ModGraph: Graphtransformationen für EMF

Thomas Buchmann, Bernhard Westfechtel und Sabine Winetzhammer
Lehrstuhl für Angewandte Informatik 1 - Software Engineering
Universität Bayreuth
Universitätsstraße 30
95440 Bayreuth
thomas.buchmann@uni-bayreuth.de
bernhard.westfechtel@uni-bayreuth.de
sabine.winetzhammer@uni-bayreuth.de

Abstract: Das Eclipse Modeling Framework (EMF) ist ein weit verbreitetes Rahmenwerk zur modellgetriebenen Softwareentwicklung. Mit EMF lassen sich strukturelle Modelle als Instanzen des Ecore-Metamodells erstellen. Aus einem Modell lässt sich Code erzeugen, der im Falle benutzerdefinierter Operationen jedoch lediglich leere Methodenrümpfe enthält. Das von uns entwickelte Werkzeug ModGraph erweitert EMF um Graphtransformationsregeln zur Verhaltensmodellierung. Eine im Ecore-Modell definierte Operation kann mit Hilfe einer Graphtransformationsregel spezifiziert werden, aus der ausführbarer Code erzeugt wird. Mit Hilfe von ModGraph lassen sich komplexe Operationen auf einer hohen Abstraktionsebene spezifizieren.

1 Einleitung

Ein wichtiges Ziel der modellgetriebenen Softwareentwicklung besteht darin, die Effizienz des Entwicklungsprozesses zu steigern, indem ausführbare Modelle auf einer möglichst hohen Abstraktionsebene erstellt werden. In vielen Fällen sind Modelle jedoch allenfalls partiell ausführbar. Dies gilt beispielsweise für gängige Werkzeuge zur objektorientierten Modellierung, die aus Klassendiagrammen Code erzeugen. Durch Klassendiagramme wird lediglich die Struktur eines Softwaresystems modelliert, das Verhalten wird dagegen nicht beschrieben.

Diese Beschränkung trifft insbesondere auf das weit verbreitete Eclipse Modeling Framework (EMF [SBPM09]) zu. Mit Hilfe von EMF lassen sich strukturelle Klassenmodelle als Instanzen des Ecore-Metamodells erstellen. Der EMF-Codegenerator erzeugt Code für Klassen, Attribute und Referenzen. Ferner erzeugt er ausführbaren Code für Elementaroperationen, z.B. zum Setzen von Attributwerten bzw. zum Erzeugen und Löschen von Links (Instanzen von Referenzen). Für benutzerdefinierte Operationen wird jedoch lediglich ein Implementierungsrahmen (eine Methode mit leerem Rumpf) erzeugt, der vom Programmierer ausgefüllt werden muss.

An dieser Stelle setzt das von uns entwickelte Werkzeug ModGraph [BWW11] an. ModGraph basiert auf Graphtransformationsregeln, mit deren Hilfe sich komplexe Operationen auf EMF-Modellinstanzen beschreiben lassen. Dabei wird eine EMF-Modellinstanz als

Graph aufgefasst, dessen Knoten und Kanten den Objekten und Links entsprechen, die aus den Klassen und Referenzen des zugrunde liegenden Ecore-Modells instanziiert werden (d.h. der Graph repräsentiert die abstrakte Syntax der Modellinstanz). Eine benutzerdefinierte Operation des Ecore-Modells lässt sich mit Hilfe einer Graphtransformationsregel spezifizieren, aus der ausführbarer Code erzeugt wird, der den vom EMF-Codegenerator erzeugten Code erweitert. Damit trägt ModGraph dazu bei, die vom EMF-Rahmenwerk offen gelassene Lücke in der Verhaltensmodellierung zu schließen.

Während wir in [BWW11] einen ersten Überblick über einen frühen Entwicklungsstand von ModGraph vermittelt haben, basiert der vorliegende Aufsatz auf einem vollständig implementierten Werkzeug (grafischer Editor, Validierung und Codegenerierung). Der Schwerpunkt liegt auf der umfassenden Beschreibung der Sprachkonstrukte von Graphtransformationsregeln (Abschnitt 2). Der anschließende Abschnitt 3 geht auf die Implementierung ein. Verwandte Arbeiten werden in Abschnitt 4 diskutiert. Abschnitt 5 rundet den Aufsatz mit einer kurzen Zusammenfassung und einem Ausblick auf zukünftige Arbeiten ab.

2 Graphtransformationsregeln

Eine ModGraph-Regel modelliert den Rumpf einer benutzerdefinierten Operation des Ecore-Modells. Dabei wird jede EMF-Modellinstanz als Graph aufgefasst und Änderungen an diesen Instanzen, erzeugt durch Operationsaufrufe, als Transformation des Graphen beschrieben.
Im Folgenden besprechen wir zunächst die in ModGraph verfügbaren Sprachkonstrukte anhand des Metamodells für Graphtransformationsregeln. Dabei handelt es sich um ein Metamodell, das die spezifischen Anforderungen der Werkzeugunterstützung (basierend auf GMF) berücksichtigt. Anschließend wird die Anwendung der Sprachkonstrukte anhand von Beispielen illustriert.

2.1 Sprachkonstrukte

Abbildung 1 zeigt die Hauptkomponenten einer hierarchisch aufgebauten ModGraph-Regel. Eine ModGraph-Regel referenziert die benutzerdefinierte Operation im Ecore-Modell, deren Rumpf sie implementiert. Jeder Regel wird eine Ausnahme (`GTFailure`) zugeordnet, die im Falle des Scheiterns der Regel ausgelöst wird. Zudem besteht die Möglichkeit der Erstellung eines Kommentars (`GTComment`).

Die Regel selbst besteht primär aus einem Graphmuster (`GTGraphPattern`). Dieses stellt sowohl den zu suchenden Teilgraphen innerhalb der EMF-Instanz dar, als auch die anzuwendenden Änderungen. Das Graphmuster kann durch Anwendbarkeitsbedingungen ergänzt werden. Dazu stehen hier Vor- und Nachbedingungen (`GTOCLPrecondition` und `GTOCLPostcondition`), die in OCL formuliert werden, zur Verfügung. Außerdem kann das Graphmuster mit einer negativen Anwendbarkeitsbedingung (`GTNegative-ApplicationCondition`) versehen werden. Die negative Anwendbarkeitsbedingung

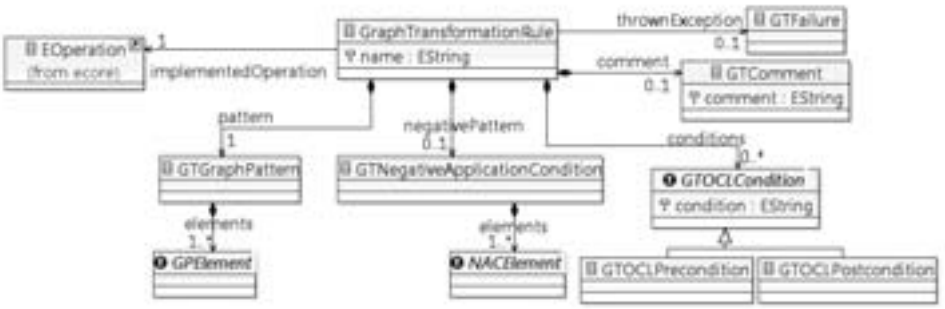

Abbildung 1: Hauptkomponenten einer ModGraph-Regel. Ausschnitt aus dem ModGraph-Metamodell für Graphtransformationen.

wird ebenfalls durch einen Graphen repräsentiert.

Das Metamodell des primären Graphmusters ist in Abbildung 2 dargestellt. Dieses steuert den Ablauf der Transformation durch die dargestellten Elemente. Es ähnelt einem UML-Kommunikationsdiagramm, welches auch aus einem statischen Teil - dem zugrundeliegenden Objektdiagramm - und einem dynamischen Teil - den Änderungsoperationen - besteht.

Konkret werden im Graphmuster Elemente mit und ohne Status unterschieden. Der Status (`GTStatus`) gibt dabei an, ob ein Element erhalten, gelöscht oder erzeugt wird. Allen Elementen, mit Ausnahme des this-Objekts und der Pfade, ist ein Status zugeordnet.

Das Graphmuster besteht aus mindestens einem gebundenen Knoten (`GPBoundNode`). Dieser Knoten ist entweder das this-Objekt (`GTThisObject`) oder ein Übergabeparameter der Methode, der ein Objekt repräsentiert. Die Übergabeparameter werden, je nach Wertigkeit in mehrwertige Multiparameter (`GPMultiParameter`) und einwertige Parameter (`GPParameter`) unterteilt. Primitive Parameter der Methode können für Attributbedingungen und Attributzuweisungen verwendet werden.

Zudem kann ein Graphmuster eine beliebige Anzahl von ungebundenen Knoten (`GPUnboundNode`) enthalten. Ungebundene Knoten werden durch die Mustersuche gebunden. Sie sind, analog zu den Parametern, in mehrwertige Multiobjekte (`GPMultiObject`) und einwertige Objekte (`GPObject`) unterteilt. Ungebundene Knoten müssen auf Klassen im Ecore-Modell referenzieren, während dies für gebundene Knoten nicht zwingend notwendig ist.

Jeder dieser Knoten kann als Rückgabewert der Methode (`GPNode.outParameter`) oder als optional (`GPNode.optional`) gekennzeichnet werden. Bei Parametern und Multiparametern kann eine Typumwandlung stattfinden.

Alle Knoten im Graphmuster können durch zwei Arten von Kanten verbunden werden, entweder durch Links oder durch Pfade.

Pfade (`GTPath`) stehen für abgeleitete Referenzen. Sie sind mit einem Pfadausdruck

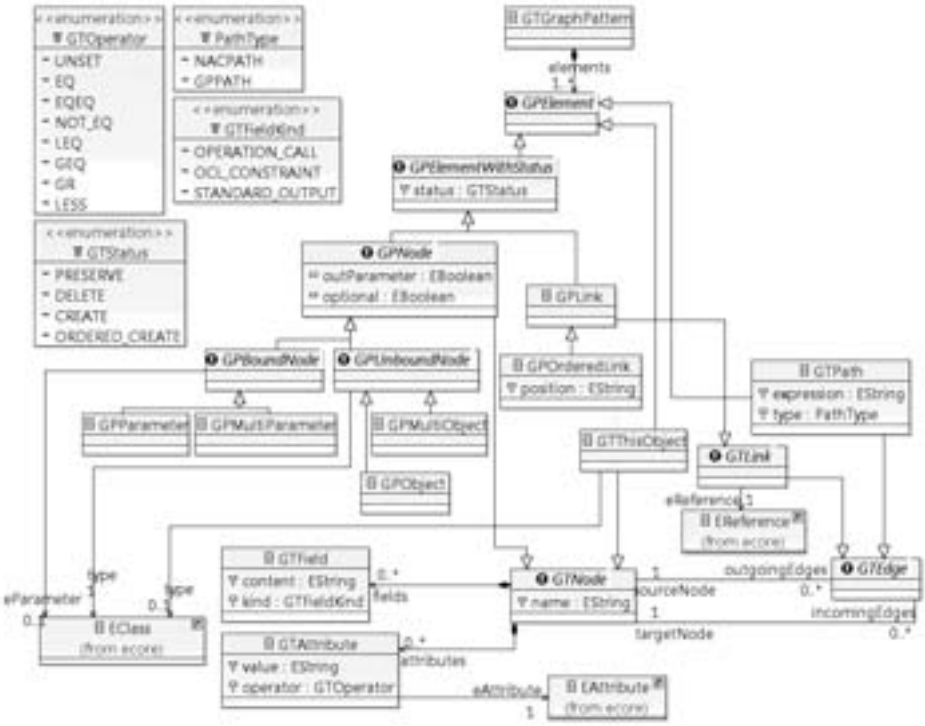

Abbildung 2: Aufbau des primären Graphmusters. Ausschnitt aus dem ModGraph-Metamodell für Graphtransformationen.

markiert, der auf dem Quellobjekt ausgewertet wird, und eine Menge von Zielobjekten zurückliefert. Ein Pfadausdruck ist entweder in OCL oder Java formuliert.

Links (`GPLink`) stellen Instanzen von Referenzen im Ecore-Modell dar und referenzieren diese. Jedem Link ist ein Status (erhalten, erzeugen oder löschen) zugeordnet. Ein zu erzeugender Link, der eine mehrwertige Referenz instanziiert, kann geordnet sein (`GPOrderedLink`). Bei geordneten Links wird das Einfügen des Zielobjekts an einer bestimmten Position, die durch einen Ausdruck am Link vorgegeben ist, erreicht. Links, die in optionalen Knoten enden, werden als optional angenommen.

Innerhalb der Knoten im Graphmuster stehen, je nach Status, verschiedene Möglichkeiten zur Wahl: An einen zu erhaltenden oder zu löschenden Knoten können Bedingungen gestellt werden. Diese Bedingungen müssen vom Knoten erfüllt werden. Sie werden entweder, mit Hilfe eines Feldes innerhalb des Knotens (`GTField`) als OCL-Bedingung formuliert, oder als Java-Bedingung an ein Attribut des Knotens (`GTAttribute`). An einem zu erhaltenden oder zu erstellenden Knoten können Änderungen an Attributen vorgenommen werden. Ein Attributwert kann durch einen OCL- oder Java-Ausdruck spezifiziert werden. Zudem besteht die Möglichkeit, Methoden auf dem Knoten aufzurufen.

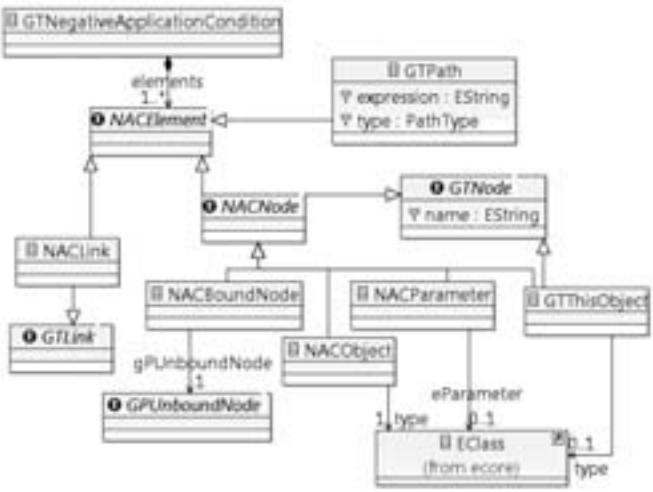

Abbildung 3: Aufbau der negativen Anwendbarkeitsbedingung. Ausschnitt aus dem ModGraph-Metamodell für Graphtransformationen.

Die negative Anwendbarkeitsbedingung, deren Metamodell in Abbildung 3 dargestellt ist, wird nach der Mustersuche des primären Graphmusters ausgewertet. Zu diesem Zeitpunkt ist die EMF-Modellinstanz unmodifiziert. Die negative Anwendbarkeitsbedingung beinhaltet ein Graphmuster, das keinesfalls auftreten darf, sobald die Regel angewendet wird.

Dieses Graphmuster beinhaltet ebenfalls mindestens einen gebundenen Knoten (`NACBoundNode`). Gebundene Knoten sind hier, wie im Graphmuster des Graph Patterns, das this-Objekt (`GTThisObject`) und die Übergabeparameter der Methode, die ein Objekt repräsentieren (`NACParameter`). Hinzu kommen die gebundenen Knoten der negativen Anwendbarkeitsbedingung (`NACBoundNode`). Diese stehen für Knoten, die bereits im primären Graphmuster als ungebundene Knoten vorkommen, und referenzieren diese gleichzeitig. Dies ist möglich, da besagte Knoten zum Zeitpunkt des Prüfens der negativen Anwendbarkeitsbedingung bereits gebunden sind. Ungebundene einfache Objekte sind innerhalb der negativen Anwendbarkeitsbedingung ebenfalls zulässig (`NACObject`).

Im Gegensatz zum Graphmuster im Graph Pattern gibt es in der negativen Anwendbarkeitsbedingung keinen Status für die Elemente. Knoten können hier nicht verändert werden, aber Bedingungen dürfen an sie gestellt werden.

Alle Knoten innerhalb einer negativen Anwendbarkeitsbedingung können durch Links (`NACLink`) und Pfade (`GTPath`) verbunden werden.

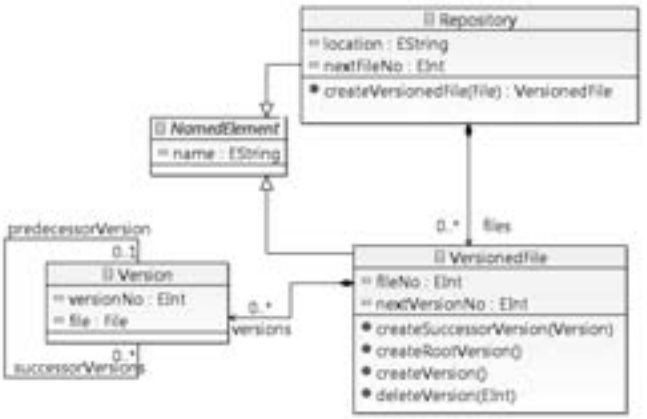

Abbildung 4: Beispiel-Ecore-Modell eines einfachen Versionsverwaltungssystems.

2.2 Beispiel

Um die verfügbaren Sprachkonstrukte näher zu erläutern, betrachten wir das Beispiel eines einfachen Versionsverwaltungssystems. Das Beispiel wurde so konstruiert, dass bei minimalem Umfang viele Sprachkonstrukte abgedeckt werden. Das Ecore-Klassendiagramm ist in Abbildung 4 dargestellt. Das hier betrachtete Versionsverwaltungssystem besteht aus einem Repository, repräsentiert durch die gleichnamige Klasse, das lediglich Dateien versionieren kann. Diese Dateien werden durch die Klasse `VersionedFile` modelliert und besitzen je eine eindeutige, durch das Repository vergebene, Nummer. Zudem hat das Repository, sowie jede versionierte Datei, einen Namen, den diese von der abstrakten Klasse `NamedElement` erben. Jede der versionierten Dateien ist in verschiedenen Versionen, modelliert durch die Klasse `Version`, verfügbar. Jede Version hat eine eindeutige Versionsnummer. Die Versionsnummern werden von der Klasse `VersionedFile` verwaltet. Zudem besitzt jede Version maximal eine Vorgängerversion und beliebig viele Nachfolgerversionen, sowie einen ihr zugeordneten Dateiinhalt. Dieser Dateiinhalt ist durch das Attribut `file` des Datentyps `File` dargestellt.

Die erste hier betrachtete Regel implementiert die Operation `createVersionedFile` - die eine initiale Commit-Operation einer Datei modelliert - und ist in Abbildung 5 dargestellt. Die implementierte Methode und deren Klasse sind in der Eclipse-Properties-View sichtbar (nicht abgebildet), weswegen kein Aufrufpattern innerhalb der Regel existiert.

Die Vorbedingung oben im Bild ist als OCL-Bedingung formuliert. Sie bestimmt die Anwendbarkeit der Regel, indem sie in diesem Fall prüft ob die übergebene Datei existiert.

Das Graph Pattern enthält ein this-Objekt, welches das aktuelle Repository repräsentiert. Wie jedes zu erhaltende Objekt, enthält das this-Objekt drei Bereiche. Der erste Bereich zeigt den Bezeichner des Objekts, der zweite - hier leere Bereich - enthält Bedingungen an das Objekt und der dritte Bereich alle Änderungen an dem Objekt. Die Änderung hier

Abbildung 5: Graphtransformationsregel zum Einfügen einer versionierten Datei in ein Repository.

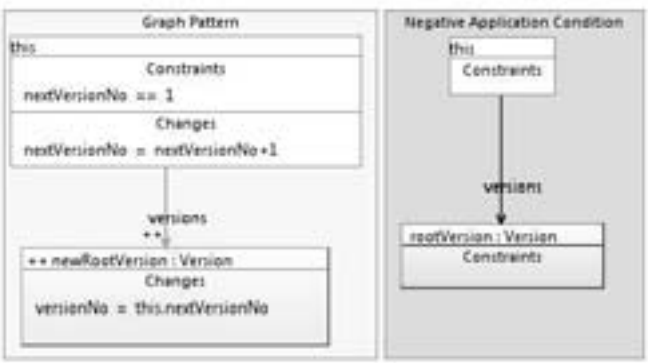

Abbildung 6: Graphtransformationsregel zur Erzeugung der Basisversion einer versionierten Datei in einem Repository.

ist, den Zähler für die aktuellen Dateinummern um eins zu erhöhen. Das zweite Objekt im Graph Pattern, `versionedFile`, vom gleichnamigen Typ, ist ein neu zu erzeugendes Objekt (++), die neu erzeugte versionierte Datei. Dieses Objekt enthält nur zwei Bereiche. Der obere Bereich, in dem primär der Bezeichner und die Klasse des Objekts dargestellt sind, enthält hier zusätzlich eine Markierung mit `<<out>>`, die das Objekt als Rückgabeparameter der Methode kennzeichnet. Im unteren Bereich können Änderungen am neu erzeugten Objekt vorgenommen werden. Zuerst wird mit Hilfe eines direkten Attributzugriffs die Dateinummer gleich der `nextFileNo` des this-Objekts gesetzt, dann über einen OCL-Ausdruck (gekennzeichnet durch []) der Name der versionierten gleich

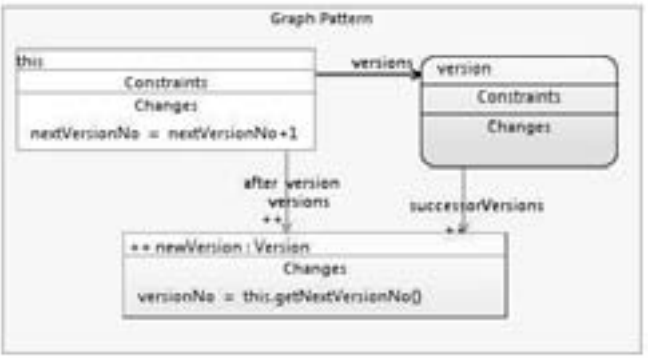

Abbildung 7: Graphtransformationsregel zur Erzeugung einer Folgeversion einer versionierten Datei in einem Repository.

dem der unversionierten Datei gesetzt. Zuletzt wird die Methode `createRootVersion` auf dem neu erzeugten Objekt aufgerufen.

Die ModGraph-Regel zu dieser Methode ist in Abbildung 6 dargestellt. Sie besteht aus einem Graph Pattern und einer negativen Anwendbarkeitsbedingung. Im Graph Pattern befindet sich ein this-Objekt, das hier für eine Instanz der Klasse `VersionedFile` steht. Die Bedingung an das this-Objekt, dass die nächste zu vergebende Versionsnummer gleich eins ist, ist als Attributbedingung dargestellt. Analog zur Dateinummer der Regel aus Abbildung 5, wird hier eine eindeutige Versionsnummer vom this-Objekt verwaltet und an die Versionen zugewiesen. Der zu erzeugende Link ordnet die neue Basisversion der versionierten Datei zu. Die negative Anwendbarkeitsbedingung (links in Abbildung 6) verbietet die Existenz einer Version zur versionierten Datei. Die versionierte Datei ist wiederum durch das this-Objekt repräsentiert. Die Version der Datei ist ein ungebundenes Objekt der negativen Anwendbarkeitsbedingung.

Abbildung 7 zeigt eine ModGraph-Regel zur Erzeugung einer Folgeversion, der die Vorgängerversion übergeben wird. Das this-Objekt, sowie das neu zu erzeugende Objekt - die Folgeversion - sind analog zu der Regel in Abbildung 6 zu sehen. Die Versionsnummer wird hier jedoch durch einen Methodenaufruf, statt durch direkten Attributzugriff, zugewiesen. Beide Varianten können gleichwertig verwendet werden, frei nach Belieben des Modellierers.

Der Übergabeparameter der Methode, `version`, ist als Knoten mit abgerundeten Ecken dargestellt. Der Link zwischen dem this-Objekt und diesem Objekt stellt sicher, dass die übergebene Version eine Version der versionierten Datei ist.

Der Link zwischen this-Objekt und neuem Objekt ist hier ein geordneter Link. In der Liste der Versionen der versionierten Datei wird die neue Version direkt hinter die Vorgängerversion eingeordnet. (Dies ist an dieser Stelle möglich, da in Ecore-Modellen alle Referenzen implizit geordnet sind.) Das Einfügen wird durch das Schlüsselwort `after` und dem Bezeichner des Vorgängerknotens dargestellt. Der Link zwischen der Vorgängerversion und

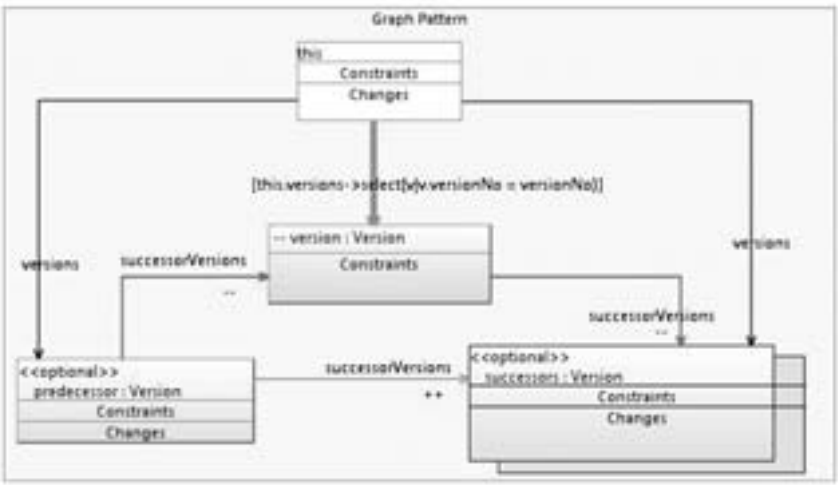

Abbildung 8: Graphtransformationsregel zum Löschen einer Version einer versionierten Datei in einem Repository.

der neuen Version wird ebenfalls erzeugt.

Das Löschen einer Version - die wie oben beschrieben erzeugt wurde - ist in Abbildung 8 dargestellt. Die hier modellierte Operation `deleteVersion` erwartet als Übergabeparameter die Versionsnummer (`versionNo`) der zu löschenden Version. Auch hier repräsentiert das this-Objekt eine Instanz der Klasse `VersionedFile`. Es ist über einen Pfadausdruck mit der zu löschenden Version verbunden, die durch ein zu löschendes Objekt mit Bezeichner `version` dargestellt ist. Deren Vorgängerversion ist durch das optionale Objekt `predecessor` dargestellt, die Nachfolgerversionen durch das ebenfalls optionale Multiobjekt `successors`. Vor- und Nachfolgerversion werden durch optionale, ungebundene Knoten dargestellt, da ihre Existenz nicht in jedem Falle gesichert ist. Wird beispielsweise die neueste Version gelöscht, gibt es keine Nachfolgerversionen. Die mit `versions` markierten Links stellen sicher, dass die Vorgängerversion und die Nachfolgerversionen tatsächlich zur selben versionierten Datei gehören. Die neuen Nachfolgerbeziehungen werden durch das Löschen der Nachfolgerbeziehungen zu `version` und das Erzeugen selbiger von `predecessor` zu `successors` erreicht.

3 Implementierung

3.1 Grafische Oberfläche

ModGraph bietet einen grafischen Editor mit Validierung und Codegenerierung. Dazu erleichtert ein Dashboard die Handhabung für den Benutzer. Abbildung 9 zeigt die aktuelle

Oberfläche. Neben dem Eclipse-Package-Explorer findet sich der GMF-basierte grafische Editor für Graphtransformationsregeln. Hier können die Regeln erstellt werden. Während der Erstellung werden durch die Live-Validierung erste Fehler vermieden. Mit einer Batch-Validierung, die auf der Sprache Check[1] basiert, kann die Regel zusätzlich überprüft werden. Die Prüfung erfolgt gegen das ModGraph-Metamodell für Graphtransformationsregeln und gegen das Ecore-Klassendiagramm, in dem die implementierte Operation modelliert ist.

Rechts davon befindet sich ein Kommentarreiter. Dieser ermöglicht die Erstellung von JavaDoc-Kommentaren zur implementierten Methode. Unten im Bild ist das ModGraph-Dashboard zu sehen. Unser Dashboard fasst die verschiedenen Entwicklungsschritte vom Ecore-Klassendiagramm über die Graphtransformationsregeln bis zum ausführbaren Modell kompakt zusammen. Nach der Auswahl oder Erstellung des Ecore-Klassendiagramms ermöglicht es zum Beispiel das Erzeugen eines zugehörigen Paketdiagramms, das mit unserem Paketdiagrammeditor [BDK09] bearbeitet werden kann. Graphtransformationsregeln können im Dashboard ganz einfach erstellt und validiert werden. Auch die Java Quelltextgenerierung ist hier mittels eines Klicks aufrufbar. Zudem können Methoden, die in Java implementiert werden, direkt ausgewählt werden. Dies ist von besonderer Bedeutung, da ModGraph bislang keine Möglichkeit zur Kontrollflussmodellierung bietet. Kontrollflüsse werden aktuell in Java geschrieben. Ein Werkzeug zur Darstellung von Kontrollflüssen ist jedoch bereits in Entwicklung und rechts unten im Dashboard vorgesehen.

3.2 Zusammenspiel von EMF und ModGraph

Einfügen in den EMF-Code. Da ModGraph für und mit EMF entworfen und gebaut wurde, wird der ModGraph-generierte Quelltext nahtlos in den EMF-generierten Quelltext eingefügt. Eine Modifikation der EMF-Templates findet nicht statt. Vielmehr verfügt ModGraph über eigene XPand-Templates [Gro09], die zur Generierung des Quelltextes verwendet werden. Dabei bedient sich ModGraph einer Compiler-Lösung. Die modellierten Regeln werden also in Quelltext übersetzt.

Wie der ModGraph-Codegenerator in den EMF-generierten Quelltext eingreift, ist schematisch in Abbildung 10 dargestellt. Es wird vorausgesetzt, dass die EMF-Codegenerierung vor der ModGraph-Codegenerierung stattfindet, sodass die EMF-generierten Pakete (in Abbildung 10 links dargestellt) erweitert werden können. Das erste Paket enthält Interfaces, das zweite die zugehörigen Implementierungsklassen. Das dritte Paket, im Folgenden Util-Paket genannt, stellt Utility-Klassen bereit.

Jeder ModGraph-Codegenerierung, wie sie in Abbildung 10 rechts dargestellt ist, geht zunächst eine Batch-Validierung voran. Danach wird der ModGraph-generierte Quelltext in den EMF-generierten Quelltext mit Hilfe des abstrakten Syntaxbaums der EMF-generierten Java-Dateien eingefügt.

Alle durch ModGraph implementierten Methoden können Ausnahmen auslösen, zum Bei-

[1] `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html`

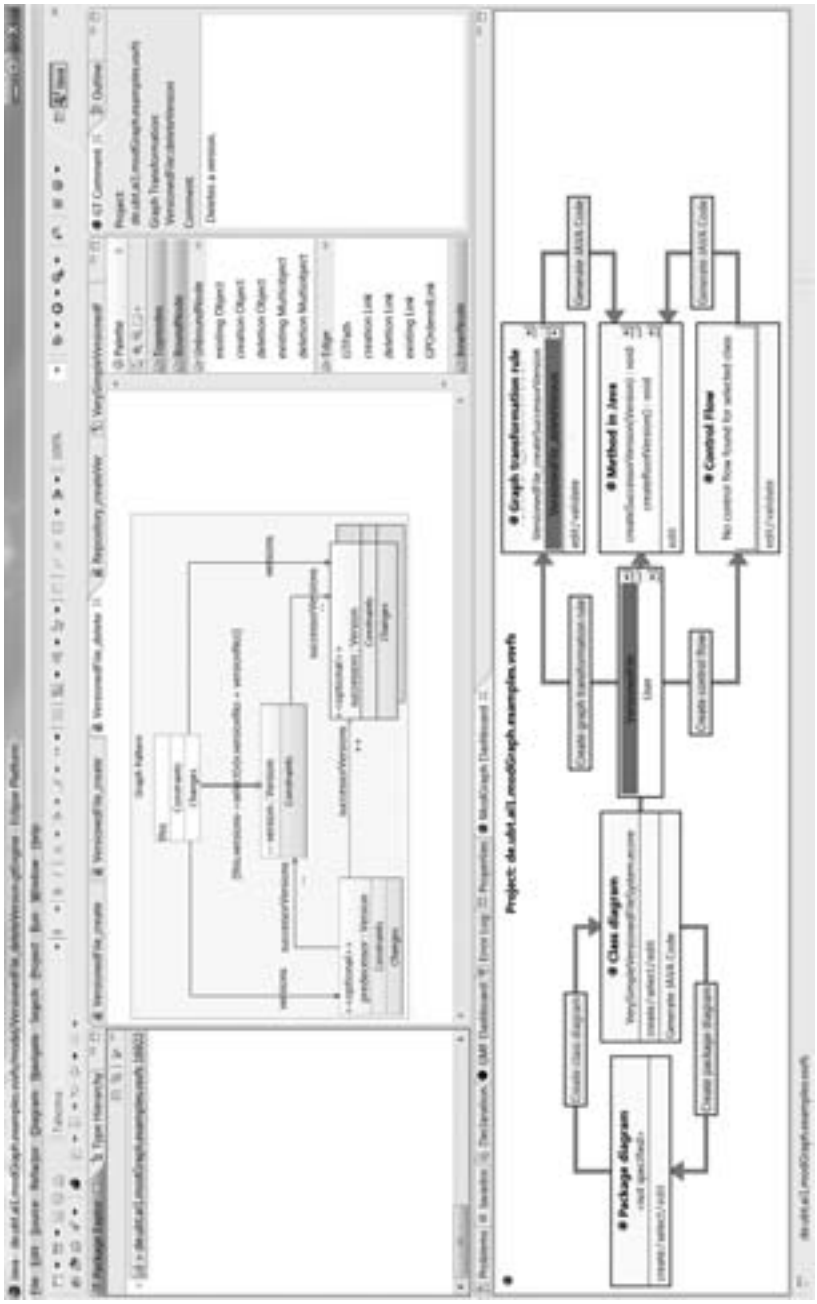

Abbildung 9: Grafische Oberfläche zur Erstellung der ModGraph-Graphtransformationsregeln.

Abbildung 10: EMF- und ModGraph-Codegenerator.

spiel durch eine fehlgeschlagene Nachbedingung. Daher greift der ModGraph Codegenerator zuerst in das EMF-generierte Interface zur Klasse ein und erweitert die Methodendeklaration darin um eine Ausnahmebehandlung.

Die zugehörige Implementierungsklasse wird analog angepasst. Zudem wird, in der Implementierungsklasse, der EMF-generierte JavaDoc-Kommentar erweitert. Ein Importmanager ergänzt die fehlenden Importe der Klasse. Fehlende Importe treten insbesondere dann auf, wenn ein Objekt einer anderen Klasse im gerade zu generierenden Methodenrumpf der Methode modifiziert oder erstellt wird. Zudem werden ModGraph-Bibliotheken eingebunden. In den Methodenrumpf der zu implementierenden Methode wird nun der vom ModGraph-Codegenerator erzeugte Quelltext eingefügt.

Die Utility-Klasse `GTUtil` wird ins EMF-generierte Util-Paket generiert. Die Klasse stellt Methoden zur Mustersuche und zur Prüfung der negativen Anwendbarkeitsbedingung zur Verfügung. Sie erbt weitere Utility-Methoden, wodurch innerhalb der Utility-Klasse Abhängigkeiten zu ModGraph-Bibliotheken entstehen.

Ausführungslogik des ModGraph-generierten Methodenrumpfs. Zuerst werden alle Vorbedingungen mit Hilfe einer ModGraph-eigenen `OCLHandler`-Klasse überprüft. Schlägt eine Vorbedingung fehl, wird eine Ausnahme ausgelöst. Sind alle Vorbedingungen erfüllt, findet die Mustersuche (siehe unten) statt. Diese ist in die generierte Klasse `GTUtil` ausgelagert. Bei erfolgreicher Mustersuche werden alle Objekte von der `GTUtil`-Klasse abgeholt und zunächst die rechten Seiten der Attributzuweisungen berechnet. Dies ist nötig, da sich Attributzuweisungen immer auf den Vorzustand des Modells beziehen. Für zu bewahrende Objekte erfolgt die Zuweisung. Die zu löschenden Elemente werden gelöscht, die zu erzeugenden werden erzeugt. Die zuvor berechneten Attributwerte werden an die neu erzeugten Objekte zugewiesen. Analog zur Vorbedingung wird die Nachbedingung geprüft, die im Falle des Scheiterns zu einer Ausnahme führt.

Mustersuche. Die Mustersuche entspricht einer Teilgraphensuche auf der EMF-Instanz. Die Teilgraphensuche wird benötigt, um die ungebundenen Elemente der Regel zu binden. Sie basiert auf der Navigierbarkeit durch die Graphen der Graphtransformationsregel und ist in unserem Fall als heuristischer Greedy-Algorithmus implementiert. Das Matching ist dabei nicht injektiv. Injektivität kann jedoch durch das Setzen entsprechender Bedingungen an die Knoten erreicht werden. Ausgangspunkt der Suche sind die bereits gebundenen Knoten der Regel. Die Suche nach den ungebundenen Knoten startet an dieser Stelle. Deswegen ist es nötig, dass alle Knoten der Regel von den gebundenen aus erreichbar

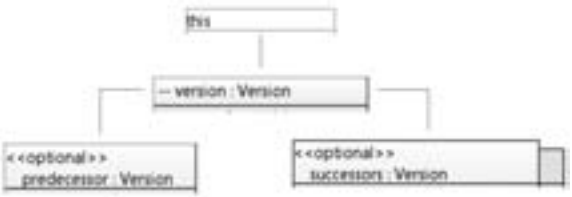

Abbildung 11: Spannwald zur Regel `deleteVersion` aus Abbildung 8.

sind, wobei es bei bidirektionalen Referenzen keine Rolle spielt, ob die Hin- oder die Rückrichtung in der Regel enthalten ist. Bei der Suche von ungebundenen Knoten, ausgehend von gebundenen, spielt die Reihenfolge in der gesucht wird, eine tragende Rolle. Ein obligatorischer Knoten darf niemals von einem optionalen Knoten aus gesucht werden. Außerdem wird ein Link, der eine einwertige Referenz implementiert, einem anderen Link, der eine mehrwertige Referenz implementiert, vorgezogen. Damit werden kombinatorisch explodierende Kosten für die Teilgraphensuche (soweit möglich) vermieden.

Der Algorithmus baut einen Spannwald aus dem gegebenen Teilgraphen auf. Er sucht also zunächst alle gebundenen Knoten und betrachtet diese als Wurzeln im Spannwald. Im nächsten Schritt werden die von allen gebundenen Objekten über Links erreichbaren Knoten betrachtet. Dabei wird derjenige Knoten in den Spannwald eingefügt, der über den Link mit den geringsten Kosten erreichbar ist. Der Knoten ist damit gebunden. Dieses Vorgehen wird zunächst solange wiederholt, bis alle obligatorischen Knoten gebunden sind. Sind optionale Knoten vorhanden, werden diese im nächsten Schritt nach demselben Vorgehen, in den bereits entstandenen Spannwald eingefügt. Der entstandene Spannwald zur Regel `deleteVersion` aus Abbildung 8, ist in Abbildung 11 dargestellt. Hier wird ausgehend vom einzigen gebundenen Objekt, dem this-Objekt, über den Pfad, die zu löschende Version gesucht. Von dieser ausgehend, wird, über die einwertige Referenz, der Vorgänger gefunden und, über die mehrwertige Referenz, die Nachfolgerknoten. Vorgänger und Nachfolger werden nach der zu löschenden Version gesucht, da sie optional sind. Dieser Spannwald bildet die Grundlage für die in der `GTUtil`-Klasse generierten Methoden.

Innerhalb der `GTUtil`-Klasse wird der Spannwald zur Generierung der Methoden ebenenweise durchlaufen. Jede Methode sucht vom Elternknoten aus den Kindknoten. Schlägt dies fehl und entspricht der Elternknoten keinem gebundenen Knoten im Graphmuster, wird der Elternknoten erneut gesucht. Entspricht der Elternknoten einem gebundenen Knoten im Graphmuster, wird eine Ausnahme ausgelöst. Der gesuchte Kindknoten ist in dieser EMF-Instanz nicht enthalten.

4 Abgrenzung zu verwandten Arbeiten

In den vergangenen Jahren hat die Forschung im Bereich der modellgetriebenen Softwareentwicklung ein breites Spektrum an Ansätzen zur Modelltransformation hervorgebracht

[CH06]. Der Begriff Modell-zu-Modell Transformationen besagt, dass ein Zielmodell aus einem Quellmodell abgeleitet wird. Die Ansätze ATL [JK05] und QVT [OMG11] beschäftigen sich mit exogenen Modell-zu-Modell Transformationen. Im Gegensatz dazu unterstützt ModGraph die Modifikation bestehender Modelle (endogene Modelltransformation).

Weiterhin bietet ModGraph im Vergleich zu textuellen Sprachen wie ATL und QVT eine grafische Notation mit darin eingebetteten textuellen Fragmenten, um Modelltransformationsregeln zu beschreiben. Der Vorteil der grafischen Notation liegt in der einfacheren Verständlichkeit von komplexen Modelltransformationen. Das zugrunde liegende Konzept hinter ModGraph sind Graphtransformationen. Im Kontext von Graphtransformationen wurden über die Jahre viele Sprachen und Werkzeuge entwickelt [EEKR99], die in den meisten Fällen aber keine Verbindung zum EMF Rahmenwerk bieten. Nur eine kleine Anzahl von Ansätzen, auf die wir uns hier im weiteren Verlauf beziehen, bietet eine Unterstützung für EMF.

Ein Ansatz, der statt der grafischen Notation - wie in ModGraph - auf eine textuelle Repräsentation der Graphtransformationsregeln setzt, ist VIATRA2 [VB07]. VIATRA2 wurde *mit* Hilfe des EMF Rahmenwerks gebaut, indem das eigene Metamodell mit Ecore beschrieben wurde. Im Gegensatz dazu wurde ModGraph *für* das EMF Rahmenwerk gebaut. Im besonderen verwenden wir Ecore für das strukturelle Modell.

Wie ModGraph verwendet auch TIGER [BEK+06] Ecore zur strukturellen Modellierung. Allerdings bietet TIGER nur Unterstützung für Graphtransformationsregeln, die viele der fortgeschrittenen Konstrukte, die ModGraph bietet (Mengenknoten, Vor- und Nachbedingungen, Pfade, Methodenaufrufe und OCL Integration), vermissen lassen. Der Nachfolger namens Henshin [ABJ+10] erweitert die Graphtransformationsregeln von TIGER mit Mengenknoten, Kontrollstrukturen und Vor- bzw. Nachbedingungen. Pfade und Methodenaufrufe und die Möglichkeit, eigene OCL-Constraints anzugeben, fehlen bisher. Wie ModGraph bieten auch TIGER und sein Nachfolger Henshin die Möglichkeit, negative Anwendbarkeitsbedingungen zu spezifizieren.

Graphtransformationsregeln werden in TIGER in eigene Klassen anstatt Methoden übersetzt. Möchte nun ein Anwendungsentwickler eine benutzerdefinierte Operation, die im Ecore Modell spezifiziert wurde, implementieren, so muss dieser zunächst Code schreiben, um die der Regel zugeordnete Klasse zu instanziieren. Anschließend müssen der Instanz die notwendigen Parameter übergeben werden, bevor schließlich die Regel ausgeführt werden kann. Dies bedeutet aber, dass mehrere Schritte notwendig sind, um eine Regel aufzurufen, was zu hohem Programmieraufwand und hohen Laufzeiten führt. Im Gegensatz dazu ist in ModGraph eine Graphtransformationsregel einer benutzerdefinierten Operation zugeordnet. Weiterhin wird durch den ModGraph Compiler eine gewöhnliche Java Methode generiert. Aus der Sicht eines Anwendungsentwicklers bietet ModGraph somit eine nahtlose und effiziente Integration des aus den Graphtransformationsregeln generierten Java Quelltextes, was TIGER nicht bietet. Bis jetzt bietet Henshin nur einen Interpreter, was die Einbettung von Graphtransformationsregeln in eigene Anwendungen erschwert.

Die objektorientierte Modellierungsumgebung Fujaba [Zün01] bietet Klassendiagramme

zur Modellierung der statischen Struktur, sowie Storydiagramme zur Verhaltensmodellierung. Ein Storydiagramm realisiert hierbei genau eine Methode aus dem Klassendiagramm und stellt ein Kontrollflussdiagramm dar. Es enthält sog. Statement-Aktivitäten (dies sind reine Java Quelltextfragmente), sowie Story-Muster, die Graphtransformationsregeln darstellen. Fujaba wurde unabhängig von EMF entwickelt, die Idee der Storydiagramme wurde aber zumindest partiell in EMF reimplementiert [GHS09]. Das entsprechende Werkzeug MDELab verwendet Ecore Klassendiagramme zur Beschreibung der statischen Struktur und bietet einen grafischen Editor und einen Interpreter für Storydiagramme. Dies bedeutet, dass der Endanwender auf den Interpreter beschränkt ist, um mit Modellinstanzen zu arbeiten, während in unserem Ansatz EMF kompatibler Quelltext erzeugt wird, der auf alle mögliche Arten verwendet werden kann, so z. B. auch für die Generierung eines grafischen Editors mit GMF, das seinerseits EMF-generierten Code voraussetzt.

Die Ausdrucksmächtigkeit von Story-Mustern in Fujaba ist geringer als die von Graphtransformationsregeln in ModGraph, da nur zweitere Vor- und Nachbedingungen, negative Anwendbarkeitsbedingungen (diese stellen verbotene Muster im Graphen dar) sowie eine OCL-Integration bieten[2].

Der Literaturvergleich zeigt, dass es bereits einige EMF-basierte Systeme für die Erstellung und die Ausführung von Graphtransformationsregeln gibt. ModGraph zeichnet sich jedoch durch eine nahtlose Integration mit dem EMF-Rahmenwerk und den Compiler-Ansatz aus. Es ist das einzige System, das aus Graphtransformationsregeln Code erzeugt, der den von EMF generierten Code erweitert.

5 Zusammenfassung und Ausblick

In diesem Beitrag haben wir ModGraph, ein Werkzeug das EMF mit Verhaltensmodellierung ergänzt, vorgestellt. Mit Hilfe von Graphtransformationsregeln bietet ModGraph die Möglichkeit das Verhalten von Operationen, die im Ecore-Klassendiagramm spezifiziert wurden, zu beschreiben. Somit kann eine Lücke im modellgetriebenen Entwicklungsprozess mit EMF geschlossen werden. Derzeit unterstützt ModGraph die grafische Notation von Graphersetzungsregeln und ist in der Lage, aus diesen Regeln ausführbaren Java Quelltext zu erzeugen. Aktuelle und künftige Arbeiten zielen darauf ab, die Sprache dahingehend zu erweitern, dass auch Kontrollstrukturen auf einer höheren (und damit plattformunabhängigen) Ebene spezifiziert werden können.

Literatur

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause und Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations.

[2] Bezüglich der OCL-Integration gab es einen prototypischen Ansatz für Fujaba, der es aber nicht bis zum Release Stadium geschafft hat [SZG06].

In Dorina C. Petriu, Nicolas Rouquette und Øystein Haugen, Hrsg., *Proceedings 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*, *Lecture Notes of Computer Science*, Band 6394, Seiten 121–135, Oslo, Norway, Oktober 2010. Springer-Verlag.

[BDK09] Thomas Buchmann, Alexander Dotor und Martin Klinke. Supporting Modeling in the Large in Fujaba. In Pieter van Gorp, Hrsg., *Proceedings of the 7th International Fujaba Days*, Seiten 59–63, Eindhoven, The Netherlands, November 2009.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer und Eduard Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In Oscar Nierstrasz, Jon Whittle, David Harel und Gianna Reggio, Hrsg., *Proceedings 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, *Lecture Notes of Computer Science*, Band 4199, Seiten 425–439, Genova, Italy, Oktober 2006. Springer-Verlag.

[BWW11] Thomas Buchmann, Bernhard Westfechtel und Sabine Winetzhammer. MODGRAPH - A Transformation Engine for EMF Model Transformations. In Maria Jose Escalona, Boris Shishkov und José Cordeiro, Hrsg., *Proceedings of the 6th International Conference on Software and Data Technologies*, Seiten 212 – 219. INSTICC, 2011.

[CH06] Krzysztof Czarnecki und Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski und Grzegorz Rozenberg, Hrsg. *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, Band 2, World Scientific, Singapore, 1999.

[GHS09] Holger Giese, Stephan Hildebrandt und Andreas Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In Artur Boronat und Reiko Heckel, Hrsg., *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, *Electronic Communications of the EASST*, Band 18, York, UK, März 2009. 12 p.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. The Eclipse Series. Addison-Wesley, Boston, MA, 1st. Auflage, 2009.

[JK05] Frédéric Jouault und Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, Hrsg., *MoDELS Satellite Events*, *Lecture Notes of Computer Science*, Band 3844, Seiten 128–138. Springer-Verlag, 2005.

[OMG11] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1*. OMG, Januar 2011.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro und Ed Merks. *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Boston, MA, 2. Auflage, 2009.

[SZG06] Mirko Stölzel, Steffen Zschaler und Leif Geiger. Integrating OCL and Model Transformations in Fujaba. In Dan Chiorean, Birgit Demuth, Martin Gogolla und Jos Warmer, Hrsg., *Proceedings of the 6th OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)*, *Electronic Communications of the EASST*, Band 5, Genova, Italy, Oktober 2006. 16 p.

[VB07] Dániel Varró und András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.

[Zün01] Albert Zündorf. Rigorous Object Oriented Software Development. Bericht, University of Paderborn, Germany, 2001.