzeuge integrieren lässt. Durch diese Integration kann eine Interoperabilität dieser Werkzeuge erreicht werden, die beim Aufbau heterogener Werkzeuglandschaften von großem Nutzen ist.

Das Hauptanwendungsgebiet für Modellsynchronisationen sind große Modelle. Auf der Grundlage der prototypisch erstellten Werkzeuge wurden Leistungsmessungen durchgeführt. Die Leistungsmessungen belegen, dass die Modellsynchronisation mit dem hier vorgestellten Ansatz effizient durchführbar ist und auch bei großen Modellen sehr gut skaliert. Damit konnte gezeigt werden, dass die Synchronisation großer Modelle mit vertretbarem Aufwand möglich ist.

Insgesamt haben wir in dieser Arbeit – aufbauend auf dem Ansatz der Tripel-Graph-Grammatiken – eine Technik zur Modellsynchronisation vorgeschlagen, mit der inkrementelle Modellsynchronisationen durchgängig mo-

dellbasiert entwickelt und realisiert werden können. Mit der Bereitstellung und Anwendung verschiedener Methoden, Notationen und Werkzeuge konnte daher belegt werden, dass die Entwicklung von Werkzeugen zur inkrementellen Modellsynchronisation mit relativ geringem Aufwand möglich und praktikabel ist.

9.2 Ausblick

Am Ende einer Arbeit bleiben immer Wünsche und Ideen übrig, die nicht berücksichtigt werden konnten. So können beispielsweise sowohl der beschriebene Algorithmus zur Modellsynchronisation als auch die verwendete Spezifikationstechnik in verschiedener Hinsicht erweitert und verbessert werden. Einige dieser Ideen werden nachfolgend kurz vorgestellt.

Der in dieser Arbeit vorgestellte Algorithmus zur Modellsynchronisation arbeit vollständig inkrementell und kann entweder automatisch nach jeder Änderung oder auf Anforderung durch den Benutzer ausgeführt werden. Darüber hinaus kann eine Modellsynchronisation bidirektional stattfinden. Allerdings werden hierfür zwei unidirektionale Modellsynchronisationen in entgegengesetzter Richtung ausgeführt, d. h., der Algorithmus wird für jede Richtung separat ausgeführt. Hier wäre es jedoch wünschenswert, dass der Algorithmus in der Lage ist, eine bidirektionale Modellsynchronisation in einem einzigen Schritt durchzuführen. Ein Problem hierbei entsteht jedoch, wenn nicht sofort nach jeder Änderung synchronisiert wird, sondern ein Benutzer Änderungen an beiden Modellen vornehmen kann, ohne dass er zwischendurch die Modelle miteinander synchronisiert. In diesem Fall können die vom Benutzer durchgeführten Änderungen zueinander in Konflikt stehen. Hier ist zu untersuchen, wie solche Konflikte während einer Modellsynchronisation erkannt und wie mit ihnen umgegangen wird. Die in dieser Arbeit eingeführte Notation für Bedingungen ermöglicht bereits eine Erkennung von Konflikten zwischen „parallel“ geänderten Attributwerten. In der Arbeit von Simon Becker wurde ein Algorithmus mit Benutzerinteraktionen vorgestellt, der mit Konflikten zwischen Regelanwendungen umgehen kann [Bec07]. Hier wäre zu untersuchen, wie die beiden Ansätze miteinander kombiniert werden könnten, um auch strukturelle Konflikte während eines einzigen Durchlaufs einer Modellsynchronisation erkennen und beheben zu können.

Ein anderer möglicher Anknüpfpunkt für weiter gehende Untersuchungen bildet die Spezifikationssprache selbst. In dieser Arbeit wurden TGGs eingesetzt, um Beziehungen zwischen Modellen zu beschreiben, die in ihrer

Struktur sehr ähnlich sind. Für Modelle, die strukturell voneinander stark abweichen, sind TGGs nicht so gut geeignet. Hier wäre zu untersuchen, wie TGGs erweitert und mit anderen Ansätzen, wie z. B. operationalen Spezifikationstechniken, zu einem hybriden Ansatz kombiniert werden können, um die Ausdrucksstärke der TGGs zu erhöhen.

Bisher wurden TGGs in der abstrakten Syntax formuliert. Dadurch können die TGG-Regeln – verglichen mit den zugehörigen Modellen in ihrer konkreten Syntax – sehr groß und unübersichtlich werden. Um diesem Problem zu begegnen, haben wir in dieser Arbeit einen Ansatz zur Spezifikation durch Beispielpaare vorgestellt, aus denen die TGG-Regeln automatisch synthetisiert werden.

Als eine echte Alternative hierzu könnte sich eine direkte Spezifikation von TGG-Regeln in der konkreten Syntax der beteiligten Modelle herausstellen. Bisher war ein solcher Ansatz allerdings nicht praktikabel, da in Abhängigkeit der beteiligten Modelle ein spezieller Regeleditor entwickelt werden musste, der die konkrete Syntax beider Modelle gleichzeitig unterstützt. Die manuelle Entwicklung eines solchen Editors ist mit erheblichem Aufwand verbunden. Mit dem Aufkommen von Rahmenwerken wie z. B. EMF oder GMF wird eine nahezu automatische Erzeugung graphischer Editoren ermöglicht. Hier wäre zu untersuchen, inwieweit ein solcher – an die beteiligten Modelle speziell angepasster – Regeleditor automatisch generiert werden kann. Dies würde den Aufwand signifikant verringern und diesen Ansatz praktikabel machen. Dies ist jedoch noch zu erforschen.

Ein weiteres, offenes Forschungsgebiet stellt die Validierung und Verifikation von Modellsynchronisationen dar. Mit der Technik der TGGs werden Korrespondenzbeziehungen zwischen zwei Modellen in einer lokalen Art und Weise definiert, so dass auf dieser Grundlage beispielsweise ein Nachweis der semantischen Korrektheit möglich ist. Ein erster Ansatz hierzu wurde im Rahmen dieser Arbeit an einem Beispiel vorgestellt, bei dem ein I/O-Automat in SPS-Code übersetzt wurde. Für umfangreichere Spezifikationen und Szenarien, in denen alle hier vorgestellten Konzepte vorkommen, sind jedoch weitere Untersuchungen notwendig. Dies gilt auch für Techniken zur Validierung durch Tests.

Literatur

[AC07] Antkiewicz, Michal ; Czarnecki, Krzysztof: Design Space of Heterogeneous Synchronization. In: *Proceedings of 2nd Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE07)*, 2007, S. 1–41. – Draft submitted to post-proceedings.

[Alt08] Altan, Güzide S.: *On the Usability of Triple Graph Grammars for the Transformation of Business Process Models - An Evaluation based on FUJABA*, Technische Universität Wien, Magisterarbeit, Januar 2008

[AMD] *AndroMDA.* `http://www.andromda.org/`. : *AndroMDA.* `http://www.andromda.org/`. – Stand: Mai 2008

[ANRS06] Aizenbud-Reshef, N. ; Nolan, B. T. ; Rubin, J. ; Shaham-Gafni, Y.: Model traceability. In: *IBM System Journal* 45 (2006), Juli, Nr. 3, S. 515–526

[ARC] Interactive Objects (Hrsg.): *ArcStyler.* `http://www.interactive-objects.com/`. Interactive Objects. – Stand: Mai 2008

[Aßm03] Assmann, Uwe: Automatic Roundtrip Engineering. In: *Electronic Notes in Theoretical Computer Science* 82 (2003), April, Nr. 5, S. 1–9

[ASU86] Aho, Alfred V. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools.* Reading, MA : Addison-Wesley, 1986

[Bak06] Baksmeier, Jörg: *Inkrementelle Modellsynchronisation mit Tripel-Graph-Grammatiken*, Universität Paderborn, Diplomarbeit, November 2006

[Bal91] Balzer, Robert: Tolerating inconsistency. In: *Proceedings of the 13th international conference on Software engineering (ICSE)*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1991, S. 158–165

[BDTM+06] Baudry, Benoit ; Dinh-Trong, Trung ; Mottu, Jean-Marie ; Simmonds, Devon ; France, Robert ; Ghosh, Sudipto ; Fleurey, Franck ; Le Traon, Yves: Model transformation testing challenges. In: *Proceedings of ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing*, 2006

[Bec07] Becker, Simon M.: *Integratoren zur Konsistenzsicherung von Dokumenten in Entwicklungsprozessen*, RWTH Aachen, Dissertation, 2007

[Bet03] Bettin, Jorn: Ideas for a Concrete Visual Syntax for Model-to-Model Transformation. In: *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003

[BGMT] Brun, Cedric ; Goubet, Laurent ; Musset, Jonathan ; Toulme, Antoine ; The Eclipse Foundation (Hrsg.): *Eclipse Modeling Framework Technology Project - EMF Compare.* `http://www.eclipse.org/modeling/emft/`. The Eclipse Foundation. – Stand: Mai 2008

[BGN+04] Burmester, Sven ; Giese, Holger ; Niere, Jörg ; Tichy, Matthias ; Wadsack, Jörg P. ; Wagner, Robert ; Wendehals, Lothar ; Zündorf, Albert: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In: *International Journal on Software Tools for Technology Transfer (STTT)* 6 (2004), August, Nr. 3, S. 203–218

[BHP00] Bernstein, Phillip A. ; Halevy, Alon Y. ; Pottinger, Rachel A.: A vision for management of complex models. In: *SIGMOD Record (ACM Special Interest Group on Management of Data)* 29 (2000), Nr. 4, S. 55–63

[BLW04] Becker, Simon ; Lohmann, Sebastian ; Westfechtel, Bernhard: Rule Execution in Graph-Based Incremental Interactive Integration Tools. In: *Proc. Intl. Conf. on Graph*

Transformations (ICGT 2004) Bd. 3256. Berlin/Heidelberg : Springer-Verlag, 2004 (LNCS), S. 22–38

[Bor06] Borland GmbH (Hrsg.): *Together Architect.* `http://www.borland.com/us/products/together/`. Borland GmbH, 2006. – Stand: März 2007

[BW06] Baar, Thomas ; Whittle, Jon: On the Usage of Concrete Syntax in Model Transformation Rules. In: *6th International Andrei Ershov Memorial Conference Perspectives of System Informatics* Bd. 4378. Berlin/Heidelberg : Springer-Verlag, Juni 2006 (Lecture Notes in Computer Science (LNCS)), S. 84–97

[CC90] Chikofsky, Elliot J. ; Cross II, James H.: Reverse Engineering and Design Recovery: A Taxonomy. In: *IEEE Software* 7 (1990), Januar, Nr. 1, S. 13–17

[Ceb07] Cebeci, Yascha: *Spezifikation von Codetransformationen anhand konkreter Beispiele*, Universität Paderborn, Studienarbeit, August 2007

[CGP00] Clarke, Edmund M. ; Grumberg, Orna ; Peled, Doron A.: *Model Checking.* The MIT Press, 2000

[CH03] Czarnecki, Krzysztof ; Helsen, Simon: Classification of Model Transformation Approaches. In: *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003

[CH06] Czarnecki, Krzysztof ; Helsen, Simon: Feature-based survey of model transformation approaches. In: *IBM System Journal* 45 (2006), July, Nr. 3

[CoG] Manyeta Informatics (Hrsg.): *Codagen Architect.* `http://www.manyeta.com/en/Technology/codagen_architect_v3.2`. Manyeta Informatics. – Stand: Mai 2008

[CS06] Claus, Volker ; Schwill, Andreas: *Duden: Informatik A-Z. Fachlexikon für Studium, Ausbildung und Beruf.* 4. Auflage. Mannheim, Leipzig, Wien, Zürich : Dudenverlag, 2006

[Det06] Detten, Markus von: *Template-basierte Codegenerierung für Speicherprogrammierbare Steuerungen*, Universität Paderborn, Studienarbeit, April 2006

[DGS05] Demathieu, S. ; Griffin, C. ; Sendall, S.: Model Transformation with the IBM Model Transformation Framework. In: *developerWorks. IBM's resource for developers.* `http://www.ibm.com/developerworks/` (2005), Mai

[Dud06] Dudenredaktion (Hrsg.): *Duden: Das Fremdwörterbuch.* Bd. 5. 9., aktualisierte Auflage. Mannheim, Leipzig, Wien, Zürich : Dudenverlag, 2006

[EC01] Easterbrook, Steve ; Chechik, Marsha: 2nd international workshop on living with inconsistency (IWLWI01). In: *SIGSOFT Softw. Eng. Notes* 26 (2001), Nr. 6, S. 76–78

[Eck07] Eckes, Raimund: *Augmented Reality – basiertes Verfahren zur Unterstützung des Anlaufprozesses von automatisierten Fertigungssystemen*, Universität Paderborn, Dissertation, April 2007

[EKHG01] Engels, Gregor ; Küster, Jochem M. ; Heckel, Reiko ; Groenewegen, Luuk: A methodology for specifying and analyzing consistency of object-oriented behavioral models. In: *SIGSOFT Softw. Eng. Notes* 26 (2001), September, Nr. 5, S. 186–195

[EKT08] Ehrig, Karsten ; Küster, Jochen ; Taentzer, Gabriele: Generating instance models from meta models. In: *Software and Systems Modeling* (2008), Juli, S. 1–22. – Online First. DOI: http://dx.doi.org/10.1007/s10270-008-0095-y

[ELI] *Abstract Syntax Tree Unparsing. Eli Online Documents Version 4.4.* `http://eli-project.sourceforge.net/elionline/idem_toc.html`. : *Abstract Syntax Tree Unparsing. Eli Online Documents Version 4.4.* `http://eli-project.sourceforge.net/elionline/idem_toc.html`. – Stand: Mai 2008

[FBMT08] Fleurey, Franck ; Baudry, Benoit ; Muller, Pierre A. ; Traon, Yves: Qualifying input test data for model transformations. In: *Software and Systems Modeling* (2008), Juli, S.

1–19. – Online First. DOI: http://dx.doi.org/10.1007/s10270-007-0074-8

[FBV06] Fabro, Marcos Didonet D. ; Bézivin, Jean ; Valduriez, Patrick: Weaving Models with the Eclipse AMW plugin. In: *Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany*, 2006

[FNT98] Fischer, Thorsten ; Niere, Jörg ; Torunski, Lars: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*, Universität Paderborn, Diplomarbeit, Juli 1998

[FNTZ98] Fischer, Thorsten ; Niere, Jörg ; Torunski, Lars ; Zündorf, Albert: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Engels, G. (Hrsg.) ; Rozenberg, G. (Hrsg.): *Proceedings of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany.* Berlin/Heidelberg : Springer-Verlag, November 1998 (LNCS 1764), S. 296–309

[Fra06] France Telecom (Hrsg.): *SmartQVT: An open source model transformation tool implementing the MOF 2.0 QVT-Operational language.* `http://smartqvt.elibel.tm.fr/`. France Telecom, 2006. – Stand: März 2007)

[FSB04] Fleurey, Franck ; Steel, Jim ; Baudry, Benoit: Validation in Model-Driven Engineering: Testing Model Transformations. In: *Proceedings of the First International Workshop on Model, Design and Validation.*, 2004, S. 29–40

[Fuj] University of Paderborn, Germany (Hrsg.): *Fujaba Tool Suite.* `http://www.fujaba.de/`. University of Paderborn, Germany. – Stand: November 2007

[Geb06] Geburzi, Alexander: *Synthese von Modelltransformationsregeln aus Übersetzungsbeispielen*, Universität Paderborn, Diplomarbeit, November 2006

[GGL+06] Giese, Holger ; Glesner, Sabine ; Leitner, Johannes ; Schäfer, Wilhelm ; Wagner, Robert: Towards Verified Model Transformations. In: Hearnden, David (Hrsg.) ; Süss, Jörn Guy (Hrsg.) ; Baudry, Benoît (Hrsg.) ; Rapin, Nicolas (Hrsg.): *Proceedings of the 3rd International Workshop on Model Development, Validation and Verification (MoDeV²a), Genova, Italy*, Le Commissariat à l'Energie Atomique - CEA, October 2006, S. 78–93

[GH08] Giese, Holger ; Hildebrandt, Stephan: Incremental model synchronization for multiple updates. In: *International Workshop on Graph and Model Transformations (GRaMoT)*. New York, NY, USA : ACM, 2008, S. 1–8

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[GL04] Guerra, Esther ; Lara, Juan de: Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation. In: *International Conference on Graph Transformation (ICGT'2004)* Bd. 3265. Berlin/Heidelberg : Springer-Verlag, 2004 (LNCS), S. 54–69

[GM80] Ghezzi, Carlo ; Mandrioli, Dino: Augmenting Parsers to Support Incrementality. In: *Journal of the ACM* 27 (1980), Juli, Nr. 3, S. 564–579

[GMW06] Giese, Holger ; Meyer, Matthias ; Wagner, Robert: A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In: Giese, Holger (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *Proceedings of the 4th International Fujaba Days 2006, Bayreuth, Germany* Bd. tr-ri-06-275, University of Paderborn, September 2006 (Technical Report)

[GPR05] Gruhn, Volker ; Pieper, Daniel ; Röttgers, Carsten: *MDA - Effektives Siftware-Engineering mit UML 2 und Eclipse*. Berlin/Heidelberg : Springer-Verlag, 2005

[Gre06] Greenyer, Joel: *A Study of Model Transformation Technologies: Reconciling TGGs with QVT*, Universität Paderborn, Diplomarbeit, July 2006

[Gri04] Griffin, Catherine ; IBM alphaWorks (Hrsg.): *Model Transformation Framework (MTF).* `http://www.alphaworks.ibm.com/tech/mtf/`. IBM alphaWorks, 2004. – Stand: Mai 2008

[GSCK04] Greenfield, Jack ; Short, Keith ; Cook, Steve ; Kent, Stuart: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004

[GW09] Giese, Holger ; Wagner, Robert: From model transformation to incremental bidirectional model synchronization. In: *Software and Systems Modeling* 8 (2009), Februar, Nr. 1, S. 21–43. – Online First. DOI: `http://dx.doi.org/10.1007/s10270-008-0089-9`

[Her03] Herrington, Jack: *Code Generation in Action.* Manning Publications, 2003

[Hil07] Hildebrandt, Stephan: *Effiziente Modellsynchronisation mit Tripel-Graph-Grammatiken durch Wiederverwendung von Transformationsergebnissen*, Hasso-Plattner-Institut für Softwaresystemtechnik GmbH, Masterarbeit, Oktober 2007

[HL03] Henriksson, Anders ; Larsson, Henrik: A definition of round-trip engineering / Universität Linkopings. 2003. – Forschungsbericht

[HLR06] Hearnden, David ; Lawley, Michael ; Raymond, Kerry: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Nierstrasz, Oscar (Hrsg.) ; Whittle, Jon (Hrsg.) ; Harel, David (Hrsg.) ; Reggio, Gianna (Hrsg.): *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings* Bd. 4199. Berlin/Heidelberg : Springer-Verlag, Oktober 2006 (Lecture Notes in Computer Science), S. 321–335

[HR00] Harel, David ; Rumpe, Bernhard: Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff / The Weizmann Institute of Science. Weizmann Science Press of Israel, August 2000 (MCS00-16). – Forschungsbericht

[IBM] IBM (Hrsg.): *Rational Rose Developer for Java.* `http://www-306.ibm.com/software/awdtools/developer/rose/java/`. IBM. – Stand: März 2007

[IEC03] International Electrotechnical Commission (IEC) (Hrsg.): *Speicherprogrammierbare Steuerungen, Teil 3: Programmiersprachen. (IEC 61131-3:2003) Deutsche Fassung DIN EN 61131-3:2003.* International Electrotechnical Commission (IEC), 2003

[IEE90] IEEE Computer Society (Hrsg.): *Standard Glossary of Software Engineering Terminology.* IEEE Computer Society, 1990. – IEEE 610.12-1990

[IK04a] Ivkovic, Igor ; Kontogiannis, Kostas: Model synchronization as a problem of maximizing model dependencies. In: *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.* New York, NY, USA : ACM Press, 2004, S. 222–223

[IK04b] Ivkovic, Igor ; Kontogiannis, Kostas: Tracing Evolution Changes of Software Artifacts through Model Synchronization. In: *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance.* Washington, DC, USA : IEEE Computer Society, 2004, S. 252–261

[IKV] ikv++ technologies ag (Hrsg.): *medini QVT.* `http://www.ikv.de/`. ikv++ technologies ag. – Stand: Dezember 2008

[ITU96] International Telecommunication Union (ITU), Geneva (Hrsg.): *ITU-T Recommendation Z.100: Specification and Description Language (SDL).* International Telecommunication Union (ITU), Geneva, 1994 + Addendum 1996

[Jam] *The Jamda Project.* `http://jamda.sourceforge.net/`. : *The Jamda Project.* `http://jamda.sourceforge.net/`. – Stand: Mai 2008

[JCC] *Java Compiler Compiler - The Java Parser Generator.* `https://javacc.dev.java.net/doc/`. : *Java Compiler Compiler -*

The Java Parser Generator. `https://javacc.dev.java.net/doc/`. – Stand: Mai 2008

[JDT] The Eclipse Foundation (Hrsg.): *Java Development Tools.* `http://www.eclipse.org/jdt/`. The Eclipse Foundation. – Stand: Mai 2008

[JET] The Eclipse Foundation (Hrsg.): *Java Emitter Templates.* `http://www.eclipse.org/modeling/m2t/`. The Eclipse Foundation. – Stand: Mai 2008

[JTB] UCLA Compilers Group (Hrsg.): *Java TreeBuilder.* `http://compilers.cs.ucla.edu/jtb/`. UCLA Compilers Group. – Stand: Mai 2008

[KA06] Küster, Jochen M. ; Abd-El-Razik, Mohamed: Validation of Model Transformations - First Experiences using a White Box Approach. In: *Proceedings of the 3rd Workshop on Model Design and Validation (MoDeV2a)*, 2006, S. 62–77

[Kas94] Kastens, Uwe: Construction of application generators using Eli / Universität Paderborn. 1994 (Reihe Informatik tr-ri-94-143). – Forschungsbericht

[KC05a] Kim, Chang Hwan P. ; Czarnecki, Krzysztof: Synchronizing Cardinality-Based Feature Models and Their Specializations. In: Hartman, Alan (Hrsg.) ; Kreische, David (Hrsg.): *Proceedings of the First European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Nuremberg, Germany, November 7-10* Bd. 3748, Springer-Verlag, November 2005 (Lecture Notes in Computer Science), S. 331–348

[KC05b] Kim, Chang Hwan P. ; Czarnecki, Krzysztof: Synchronizing Cardinality-Based Feature Models and Their Specializations. In: Hartman, Alan (Hrsg.) ; Kreische, David (Hrsg.): *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings* Bd. 3748. Berlin/Heidelberg : Springer-Verlag, November 2005 (Lecture Notes in Computer Science), S. 331–348

[KE96] Kemper, Alfons ; Eickler, André: *Datenbanksysteme: Eine Einführung*. München, Wien : Oldenbourg, 1996

[KHE03] Küster, J. M. ; Heckel, R. ; Engels, G.: Defining and validating transformations of UML models. In: *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC)*. Washington, DC, USA : IEEE Computer Society, 2003, S. 145–152

[KKS07] Klar, Felix ; Königs, Alexander ; Schürr, Andy: Model Transformation in the Large. In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*. New York, NY, USA : ACM Press, 2007, S. 285–294

[KL03] Kort, Jan ; Lammel, Ralf: Parse-Tree Annotations Meet Re-Engineering Concerns. In: *Proceedings of the* 3^{rd} *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)* (2003), S. 161–171

[Kön08] Königs, Alexander: *Model Integration and Transformation – A Triple Graph Grammar-based QVT Implementation*, Technische Universität Darmstadt, Dissertation, 2008

[Kön07] Könemann, Patrick: *Verbesserung eines modellbasierten Softwareentwicklungsprozesses mit Hilfe von Modellsynchronisation*, Universität Paderborn,Diplomarbeit, Diplomarbeit, März 2007

[KR04] Kardos, Martin ; Rammig, Franz J.: Model Based Formal Verification of Distributed Production Control Systems. In: Ehrig, H. (Hrsg.) ; Damm, W. (Hrsg.) ; Desel, J. (Hrsg.) ; Große-Rhode, M. (Hrsg.) ; Reif, W. (Hrsg.) ; Schnieder, E. (Hrsg.) ; Westkämper, E. (Hrsg.): *Integration of Software Specification Techniques for Applications in Engineering* Bd. 3147. Berlin/Heidelberg : Springer-Verlag, September 2004, S. 451–473

[KRW04] Kindler, Ekkart ; Rubin, Vladimir ; Wagner, Robert: An Adaptable TGG Interpreter for In-Memory Model Transfor-

mation. In: *Proceedings of the Fujaba Days 2004*. Darmstadt, Germany, September 2004, S. 35–38

[KS06] KÖNIGS, A. ; SCHÜRR, A.: Tool Integration with Triple Graph Grammars - A Survey. In: HECKEL, R. (Hrsg.): *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques* Bd. 148. Amsterdam : Elsevier Science Publ., 2006 (Electronic Notes in Theoretical Computer Science 1), S. 113–150

[Küs04a] KÜSTER, Jochen M.: *Consistency Management of Object-Oriented Behavioral Models*, Universität Paderborn, Dissertation, März 2004

[Küs04b] KÜSTER, Jochen M.: Systematic Validation of Model Transformations. In: *Proceedings of the 3rd UML Workshop in Software Model Engineering (WiSME 2004)*, 2004

[KW07] KINDLER, Ekkart ; WAGNER, Robert: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios / Universität Paderborn. 2007 (Reihe Informatik tr-ri-07-284). – Forschungsbericht. – 75 S.

[Lef95] LEFERING, Martin: *Integrationswerkzeuge in einer Softwareentwicklungsumgebung*, RWTH Aachen, Dissertation, 1995

[Lei06] LEITNER, Johannes: *Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken*, Universität Karlsruhe, Diplomarbeit, März 2006

[LEO08] LAMBERS, Leen ; EHRIG, Hartmut ; OREJAS, Fernando: Efficient Conflict Detection in Graph Transformation Systems by Essential Critical Pairs. In: *Electr. Notes Theor. Comput. Sci.* 211 (2008), S. 17–26

[LS96] LEFERING, Martin ; SCHÜRR, Andy: Specification of Integration Tools. In: NAGL, Manfred (Hrsg.): *Building Thightly-Integrated (Software) Development Environments: The IPSEN Approach* Bd. 1170. Berlin/Heidelberg : Springer-Verlag, 1996 (Lecture Notes in Computer Science), S. 324–334

[LT89] Lynch, Nancy A. ; Tuttle, Mark R.: An Introduction to Input/Output Automata. In: *CWI Quarterly* 2 (1989), Nr. 3, S. 219–249

[LTM+04] Lau, Terence C. ; Tong, Tack ; Mckegney, Ross ; Kontogiannis, Kostas ; Ivkovic, Igor ; Liew, Philip ; Zou, Ying ; Zhang, Qi ; Hung, Maokeng: Model synchronization for efficient software application maintenance. In: *Proceedings of the 20th IEEE International Conference on Software Maintenance, 11-14 September, 2004* (2004), S. 1

[LVA04] Lara, Juan de ; Vangheluwe, Hans ; Alfonseca, Manuel: Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. In: *Software and Systems Modeling* 3 (2004), Nr. 3, S. 194–209

[LZG05] Lin, Yuehua ; Zhang, Jing ; Gray, Jeff: A Testing Framework for Model Transformations. In: Beydeda, Sami (Hrsg.) ; Book, Matthias (Hrsg.) ; Gruhn, Volker (Hrsg.): *Model-Driven Software Development*, Springer-Verlag, Dezember 2005, S. 219–236

[M2M] The Eclipse Foundation (Hrsg.): *M2M-Projekt: Operational QVT*. `http://www.eclipse.org/m2m/`. The Eclipse Foundation. – Stand: Dezember 2008

[MB03] Marschall, Frank ; Braun, Peter: Model Transformations for the MDA with BOTL. In: *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*. Univeristy of Twente, June 2003 (CTIT Technical Report TR-CTIT-03-27)

[MG05] Mens, Tom ; Gorp, Pieter V.: A Taxonomy of Model Transformation. In: *International Workshop on Graph and Model Transformation (GraMoT)*. Tallinn, Estonia, September 2005

[MG06] Mens, Tom ; Gorp, Pieter V.: A Taxonomy of Model Transformation. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), März, S. 125–142

[Mor] Tata Consultancy Services (Hrsg.): *ModelMorf.* `http://www.tcs-trddc.com/ModelMorf/`. Tata Consultancy Services. – Stand: Mai 2008

[MTT08] Object Management Group (Hrsg.): *MOF Model to Text Transformation Language 1.0 (MOFM2T).* 140 Kendrick Street, Needham, MA 02494, USA: Object Management Group, Januar 2008. – Document formal/2008-01-16

[NER00] Nuseibeh, Bashar ; Easterbrook, Steve ; Russo, Alessandra: Leveraging Inconsistency in Software Development. In: *Computer* 33 (2000), Nr. 4, S. 24–29

[NK08a] Narayanan, Anantha ; Karsai, Gabor: Towards Verifying Model Transformations. 211 (2008), April, S. 191–200

[NK08b] Narayanan, Anantha ; Karsai, Gabor: Verifying Model Transformations by Structural Correspondence. In: *Electronic Communications of the EASST* 10 (2008), S. 1–14

[NNZ00] Nickel, Ulrich A. ; Niere, Jörg ; Zündorf, Albert: Tool demonstration: The FUJABA environment. In: *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, ACM Press, 2000, S. 742–745

[NSW+02] Niere, Jörg ; Schäfer, Wilhelm ; Wadsack, Jörg P. ; Wendehals, Lothar ; Welsh, Jim: Towards Pattern-Based Design Recovery. In: *Proceedings of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, ACM Press, Mai 2002, S. 338–348

[NSZ03] Nickel, Ulrich A. ; Schäfer, Wilhelm ; Zündorf, Albert: Integrative Specification of Distributed Production Control Systems for Flexible Automated Manufacturing. In: Nagl, Manfred (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *DFG Workshop: Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, Wiley-VCH Verlag GmbH and Co. KGaA, 2003, S. 179–195

[OAW] *openArchitectureWare Generator Framework (oAW).* `http://www.openarchitectureware.org/`. : *openArchi-*

tectureWare Generator Framework (oAW). `http://www.openarchitectureware.org/`. – Stand: Mai 2008

[OMG04] Object Management Group (Hrsg.): *UML Profile for enterprise distributed Object Computing (EDOC) - Metamodel and UML Profile for Java and EJB, v1.0.* 140 Kendrick Street, Needham, MA 02494, USA: Object Management Group, Februar 2004. – Document formal/04-02-02

[OpJ] Compuware (Hrsg.): *OptimalJ.* `http://www.compuware.de/products/optimalj/`. Compuware. – Stand: Mai 2008

[Par07] Parr, Terence: *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Raleigh, North Carolina : The Pragmatic Bookshelf, 2007

[PMD05] Paesschen, Ellen V. ; Meuter, Wolfgang D. ; D'Hondt, Maja: SelfSync: A Dynamic Round-Trip Engineering Environment. In: Briand, Lionel C. (Hrsg.) ; Williams, Clay (Hrsg.): *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings* Bd. 3713. Berlin/Heidelberg : Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 633–647

[Pra71] Pratt, Terrence W.: Pair Grammars, Graph Languages, and String-to-Graph-Translations. In: *Journal of Computer and System Sciences* 5 (1971), S. 560–595

[PSS98] Pnueli, A. ; Siegel, M. ; Singerman, E.: Translation validation. In: Steffen, B. (Hrsg.): *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems* Bd. 1384. Berlin/Heidelberg : Springer-Verlag, April 1998 (Lecture Notes in Computer Science (LNCS)), S. 151–166

[QVT08] Object Management Group (Hrsg.): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0.* 140 Kendrick Street, Needham, MA 02494, USA: Object Management Group, April 2008. – Document formal/2008-04-03

[RFP03] Object Management Group (Hrsg.): *OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP*. `http://www.omg.org/mda/`. Object Management Group, 2003

[RKRS05] Reiter, Th. ; Kapsammer, E. ; Retschitzegger, W. ; Schwinger, W.: Model Integration Through Mega Operations. In: *Workshop on Model-driven Web Engineering (2005)* (2005)

[Roh06] Rohe, Oliver: *Model Transformation by Interpreting Triple Graph Grammars: Evaluation and Case Study*, Universität Paderborn, Studienarbeit, Januar 2006

[Roz97] Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation*. Bd. 1. Singapore : World Scientific, 1997

[Sch94] Schürr, Andy: Specification of Graph Translators with Triple Graph Grammars. In: Tinhofer, G. (Hrsg.): *Proceedings of the 20^{th} International Workshop on Graph-Theoretic Concepts in Computer Science* Bd. 903. Heidelberg : Spinger-Verlag, Juni 1994 (Lecture Notes in Computer Science (LNCS)), S. 151–163

[SK04] Sendall, Shane ; Küster, Jochen: Taming Model Round-Trip Engineering. In: *Proceedings of Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications), Vancouver, Canada*, 2004

[SK08] Schürr, Andy ; Klar, Felix: 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In: Ehrig, Hartmut (Hrsg.) ; Heckel, Reiko (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.) ; Taentzer, Gabriele (Hrsg.): *Proc. of the 4^{th} International Conference on Graph Transformations (ICGT), Leicester, United Kingdom* Bd. 5214. Berlin/Heidelberg : Springer-Verlag, September 2008 (Lecture Notes in Computer Science (LNCS)), S. 411–425

[SS05] Saito, Yasushi ; Shapiro, Marc: Optimistic replication. In: *ACM Comput. Surv.* 37 (2005), March, Nr. 1, S. 42–81

[SWGE04] Schäfer, Wilhelm ; Wagner, Robert ; Gausemeier, Jürgen ; Eckes, Raimund: An Engineer's Workstation to support Integrated Development of Flexible Production Control Systems. In: Ehrig, H. (Hrsg.) ; Damm, W. (Hrsg.) ; Desel, J. (Hrsg.) ; Große-Rhode, M. (Hrsg.) ; Reif, W. (Hrsg.) ; Schnieder, E. (Hrsg.) ; Westkämper, E. (Hrsg.): *Integration of Software Specification Techniques for Applications in Engineering* Bd. 3147. Berlin/Heidelberg : Springer-Verlag, September 2004, S. 48–68

[SWZ99] Schürr, A. ; Winter, A. J. ; Zündorf, A.: The PROGRES approach: language and environment. (1999), S. 487–550

[SZK04] Song, Guanglei ; Zhang, Kang ; Kong, Jun: Model Management Through Graph Transformation. In: *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*. Washington, DC, USA : IEEE Computer Society, 2004, S. 75–82

[Tan95] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. 2. Auflage. München, Wien, London : Carl Hanser und Prentice-Hall International, 1995

[TBWK07] Treude, Christoph ; Berlik, Stefan ; Wenzel, Sven ; Kelter, Udo: Difference computation of large models. In: *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA : ACM, 2007, S. 295–304

[TXL] *The TXL Programming Language* `http://www.txl.ca/`. : *The TXL Programming Language* `http://www.txl.ca/`. – Stand: Mai 2008

[UML05] Object Management Group (OMG) (Hrsg.): *Unified Modeling Language: Superstructure Version 2.0*. 140 Kendrick Street, Needham, MA 02494, USA: Object Management Group (OMG), August 2005. – Document formal/05-07-04

[Var06] Varró, Dániel: Model Transformation by Example. In: Nierstrasz, Oscar (Hrsg.) ; Whittle, Jon (Hrsg.) ; Harel, David (Hrsg.) ; Reggio, Gianna (Hrsg.): *Proc. 9th International*

Conference on Model Driven Engineering Languages and Systems, MoDELS 2006, Genova, Italy, October 1-6, 2006. Bd. 4199. Berlin/Heidelberg : Springer-Verlag, Oktober 2006 (Lecture Notes in Computer Science), S. 410–424

[VAS04] Vizhanyo, Attila ; Agrawal, Aditya ; Shi, Feng: Towards Generation of Efficient Transformations. In: Karsai, Gabor (Hrsg.) ; Visser, Eelco (Hrsg.): *Generative Programming and Component Engineering: Proceedings of the Third International Conference (GPCE), Vancouver, Canada* Bd. 3286. Berlin/Heidelberg : Springer-Verlag, Oktober 2004 (Lecture Notes in Computer Science (LNCS)), S. 298–316

[Vas06] Vasarhelyi, Arpad: *Lösungsansätze für Modelltransformationen in einem Softwareentwicklungsprozess auf Basis des opProzesses*, Hochschule Darmstadt, Masterarbeit, April 2006

[VP03] Varró, Dániel ; Pataricza, András: Automated Formal Verification of Model Transformations. In: Jürjens, Jan (Hrsg.) ; Rumpe, Bernhard (Hrsg.) ; France, Robert (Hrsg.) ; Fernandez, Eduardo B. (Hrsg.): *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, Technische Universität München, September 2003 (Technical Report TUM-I0323), S. 63–78

[VTE] Apache Software Foundation (Hrsg.): *The Apache Velocity Project.* `http://velocity.apache.org/`. Apache Software Foundation. – Stand: Mai 2008

[VVP02] Varró, Dániel ; Varró, Gergely ; Pataricza, András: Designing the Automatic Transformation of Visual Languages. In: *Science of Computer Programming* 44 (2002), August, Nr. 2, S. 205–227

[W3C99] W3C (Hrsg.): *XSL Transformations (XSLT) Version 1.0.* `http://www.w3.org/TR/xslt/`. W3C, November 1999. – Stand: November 2007

[WG98] Wagner, Tim A. ; Graham, Susan L.: Efficient and flexible incremental parsing. In: *Transactions on Programming Languages and Systems* 20 (1998), Nr. 5, S. 980–1013

[Wik07] Wikipedia (Hrsg.): *Wikipedia - The Free Encyclopedia.* `http://en.wikipedia.org/`. Wikipedia, 2007. – Stand: Dezember 2007

[Wil03] Willink, Edward D.: UMLX: A graphical transformation language for MDA. In: *MDAFA'03.* Entschede, Netherlands, September 2003, S. 13–24

[WR99] Wile, David S. ; Ramming, J. C.: Guest Editorial: Introduction to the Special Section "Domain-Specfic Languages (DSL)". In: *IEEE Transactions on Software Engineering* 25 (1999), Mai/Juni, Nr. 3, S. 289–290

[WSKK07] Wimmer, Manuel ; Strommer, Michael ; Kargl, Horst ; Kramler, Gerhard: Towards Model Transformation Generation By-Example. In: *Proc. of 40th Hawaii International Conference on System Sciences (HICSS'07), Hawaii, USA.* Los Alamitos, CA, USA : IEEE Computer Society, Januar 2007 (System Sciences, 2007. HICSS 2007.), S. 285

[XLH+07] Xiong, Yingfei ; Liu, Dongxi ; Hu, Zhenjiang ; Zhao, Haiyan ; Takeichi, Masato ; Mei, Hong: Towards automatic model synchronization from model transformations. In: *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* New York, NY, USA : ACM, 2007, S. 164–173

[Zün01] Zündorf, Albert: *Rigorous Object Oriented Software Development.* University of Paderborn, 2001. – Habilitation

Anhang A

Beispielspezifikationen

An dieser Stelle sind einige ausgewählte Beispielspezifikationen aufgeführt. Diese Spezifikationen wurden bereits in Ausschnitten in den vorangegangenen Kapiteln zur Illustration der TGGs verwendet. Hier sind die Beispiele vollständig aufgeführt.

A.1 Block- und Klassendiagramme

Die zur Spezifikation der Korrespondezregeln benötigten Metamodelle sind in den Abbildungen A.1-A.4 abgebildet. Die zur Modellsynchronisation benötigten Korrespondenzregeln werden in den Abbildungen A.5-A.11 dargestellt.

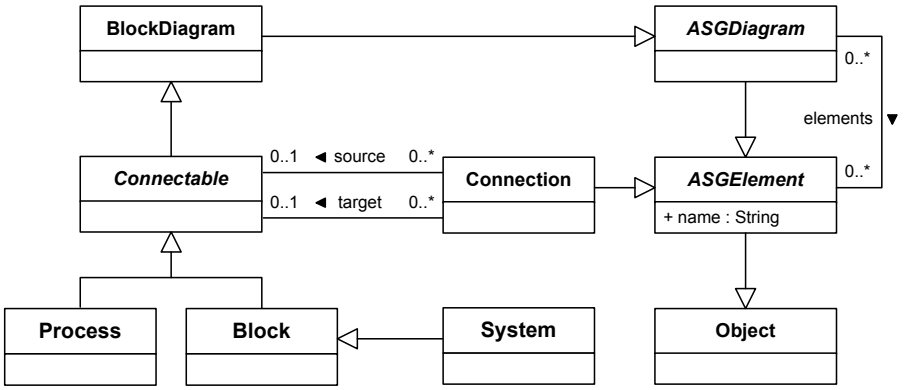


Abbildung A.1: Metamodell für Blockdiagramme

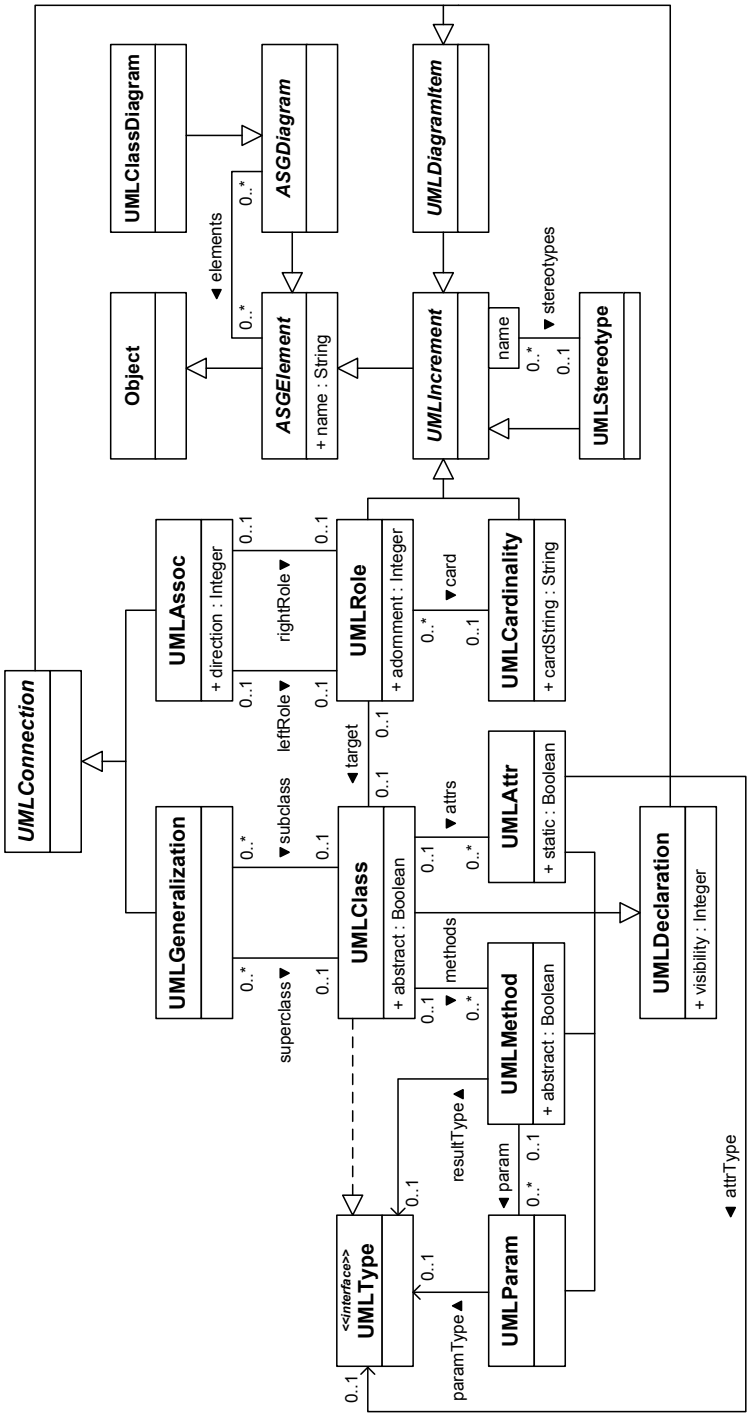


Abbildung A.2: Metamodell für Klassendiagramme

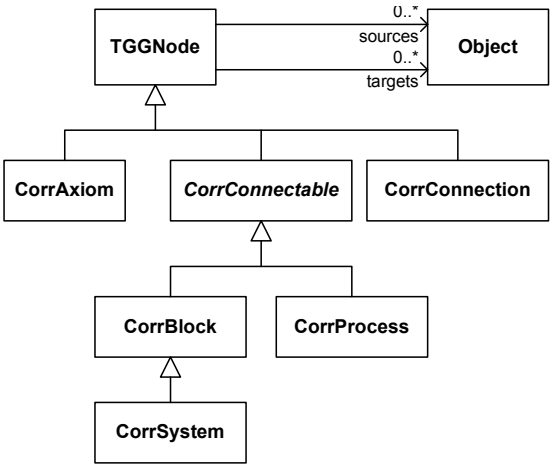


Abbildung A.3: Technisches Korrespondenzmetamodell

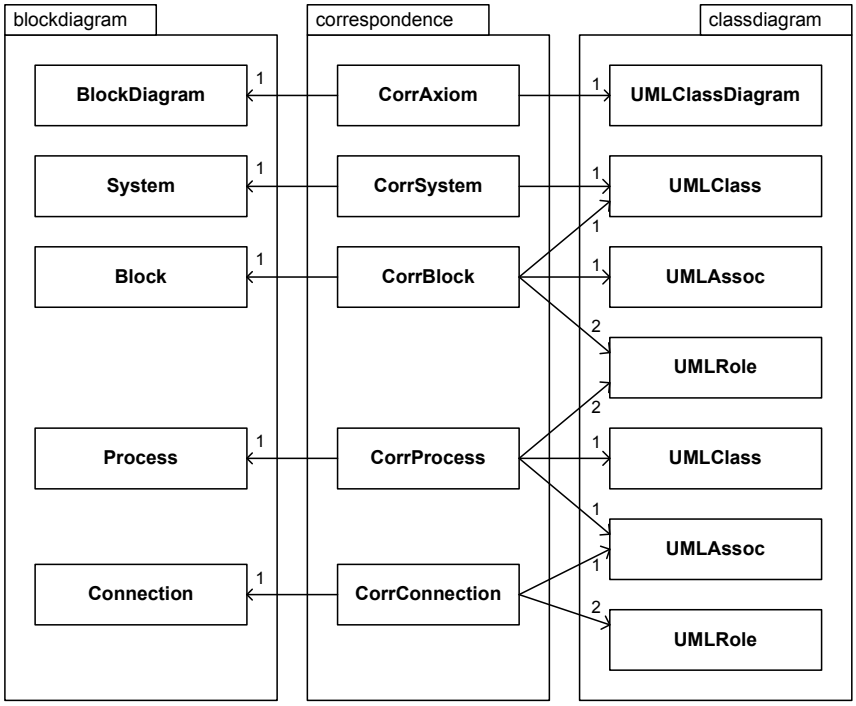


Abbildung A.4: Konzeptionelles Korrespondenzmetamodell

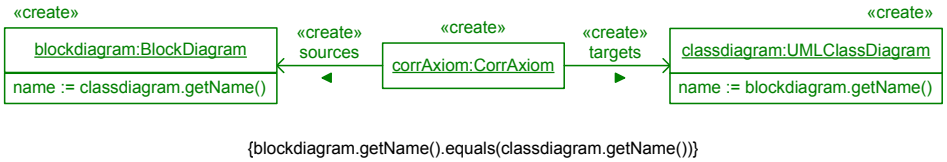


Abbildung A.5: TGG-Axiom: BlockdiagramToClassdiagram

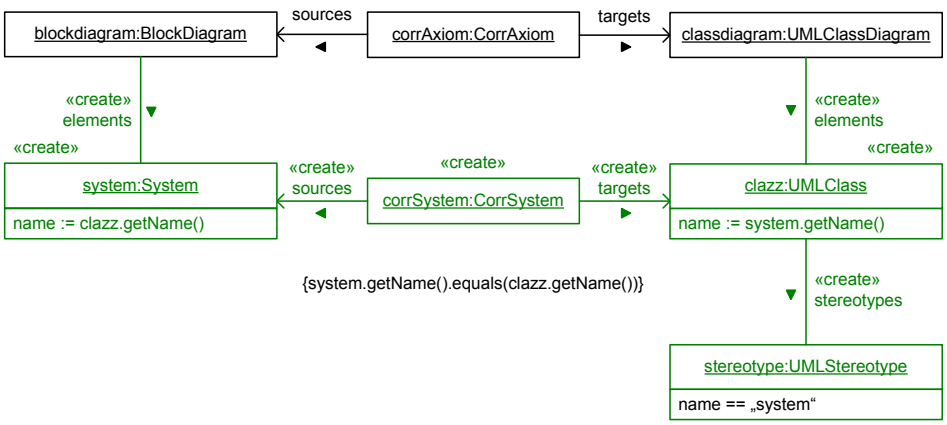


Abbildung A.6: TGG-Regel: SystemToClass

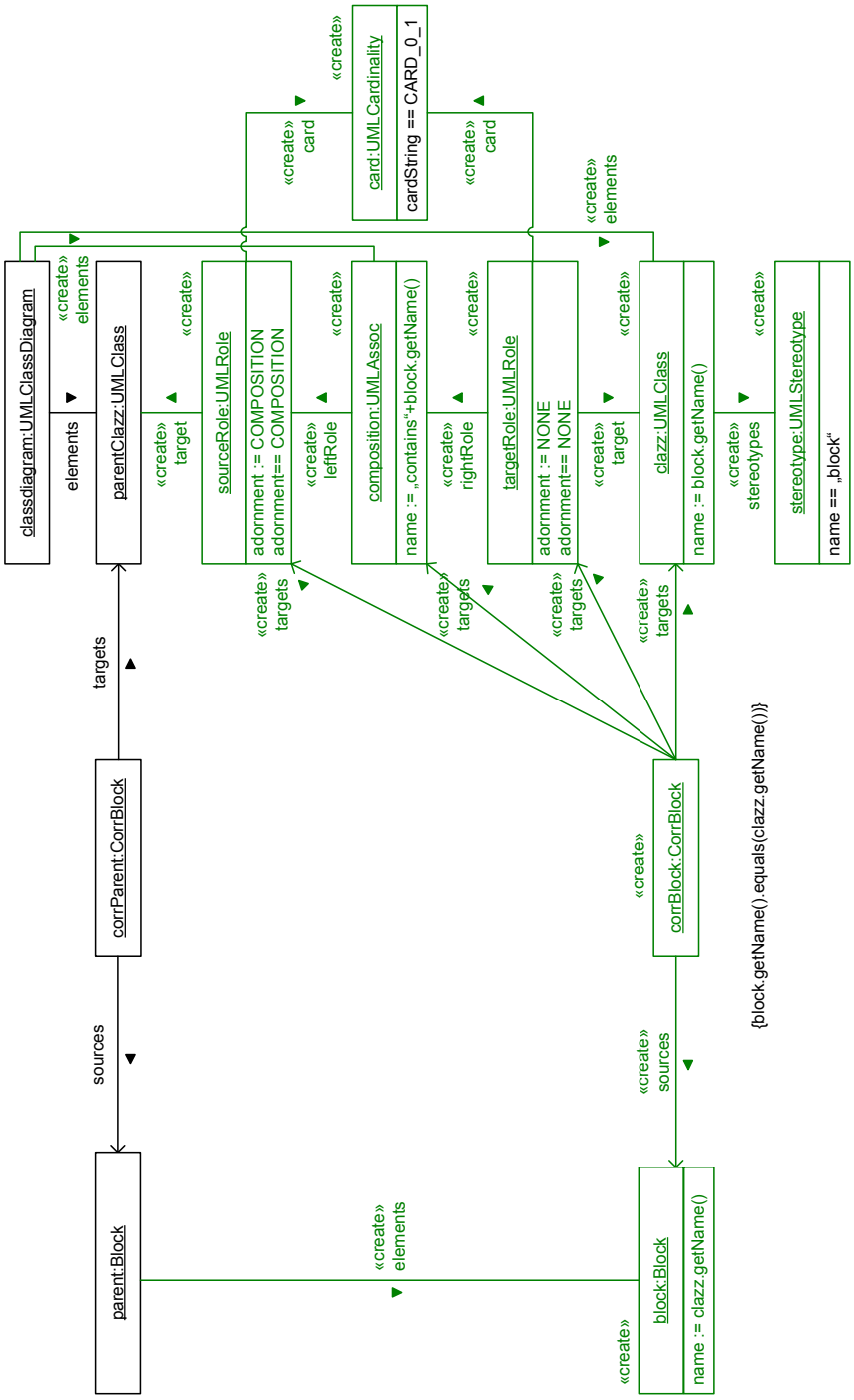


Abbildung A.7: TGG-Regel: BlockToClass

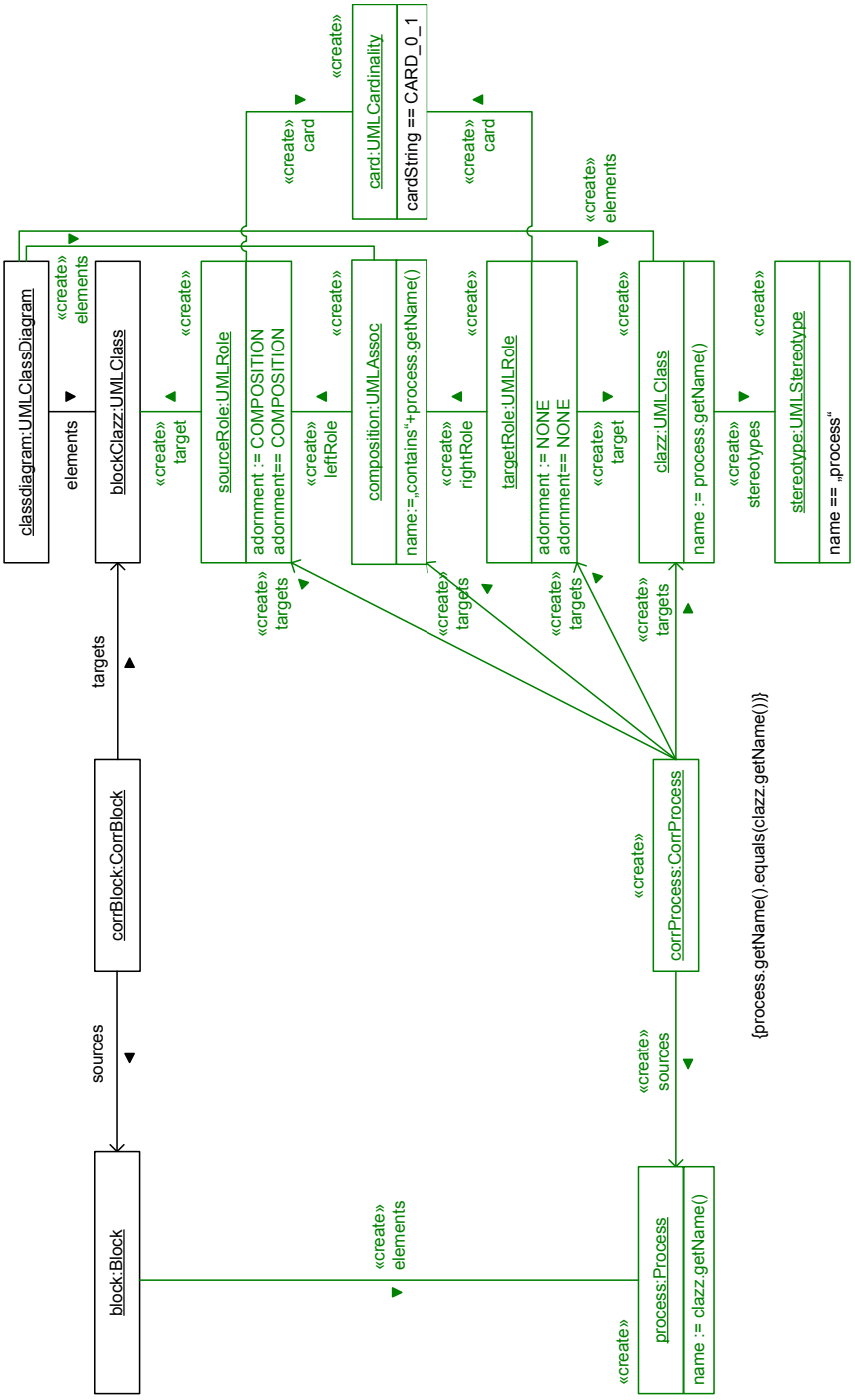


Abbildung A.8: TGG-Regel: ProcessToClass

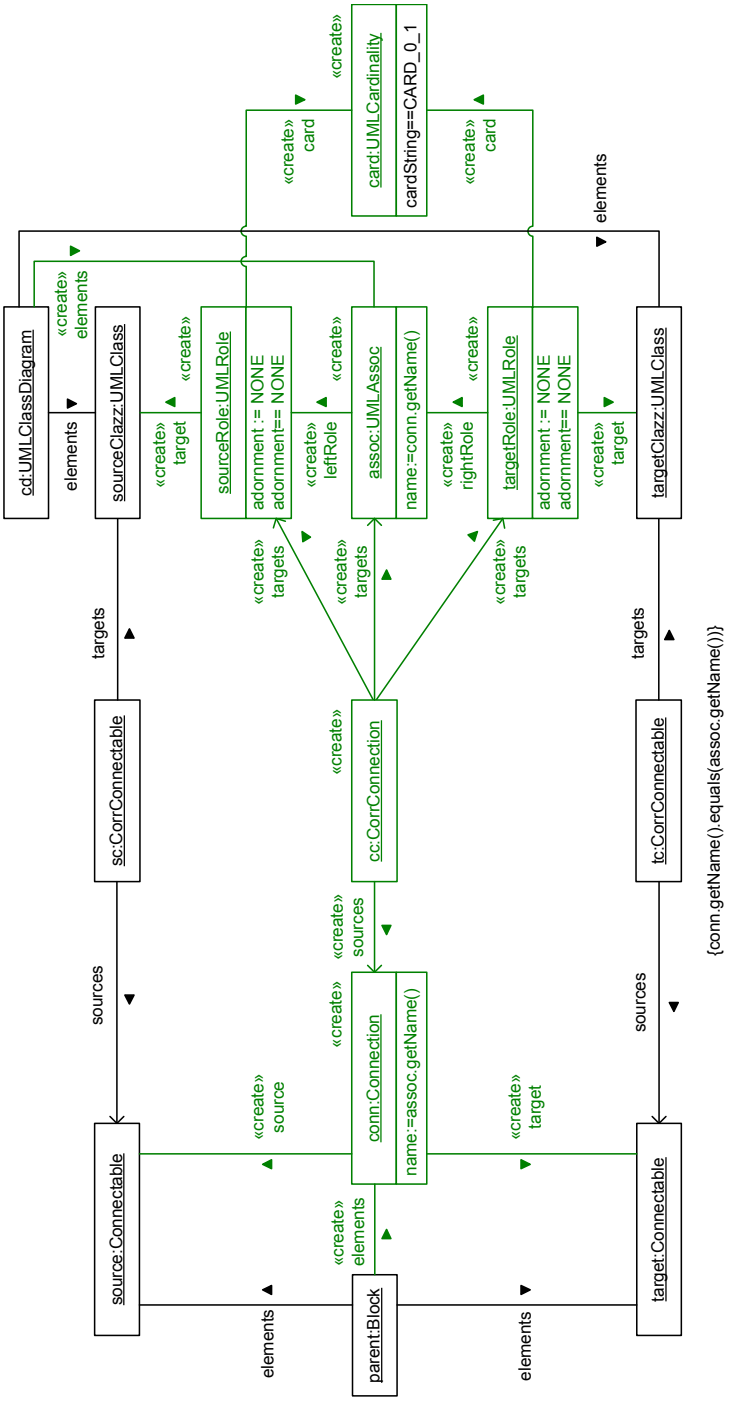


Abbildung A.9: TGG-Regel: ConnectionToAssoc (Variante 1)

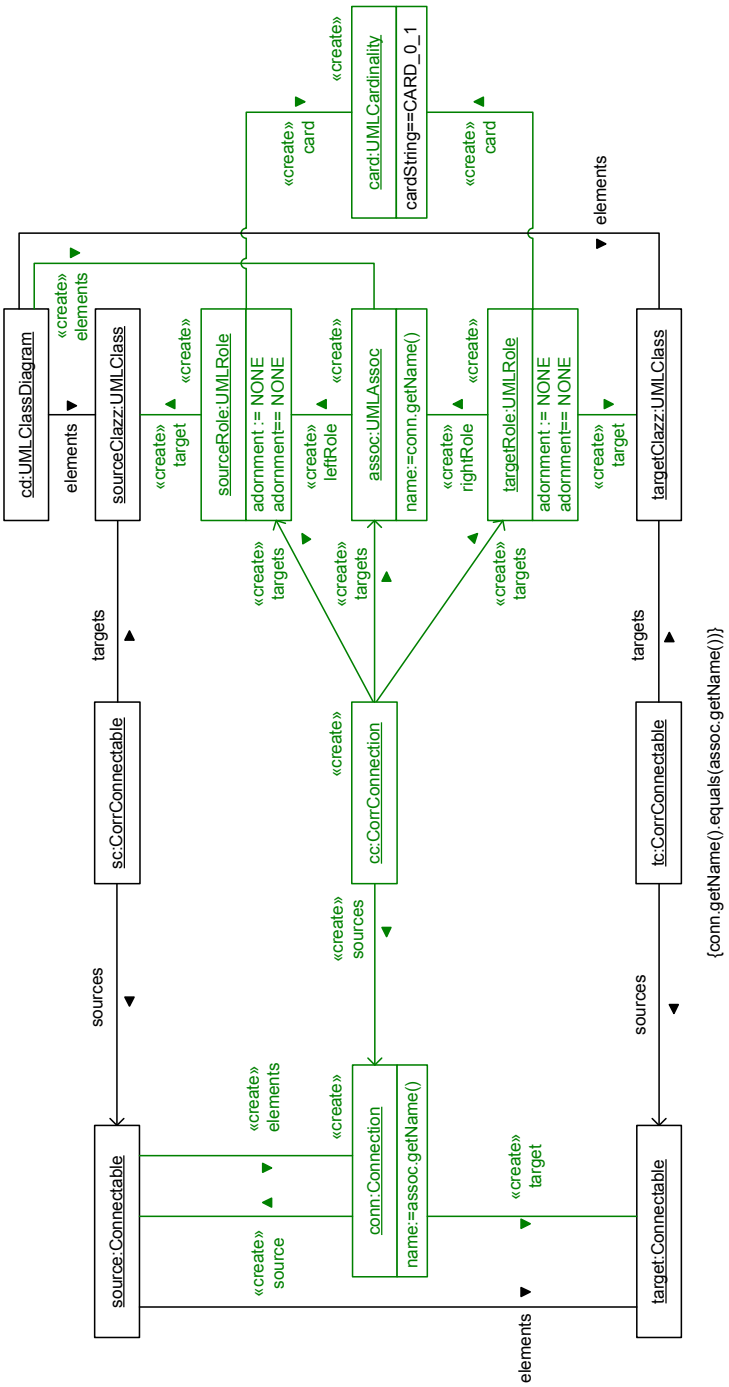


Abbildung A.10: TGG-Regel: ConnectionToAssoc (Variante 2)

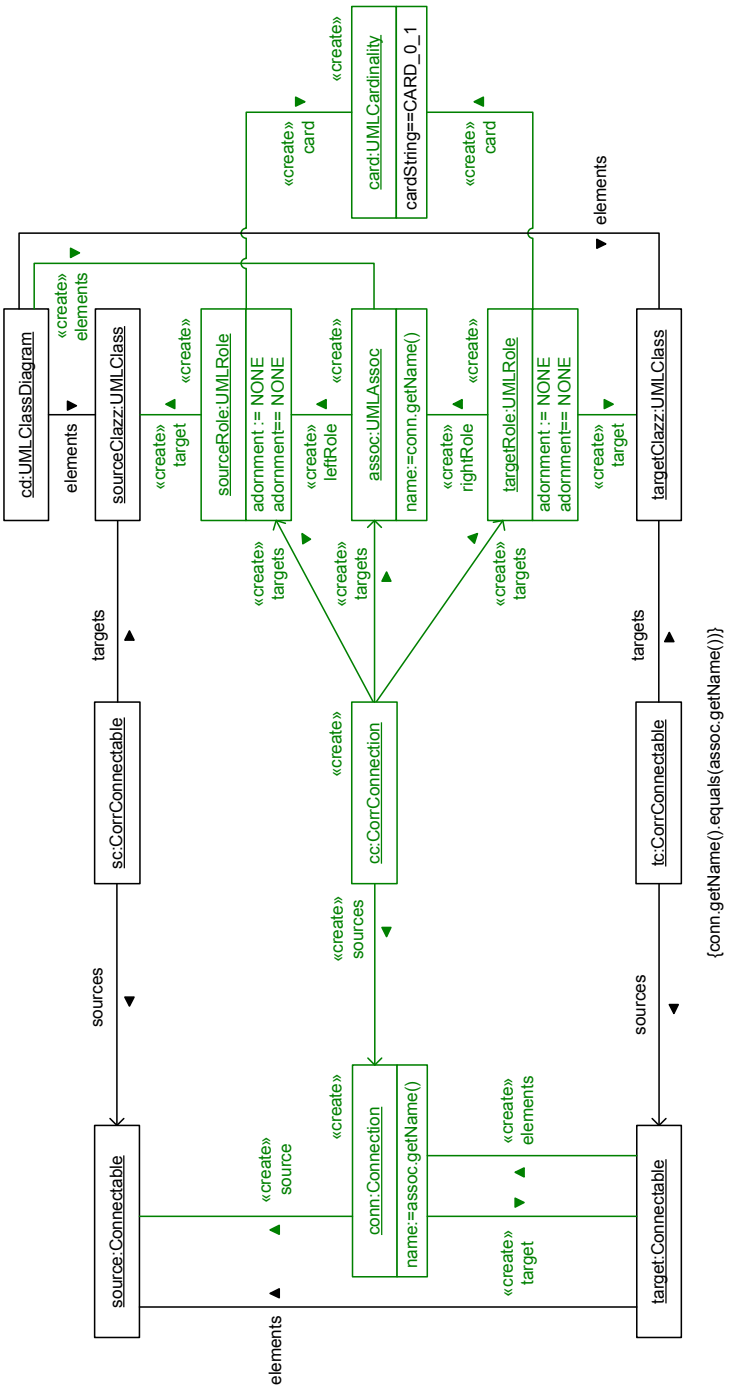


Abbildung A.11: TGG-Regel: ConnectionToAssoc (Variante 3)

A.2 I/O-Atomaten und SPS-Code

Nachfolgend ist die vollständige Spezifikation der Metamodelle und Korrespondenzregeln zur SPS-Codegenerierung aus I/O-Automaten aufgeführt. Die Metamodelle sind in den Abbildungen A.12-A.14 zu sehen, die spezifizierten TGG-Regeln hingegen in den Abbildungen A.15-A.18.

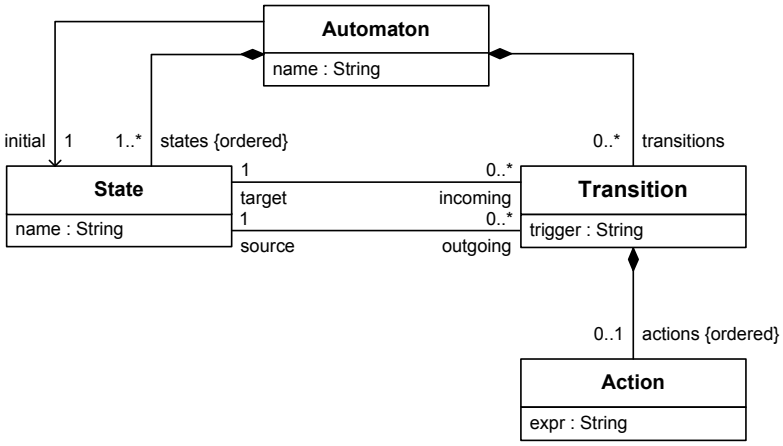


Abbildung A.12: Metamodell für Automaten

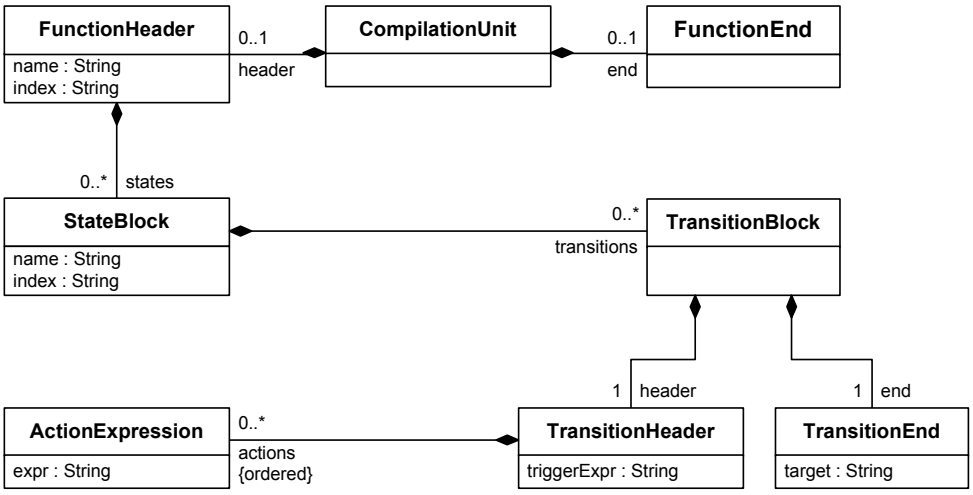


Abbildung A.13: Metamodell für SPS-Code

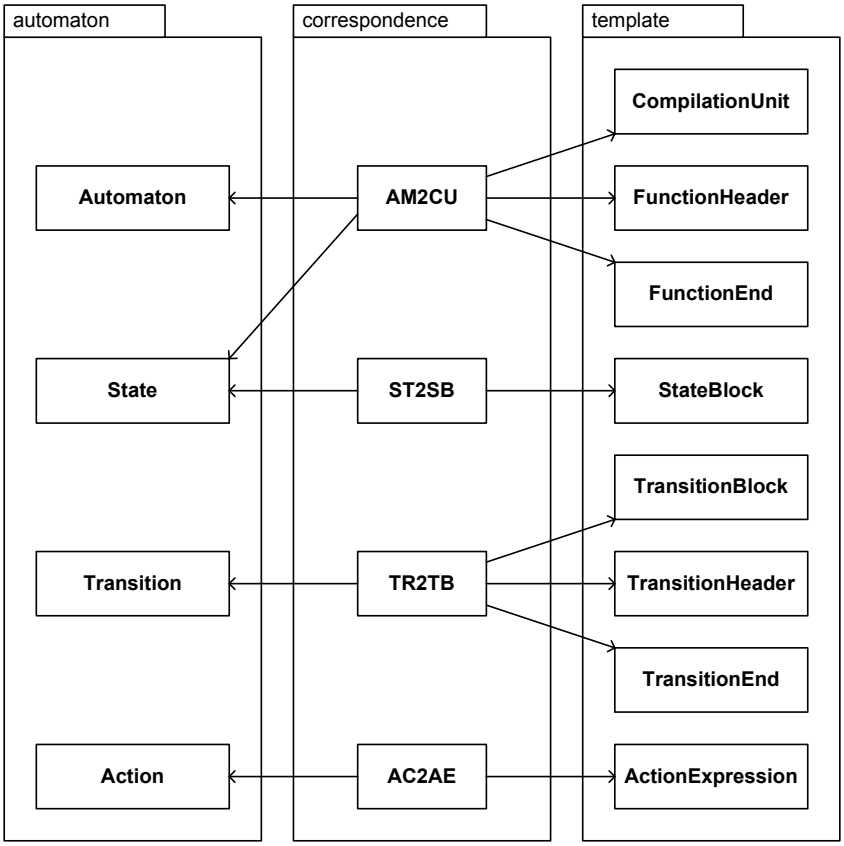


Abbildung A.14: Korrespondenzmetamodell

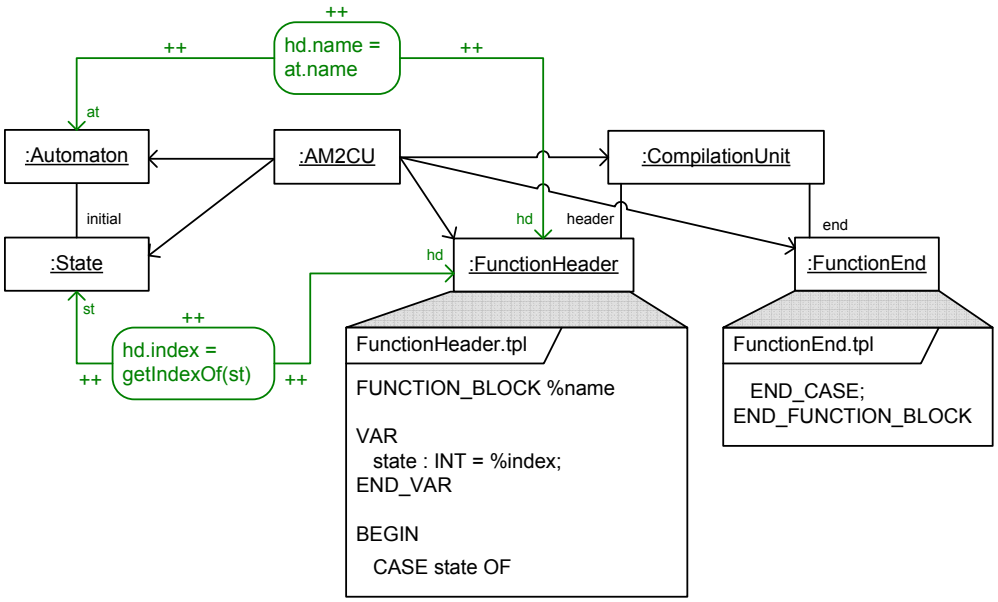


Abbildung A.15: TGG-Axiom: Automaton2CompilationUnit

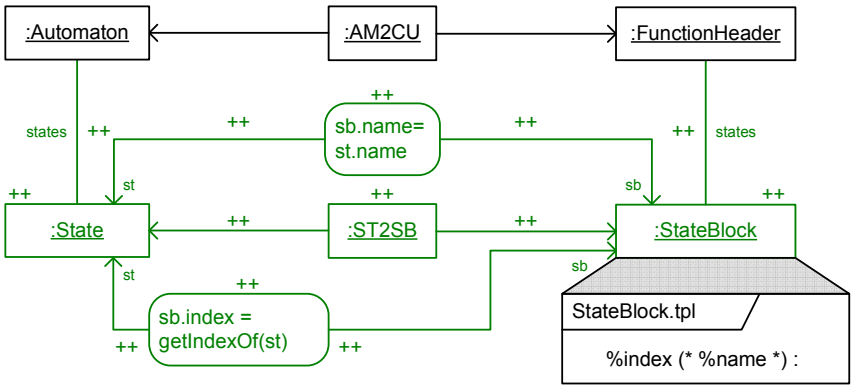


Abbildung A.16: TGG-Regel: State2StateBlock

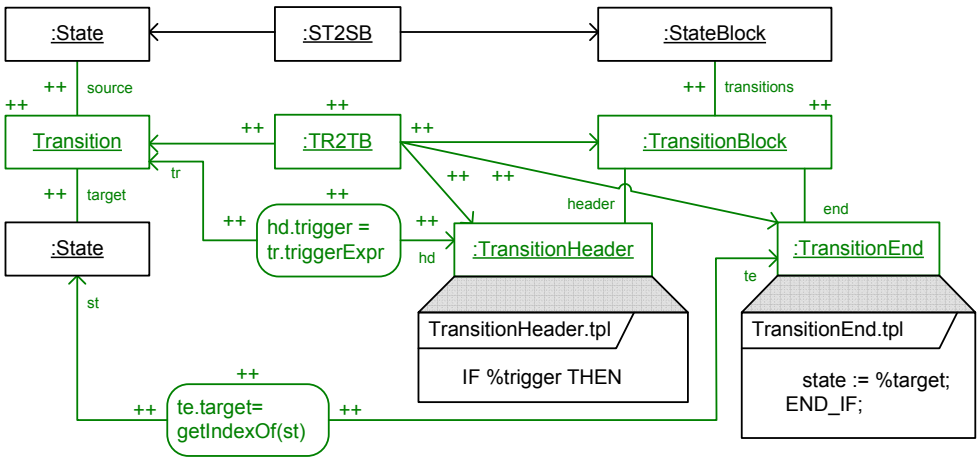


Abbildung A.17: TGG-Regel: Transition2TransitionBlock

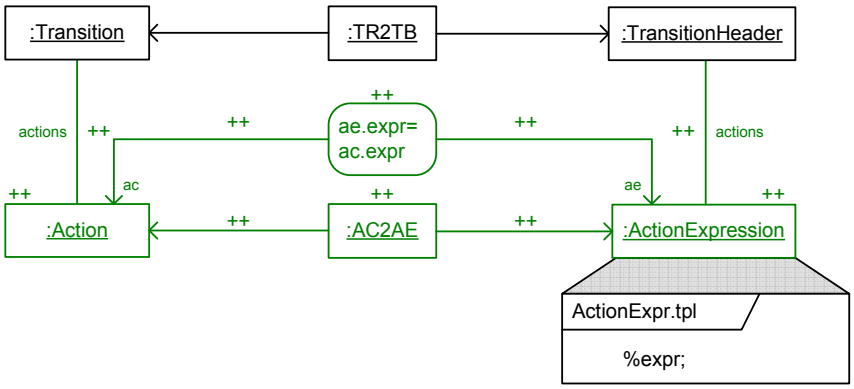


Abbildung A.18: TGG-Regel: Action2ActionBlock

Anhang B

Document Type Definition der Konfigurationsdatei

Im Folgenden ist das Datenformat der im Abschnitt 7.2.2 beschriebenen Konfigurationsdatei zu sehen.

```
<!ELEMENT configuration (triggertable,dependencies?)>
<!ELEMENT triggertable  (entry)*>
<!ELEMENT dependencies  (plugin|lib)*>

<!ELEMENT entry EMPTY>
<!ATTLIST entry trigger CDATA #REQUIRED
                rule    CDATA #REQUIRED>

<!ELEMENT plugin EMPTY >
<!ATTLIST plugin id CDATA #REQUIRED>

<!ELEMENT lib EMPTY>
<!ATTLIST lib path CDATA #REQUIRED>
```

Eine Konfigurationsdatei besteht aus einer Tabelle (`triggertable`) und einer optionalen Auflistung zusätzlich benötigter Plug-ins und Bibliotheken (`dependencies`). Ein Tabelleneintrag (`entry`) setzt sich aus den Attributen `trigger` und `rule` zusammen. Das Attribut `trigger` enthält einen vollqualifizierten Klassennamen, der den Typ eines Korrespondenzobjekts identifiziert, bei dem die Überprüfung einer TGG-Regel ausgelöst wird. Die TGG-Regel selbst wird durch das Attribut `rule` festgelegt. Die Auflistung von Plug-Ins erfolgt durch die Angabe eines Plug-in-Identifikators (Attribut `id`). Die Auflistung der zusätzlich benötigten Bibliotheken erfolgt hingegen durch die Angabe eines Dateipfades (Attribut `path`).

Abbildungen

Index

Q

R

S

T

U

V

W

Z