Graphische Zuordnung von Elementen einer Modelltransformation

Stanislaw Nasin

Bachelor-Arbeit
eingereicht im Jahr 2013

Christian-Albrechts-Universität zu Kiel
Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme

Betreut durch: Dipl.-Inf. Christoph Daniel Schulze

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Zusammenfassung

Modellgetriebene Softwareentwicklung hebt komplexe Datenstrukturen und Informationen auf eine menschenfreundlichere Abstraktionsebene. Sie erleichtert das Verständnis und erhöht die Qualität der erzeugten Software. Dabei werden die Daten in Form von Modellen erstellt, modifiziert, graphisch angezeigt und für bestimmte Zwecke in andere Modellstrukturen transformiert.

In dieser Arbeit wird ein Projekt vorgestellt, das die graphische Zuordnung der Elemente einer Modelltransformation ermöglicht und damit das Verständnis zusätzlich verbessert. Dazu wird auch eine Möglichkeit entwickelt, die einzelnen Transformationen zwischen Modellen graphisch darzustellen, was die Qualität der Transformationen und der Modelle steigert und die Suche nach Fehlern erleichtert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Mit dem Fortschritt der Technologie und den wachsenden Anforderungen der Anwender stellt sich die Informatik immer neuen Herausforderungen. Um diese Herausforderungen zu bewältigen, werden immer mehr verschiedene Ansätze verwendet. Ein populärer Ansatz ist die modellgetriebene Softwareentwicklung (Model-Driven Software Development oder MDSD) [9]. Der Sinn dabei ist es, die zugrunde liegenden Systeme oder Informationen in Form von Modellen zu beschreiben. Das soll dazu dienen, die Komplexität zu verringern und den Entwicklungsprozess handhabbar zu machen. Dabei kann diese Wirkung noch verstärkt werden.

1.1 Einführung und Motivation

Der Einsatz von modellgetriebener Softwareentwicklung steigert in de meisten Fällen die Produktivität des Entwicklers und die Qualität des Produktes [10]. Dies wird dadurch erreicht, dass einzelne Komponente der Systeme oder Informationen durch die Modelle auf eine menschenfreundlichere Abstraktionsstufe gebracht werden. Auf diese Weise lassen sich diese leichter überblicken und auf Fehler oder Unstimmigkeiten prüfen. Ein weiterer wichtiger Punkt bei der modellgetriebenen Softwareentwicklung ist das Benutzen automatisierter Operationen wie Modelltransformationen auf den Modellen. Dadurch können einzelne Informationen zu weiteren Verarbeitung in andere Strukturen überführt werden.

Zwar wird durch den Einsatz von modellgetriebener Softwareentwicklung einiges erleichtert, es könnte jedoch noch einfacher werden. Es wäre von Vorteil, wenn die einzelnen Modelle und vor allem die Modelltransformationen zusätzlich noch graphisch dargestellt werden könnten. Dadurch wäre es möglich, einzelne zusammenhängende Informationen, wie die Beziehung der Elemente innerhalb eines Modells oder die Abstammung der Elemente eines Modells von Elementen eines anderen durch eine Modelltransformation, in einer für den Modellschaffer bzw. -nutzer einfacheren Form darzustellen. Zum Beispiel wäre es möglich, die Transformation eines beliebigen Modells zu visualisieren (siehe Abbildung 1). In diesem Beispiel wird gezeigt, wie das Modell eines regulären Ausdrucks zum Modell eines nichtdeterministischen endlichen Automaten transformiert wird. Dabei wird das Modell des regulären Ausdruck „a konkateniert mit b“ durch einen Graphen mit einer Baumstruktur dargestellt. Die Operationen sind dabei die inneren Knoten und die Literale die Blätter. Das entstandene Modell des nichtdeterministischen endlichen Automaten wird durch einen gerichteten Graph dargestellt, bei dem die Beschriftung der Kanten das erwartete Literal repräsentieren. Es erwartet als erstes ein a und dann ein b. Die beiden Modelle werden graphisch dargestellt und die einzelnen

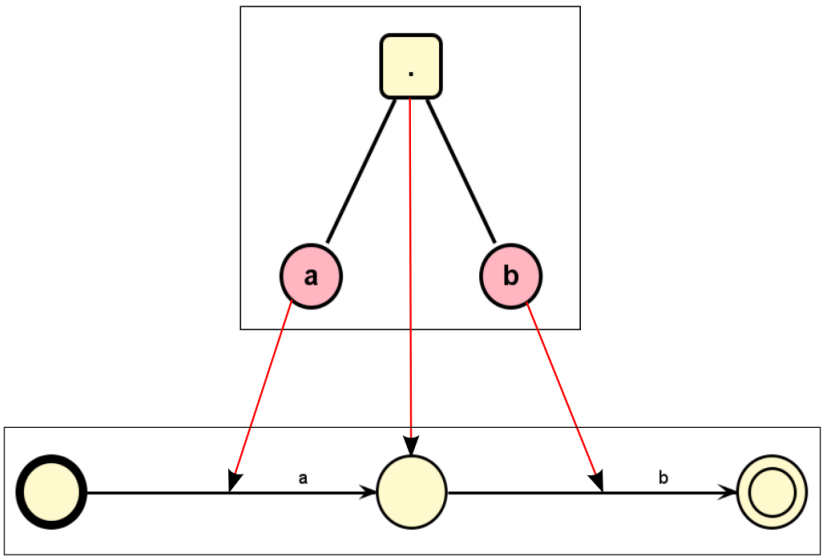

Abbildung 1. Beispiel zur Visualisierung einer Modelltransformation

Modellelemente anhand der Transformationsschritte einander zugeordnet.Auf diese Weise ist es möglich, einen noch größeren Überblick über die gegebenen Informationen und auch über deren Korrektheit zu gewinnen.

Im Vordergrund dieses Ansatzes stehen also die *Modelle* und die *Transformationen* zwischen den einzelnen Modellen [10]. Zusätzlich ist auch die *Visualisierung* der Modelle und der Transformationen wichtig. Aus diesem Grund ist es sinnvoll, als erstes diese Begriffe näher zu beschreiben.

1.1.1 Modell

Ein *Modell* ist eine abstrakte Darstellung der Wirklichkeit, die meist viel zu komplex ist, um genau abgebildt zu werden [1]. Aus diesem Grund stellt ein Modell auch nur die wesentlichen Faktoren der Wirklichkeit dar. Ein Modell kann nach Herbert Stachowiak durch mindestens drei Merkmale gekennzeichnet werden [11].

Abbildungsmerkmal sagt aus, dass ein Modell eine Abbildung eines natürlichen oder eines künstlichen Originals ist, das wiederum ein Modell sein kann.

Verkürzungsmerkmal beschreibt, dass ein Modell nicht alle Faktoren des Originals enthält. Es nimmt nur die Faktoren auf, die dem Modellentwickler oder Modellnutzer bedeutsam erscheinen.

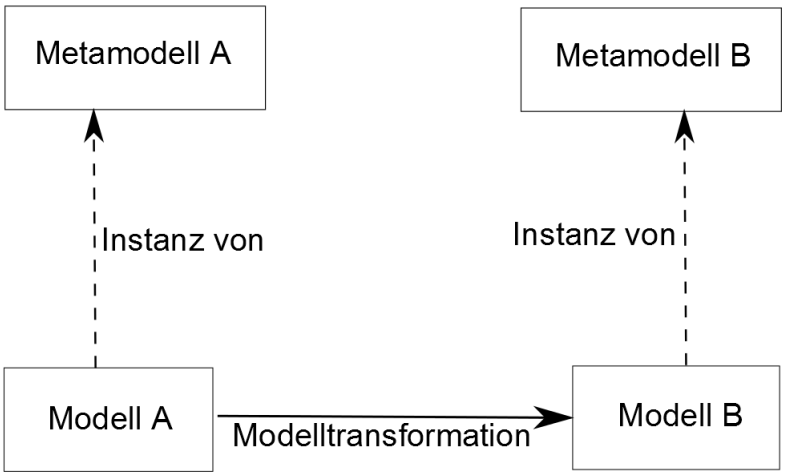

Abbildung 2. Modelltransformation

Pragmatismusmerkmal drückt schließlich aus, dass die einzelnen Modelle ihren Originalen nicht eindeutig zugeordnet werden können. Modelle ersetzen lediglich das Original zu bestimmter Zeit, für bestimmte Betrachter und unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen. Das bedeutet, dass ein Modell nur einen Zustand des Originals zu einer bestimmten Zeit darstellt und dabei auf den Betrachter und den Zweck angepasst ist.

Um zu beschreiben, wie die Modelle aufgebaut sein dürfen, brauchen Sie eine gemeinsame abstrakte Syntax, genannt *Metamodell*. Das Metamodell beschreibt, welche Elemente in einem Modell verwendet und wie diese in Beziehung zueinander gesetzt werden dürfen.

1.1.2 Modelltransformation

Allgemein bezeichnet eine *Transformation* eine Umformung oder Umgestaltung [1]. Bei der modellgetriebenen Softwareentwicklung bezeichnet die *Modelltransformation*, auch *Modell-zu-Modell-Transformation* genannt, die Überführung von einem Quellmodell in ein Zielmodell [10]. Wie in Abbildung 2 zu sehen, sind dabei das Quell- und das Zielmodell jeweils konform zu einem Metamodell.

Der Vorgang der Transformation ist in meisten Fällen automatisiert und

geschieht in Abhängigkeit von einer sogenannten *Transformationsdefinition*. So eine Transformationsdefinition ist dabei eine Menge von Regeln, den *Transformationsregeln*. Diese beschreiben, wie Elemente des Quellmodells in Elemente des Zielmodells transformiert werden [6].

1.1.3 Visualisierung

Mit der *Visualisierung* wird im Allgemeinen ein Prozess beschrieben, bei dem abstrakte Daten und Zusammenhänge in eine graphische Form gebracht werden. Nach Gershon et al. liefert die Visualisierung eine Schnittstelle zwischen dem menschlichen Verstand und dem modernen Computer [3]. Dabei ist die Visualisierung eine Transformation von Daten, Informationen oder auch Wissen in eine visuelle Form. Durch effektive visuelle Schnittstellen ist es möglich mit großen Mengen an Daten schnell und effektiv zu arbeiten, um verborgene Charakteristiken, Muster und Tendenzen zu entdecken. Zum Beispiel ist nach Price et al. die *Softwarevisualisierung* das Verwenden von Typographie, Graphikdesign, Animationen, Kinematografie und interaktiven Computergraphiken für die Verbesserung der Schnittstelle zwischen dem Softwareentwickler bzw. Softwarenutzer und dem Programm [8].

1.2 Problemstellung

Wie zuvor beschrieben, wäre es für die Modellentwickler und für die Modellnutzer wünschenswert, die Modelle und die Modelltransformationen, die bei der modellgetriebenen Sotfwareentwicklung entstehen, graphisch darzustellen. Dadurch wäre es möglich, Gebrauch von den natürlichen visuellen Fähigkeiten des Menschen zu machen, um einzelne Zusammenhänge und versteckte Charakteristiken in komplexen Systemen zu erkennen. Bei der Umsetzung dieser Idee ergeben sich jedoch einige Schwierigkeiten, sowohl bei der Visualisierung einzelner Modelle, als auch bei der Visualisierung der Modelltransformationen.

Im Folgenden wird die Aufmerksamkeit auf die Probleme im Bezug auf die Visualisierung der Transformationen zwischen den Modellen gelegt. Die Schwierigkeiten im Bezug auf die Visualisierung einzelner Modelle spielen dabei keine geringe Rolle.

Die hauptsächliche Schwierigkeit bei der Visualisierung von Modelltransformationen besteht darin, dass alle Transformationsschritte gemerkt bzw. gespeichert sein müssen, um im Nachhinein durch diese die Zuordnung der einzelnen Elemente zueinander zu finden. Dabei müssen nicht nur die einzelnen Schritte der Transformation zwischen den Modellen, sondern auch alle Schritte der Transformationen einzelner Modelle zu ihren graphischen Darstellungen gespeichert werden (siehe Abbildung 3).

Eine weitere Schwierigkeit ergibt sich gerade aus diesem Speicherverfahren. Bei der Modelltransformation kann es vorkommen, dass ein Element des Quellmodells in mehrere Elemente des Zielmodells transformiert wird. Ebenso kann es sein, dass bei der Transformation eines Modells zu seiner graphischen Darstellung ein Element mehrere graphische Elemente oder auch kein graphisches Element erzeugt. Dabei ist auch zu bedenken, dass ähnliche Elemente eines Modells oftmals auf verschiedene Weisen transformiert werden. Zum Beispiel werden die verschiedenen Operationen eines regulären Ausdrucks unterschiedlich transformiert (siehe Abbildung 4).

Für die Umsetzung dieser Idee und das Lösen der gegebenen Schwierigkeiten ist es als nächstes sinnvoll, sich ähnliche Arbeiten anzusehen, um zu schauen, wie die Schwierigkeiten gelöst werden können.

1.3 Verwandte Arbeiten

Die Idee einer Zuordnung zwischen den Elementen einer Modelltransformation ist nicht unbedingt neu. Jedoch wurde diese noch nicht so oft umgesetzt. Und eine Visualisierung dieser Zuordnung ist nur von den wenigsten Tools ermöglicht. In den meisten Fällen wird so eine Zuordnung auf Transformationen zwischen Modell und Text oder zwischen Modellen eines gleichen Typs,

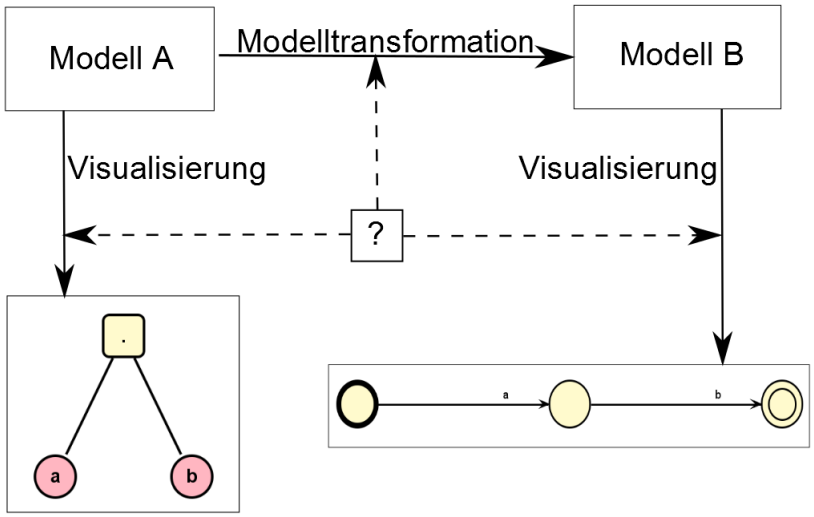

Abbildung 3. Die zu speichernden bzw. merkenden Transformationen

wie zum Beispiel Ecore oder UML, ermöglicht. Dabei werden in vielen Fällen die Daten innerhalb der einzelnen Modellelemente verglichen, um den Bezug zwischen diesen herzustellen. In anderen Fällen ist so eine Elementzuordnung nur in Verbindung mit einer bidirektionalen Transformation möglich.

Eine Möglichkeit zur Zuordnung der Elemente einer Modelltransformation bietet *QVT*, *Query/Views/Transformations*. Dafür erzeugt QVT eine Datenstruktur, die das Quell- und das Zielmodell der Transformation enthält [2]. Zusätzlich werden in dieser Struktur auch die einzelnen Beziehungen zwischen den Elementen der Modelle gespeichert. Diese Beziehungen werden von QVT bei der Modelltransformation automatisch erzeugt und verweisen auf Referenzen zu den entsprechenden Modellelementen. Auf diese Weise ermöglicht QVT die Zuordnung der Elemente einer Modelltransformation. Die von QVT unterstützte Transformationen sind templatebasiert und eignen sich deswegen nicht ideal für alle Zwecke. Zusätzlich bietet QVT keine Möglichkeit die Zuordnung der Modellelemente einer Transformation graphisch darzustellen.

Eine graphische Zuordnung der Elemente einer Modelltransformation bietet *ATL*, *ATLAS Transformation Language* [13]. Der Ansatz dieser Zuordnung ist ähnlich dem Ansatz von QVT [5]. Bei der Transformation von Modellen wird eine weitere Datei erzeugt, in der die Beziehungen zwischen den einzelnen Elementen der Modelle gespeichert werden. Dafür muss an der Modelltrans-

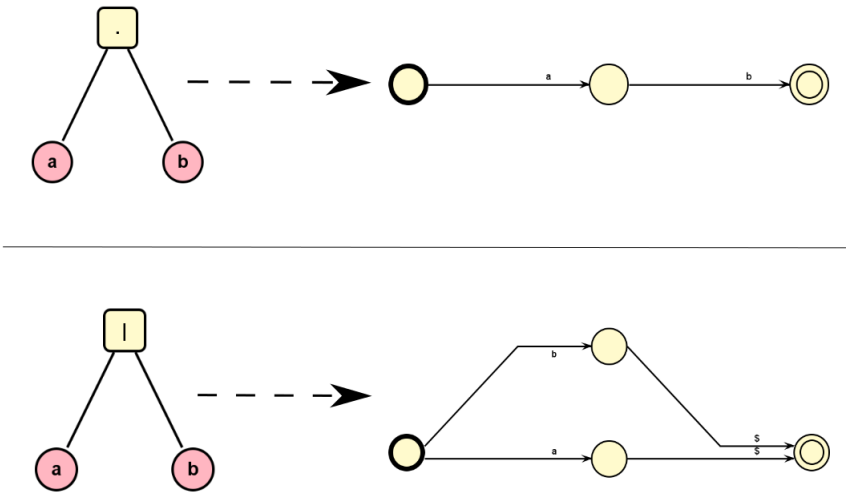

Abbildung 4. Transformation ähnlicher Elemente

formation eine notwendige Modifikation vorgenommen werden. Diese muss leider an die einzelnen Transformationen angepasst sein. Diese Modifikation bewirkt keine Änderung an der Transformationslogik. Um die entstandenen Verbindungen zu visualisieren, kann anschließend *AMW*, *ATLAS Model Weaver*, verwendet werden. Auf diese Weise ermöglicht ATL in Verbindung mit AMW die graphische Zuordnung von Elementen einer Modelltransformation, ohne Einschränkungen auf den Typ des Modells oder der Transformation.

In dieser Arbeit wird ein ähnliches Konzept vorgestellt, dass eine graphische Zuordnung von Elementen einer Modelltransformation erlaubt. Dabei müssen die Transformationsmodifikationen jedoch nicht an die Transformationen angepasst werden. Zusätzlich wird durch den Einsatz des KIELER-Projektes die Visualisierung der Zuordnung enorm erleichtert.

1.4 Aufbau dieser Arbeit

In dieser Arbeit wird ein Konzept vorgestellt, das eine graphische Zuordnung von Elementen einer Modelltransformation ermöglicht. Dabei soll die Übersichtlichkeit über die Abstammung der Elemente einzelner Modelle und die Korrektheit der Modelle an sich erhöht werden. Im folgenden Kapitel werden als erstes die in diesem Zusammenhang verwendeten Technologien vorgestellt. Als nächstes werden in Kapitel 3 verschiedene Ideen vorgestellt, die bei der Anfertigung des Konzepts entstanden sind. Dabei werden auch die Probleme erläutert, die damit verbunden waren. In Kapitel 4 wird die Implementierung des Konzepts bzw. Umsetzung der Ideen erklärt. Es wird Augenmerk auf Sichten des Entwicklers und des Anwenders gelegt und mit Beispielen verdeutlicht. Das Kapitel 5 beschreibt dann verschiedene Versuche, mit denen das Konzept getestet wurde. Zum Schluss wird in dem Kapitel 6 eine Zusammenfassung der Ergebnisse vorgenommen und ein Ausblick für die Zukunft angegeben.

2 Verwendete Technologien

Dieses Kapitel beschäftigt sich mit den Technologien, die im Rahmen dieses Projektes eingesetzt wurden. Die meisten davon sind dabei Erweiterungen von *Eclipse*.

2.1 Eclipse

Ursprünglich ist *Eclipse*[1] als eine quelloffene integrierte Entwicklungsumgebung (Integrated development environment oder IDE) für Java entstanden. Durch seine Erweiterbarkeit wird Eclipse jedoch, im Gegensatz zu anderen Entwicklungsumgebungen, auch für andere Programmier- und Modellierungssprachen eingesetzt. Zum Beispiel ist es durch bestimmte Erweiterungen möglich Sprachen wie C, Scala, Unified Modeling Language (UML) und viele weitere zu verwenden.

2.1.1 Plug-in-Architektur

An sich basiert die Entwicklungsumgebung Eclipse auf *Equinox* [4]. Es ist ein Java-basiertes *OSGi-Framework*, welches die Kernspezifikation von *OSGi* implementiert und somit das Grundgerüst für Eclipse bildet. Bei der Spezifikation

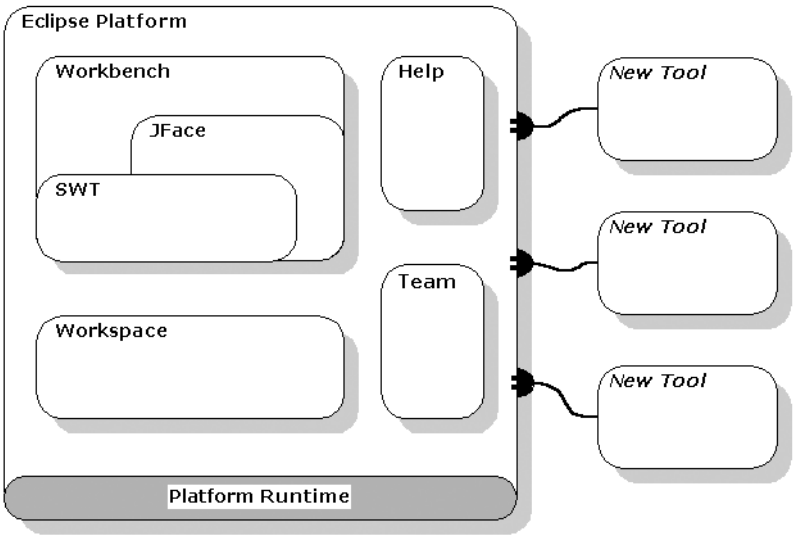

Abbildung 5. Die Plug-in-Architektur von Eclipse [7]

[1] http://www.eclipse.org/org/

von OSGi handelt es sich um eine Vorschrift für hardwareunabhängige dynamische Softwareplattform, die es erleichtert, Anwendungen und Dienste als einzelne Komponente zu modularisieren und zu verwalten. Also wird durch Equinox ermöglicht, dass einzelne Anwendungen von Eclipse in Komponenten aufgeteilt werden. Zusätzlich implementiert Equinox noch weitere Funktionalitäten, wie *Extension Registry*, die es ermöglichen einzelne Komponenten, die sogenannten *Plug-ins*, zu registrieren, zu verwenden und füreinander zur Verfügung zu stellen.

Die *Plug-ins* stellen die Erweiterungen für Eclipse dar. Dabei gilt die kleinste Einheit einer Funktion der Eclipse Plattform als ein Plug-in [7]. In meisten Fällen werden kleine Funktionalitäten durch einzelne Plug-ins eingeführt. Im Gegensatz dazu werden größere und komplexere Funktionalitäten in einzelne Funktionen aufgeteilt, die jeweils ein Plug-in bilden (siehe Abbildung 5).

Damit die Plug-ins auch miteinander arbeiten können, stellen sie *Extension-Points* und *Extensions* zur Verfügung. Dabei deutet ein *Extension-Point* darauf hin, dass diese Anwendung erweitert werden kann bzw. soll, und spezifiziert welche Klassen und Operationen dafür implementiert werden müssen. Im Gegensatz dazu stellt eine *Extension* eine Erweiterung dar, die an einem entsprechenden Extension-Point registriert werden kann und damit die benötigten Klassen und Operationen zur Verfügung stellt, um die Anwendung zu erweitern. Dieses Verfahren funktioniert wie ein Puzzle. Die Extension-Points

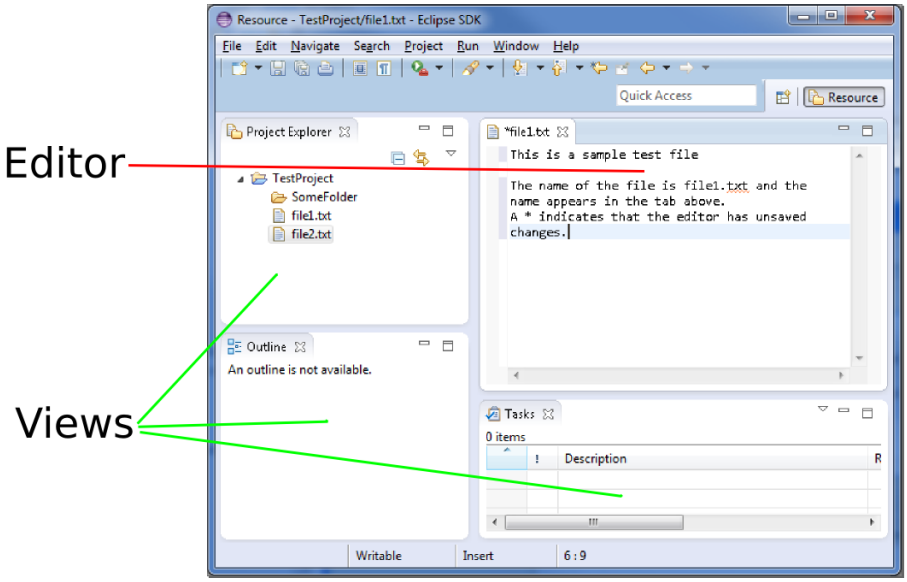

Abbildung 6. Die Benutzeroberfläche von Eclipse

stellen hierbei die Einkerbungen und die Extensions die Ausbuchtungen dar. Dabei kann es für eine Einkerbung natürlich mehrere Ausbuchtungen geben. Auf diese Weise lassen Anwendungen, die Extension-Points zur Verfügung stellen, sich beliebig groß erweitern, ohne dass etwas an dem Code innerhalb der Anwendung verändert werden muss. Auf die gleiche Weise entsteht auch die graphische Benutzeroberfläche (Graphical user interface oder GUI). Diese Oberfläche besteht aus der *Workbench*, der Arbeitsfläche, die sich aus vielen Einzelteilen, wie Editoren und Views, zusammensetzt, die untereinander interagieren (siehe Abbildung 6).

2.1.2 EMF

Das *EMF* oder *Eclipse Modeling Framework* ist ein quelloffenes *Framework*, also ein Programmiergerüst, zur Entwicklung von modellgetriebenen Anwendungen [12]. In Eclipse bildet das EMF das wichtigste Projekt für die modellgetriebene Softwareentwicklung. Das EMF ist dabei eine Einrichtung, die automatisierte Erzeugung von Quelltext anhand von strukturierten Modellen ermöglicht. Dieser Quelltext repräsentiert anschließend die Modelle und dient zur Erzeugung und Bearbeitung dieser. Für diese Funktionalität bietet EMF eine Reihe von Werkzeugen, wie die *Modellierungssprache* Ecore, die zur Bildung der Modelle dient, oder das Quelltextgenerierungs-Framework, das zur Erzeugung von Quellcode als Plug-ins für Eclipse notwendig ist. Das EMF funktioniert wie folgt.

Das EMF enthält das *Ecore-Meta-Metamodell*. Es ist ein Modell, welches zur Beschreibung aller Metamodelle fungiert. Da dieses auf dem *EMOF-Standard*, einem Standard zur Beschreibung von Modellierungssprachen, basiert, können die darauf beruhenden Modelle mithilfe des Austauschformats *XMI* (XML Metadata Interchange) persistiert werden. Anhand dieses Ecore-Meta-Metamodells wird ein Modell erstellt, das die gewünschte Datenstruktur beschreibt, das sogenannte *Metamodell*. Dieses Metamodell dient als Beschreibung für die einzelnen Modellinstanzen, auch einfach Modelle genannt, die den gewünschten Inhalt repräsentieren (siehe Abbildung 7). Zum Beispiel ist das Modell eines bestimmten nichtdeterministischen endlichen Automaten, hier bereits graphisch dargestellt, abhängig von dem Metamodell, das die abstrakte Syntax eines nichtdeterministischen endlichen Automaten beschreibt (siehe Abbildung 8). Im Weiteren wird aus dem Metamodell ein Generator-Modell erzeugt, das einige zusätzliche Informationen enthält. Mithilfe dieses Generator-Modells lässt sich schließlich Javacode generieren, mit dem sich einzelne dem Metamodell entsprechende Modellinstanzen erzeugen, abfragen, manipulieren, validieren und auch serialisieren lassen. Auf diese Weise steigert EMF die Produktivität der Entwickler und reduziert Wiederholungen und damit mögliche Fehlerquellen.

2.1.3 Xtext

Xtext[2] ist ein textuelles quelloffenes Modellierungswerkzeug und ist wie EMF ein Teil von Eclipse Modeling Project. Mithilfe von Xtext ist es möglich eigene textuelle Programmiersprachen und auch domänenspezifische Sprachen zu entwickeln, die dem Erzeugen und Lesen von EMF-basierten Modellen dienen. Im Kern von Xtext steht die Grammatiksprache, mit der eine textuelle Syntax, die Grammatik, für EMF-basierte Modelle erzeugt wird. Diese Grammatik bestimmt die Modellierungssprache und erstellt einen Parser, der Modelle aus der Grammatik entsprechenden Texten erzeugt. Eine gültige Grammatik sieht zum Beispiel wie folgt aus.

```
Entity: 'entity' name=ID '{' (properties+=Property)* '}';
Property: 'property' name=ID ':' type=ID (many?='[]')?;
```

Diese Beispiel-Grammatik entspricht dem Metamodell in der Abbildung 9. Für die Eingabe der Modellierungssprache wird zusätzlich ein Texteditor generiert, der einige Funktionen, wie Autovervollständigung, statische Analyse, Syntaxhervorhebung usw. unterstützt. Der Quelltext, der der oben angegebenen Grammatik entspricht, sieht zum Beispiel wie folgt aus.

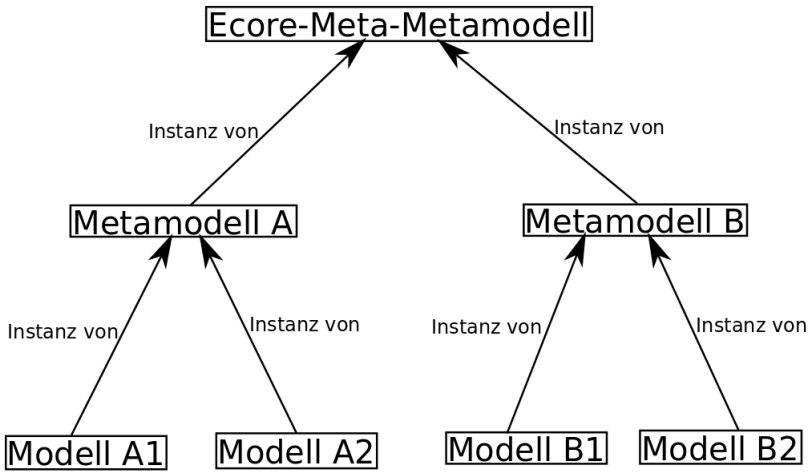

Abbildung 7. Die Abhängigkeiten der Modelle

[2]http://www.eclipse.org/Xtext/documentation.html

```
entity Client {
        property Name : String
        property ownedBooks : String[]
}
```

Auf diese Weise erlaubt Xtext schnelleres Erzeugen und Bearbeiten von einzelnen Modellinstanzen und eine leichte und unproblematische Änderung der Modellierungssprache.

2.1.4 Xtend

Xtend[3] ist eine Programmiersprache und wird in vielen Fällen zur Definition von Modell-zu-Modell-Transfomationen verwendet. Diese Sprache ist vollständig mit Jave kompatibel. Sie wird nicht zu Byte-Code compiliert, sondern zu Java-Code übersetzt. Aus diesem Grund ist Xtend Java sehr ähnilch.

Jedoch bietet Xtend einige Vorzüge. Einer dieser Vorzüge sind die *Extension-Methoden*, durch welche Xtend auch seinen Namen erhalten hat. Diese Methoden ermöglichen es, Klassen um Methoden zu erweitern ohne den Quell-

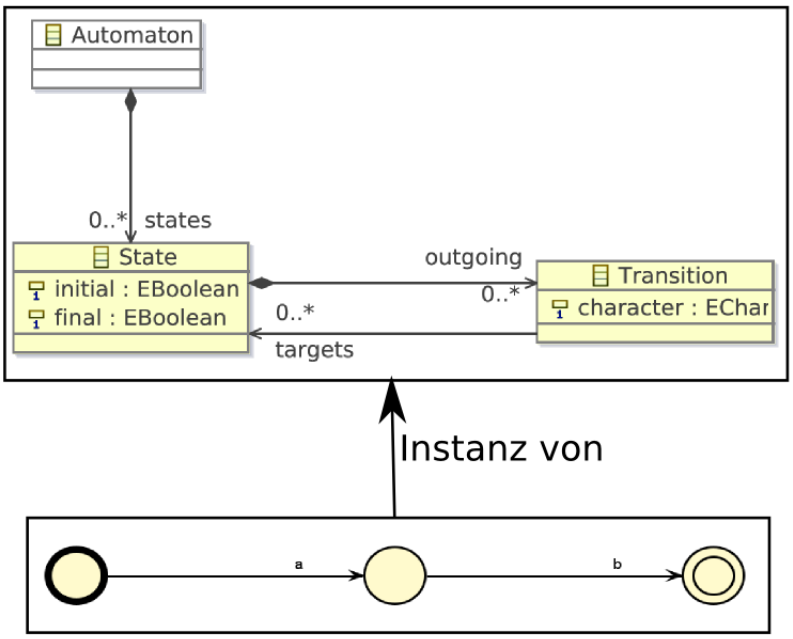

Abbildung 8. Beispiel zur Abhängigkeit der Modellinstanz vom Metamodell

[3]http://www.eclipse.org/xtend/documentation.html

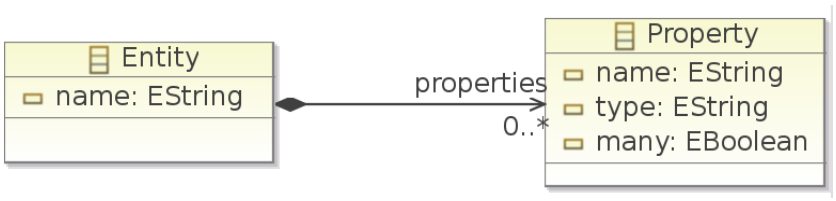

Abbildung 9. Der Xtext-Grammatik entsprechendes Metamodell

code der Klasse zu verändern. Das wird durch einen einfachen syntaktischen Trick ermöglicht. Statt beim Aufruf der Methode das Argument in den Klammern zu setzen, wird sie auf dem Argument aufgerufen, so als wäre die Methode ein Teil des Arguments. Das sieht dann wie folgt aus. Normaler Methodenaufruf:

```
toFirstUper("hello")
```

Aufruf der Methode als eine Extension-Methode:

```
"hello".toFirstUper()
```

In vielen Fällen wird auf diese Weise in Xtend geschriebener Quellcode leserlicher und übersichtlicher gemacht, ohne an Effizienz und Leistung zu verlieren. Die Xtend-Bibliothek stellt eine ganze Reihe von vordefinierten Extension-Methoden zur Verfügung, die ohne weiteres mit den vorhandenen Java-Klassen benutzt werden können. Eine sehr nützliche Extension-Methode ist dabei der *With-Operator*, der auch als => dargestellt wird. Dieser Operator erlaubt es, ein Objekt weiter zu verwenden, ohne dieses einer Variable zuweisen zu müssen. Normale Form:

```
val list = new LinkedList<Integer>()
list.add(1)
list.add(2)
```

Mit With-Operator:

```
new LinkedList<Integer>() => [
  it.add(1)
  it.add(2)
]
```

Das *it* fungiert dabei als die Bezeichnung für das Objekt, das zur Zeit verwendet wird. In diesem Beispiel stellt es also die Liste dar. Es gibt viele solche Extension-Methoden. Manche dieser Methoden erweitern aber auch nur bestimmte Klassen. Zum Beispiel erweitert die Methode *reverseView* nur Klassen, die das Interface *Iterable* implementieren. Einer der weiteren Vorzüge

von Xtend ist die Möglichkeit neben der objektorientierter Programmierung auch die funktionale Programmierung zu nutzen, zum Beispiel durch das Verwenden von Lambda-Ausdrücken. Dazu benutzen Methoden, wie *filter* oder *forEach*, Funktionen als Parameter, die auf den einzelnen Elementen ausgeführt werden. Zum Beispiel gibt der Aufruf

```
(1..99).filter[ i | i % 2 == 0].forEach[ i | println(i)]
```

alle geraden Zahlen zwischen 1 und 99 auf der Konsole aus. Dafür wendet die Methode *filter* zuerst die Funktion `i % 2 == 0` auf die Zahlen von 1 bis 99 an. Dann wendet die Methode *forEach* die Funktion `println(i)` auf die Elementen des Ergebnisses an, um die Zahlen auszugeben. Zum Schluss bietet Xtend zusätzlich die Möglichkeit *Template-Ausdrücke* zu verwenden, was das Erstellen von Quellcode, wie zum Beispiel HTML, enorm erleichtert. Dabei lassen sich auch die Klassen und Methoden innerhalb des Templates weiterverwenden. Diese werden mit « vor dem Aufruf und » nach dem Aufruf gekennzeichnet.

```
def someHTML(String content) '''
  <html>
    <body>
      «content»
    </body>
  </html>
'''
```

Durch diese Vorzüge ist Xtend ideal dafür geeignet, die Modell-zu-Modell-Transformationen zu definieren.

2.2 KIELER

KIELER[4], also *Kiel Integrated Environment for Layout Eclipse RichClient*, ist ein Forschungsprojekt, dass sich mit der Verbesserung vom graphischen modellbasierten Design von komplexen Systemen beschäftigt. Die Idee, die dem Projekt zugrunde liegt, ist, dass auf alle Komponenten der graphischen Darstellung in der Modellierungsumgebung ein automatisches Layout angewendet wird. Dadurch werden neue Möglichkeiten für das Bearbeiten, das Durchsuchen und das dynamische Visualisieren der Darstellung eröffnet. Aus diesem Grund liegt der Schwerpunkt dieses Projektes auf den *Pragmatiken* des modellbasierten Systemdesigns. Durch sie ist es möglich, die Verständlichkeit der Darstellungen, die Entwicklungs- und Wartungszeit und die Analyse des dynamischen Verhaltens zu verbessern.

[4]http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview

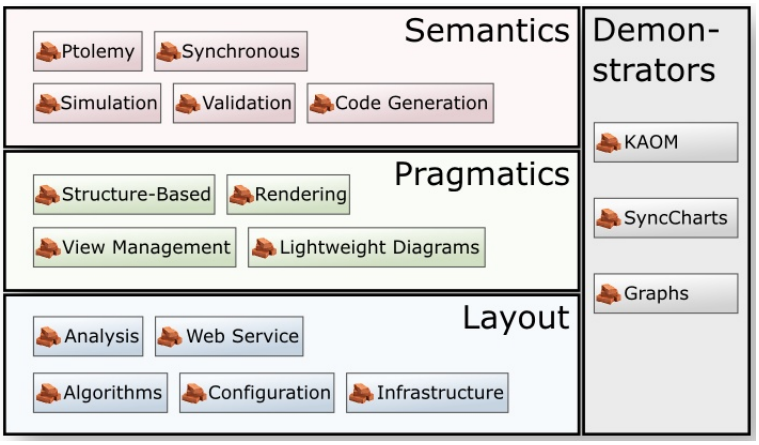

Abbildung 10. Die Teilbereiche des KIELER-Projekts

Das KIELER-Projekt besteht aus vier Bereichen (siehe Abbildung 10). Der Bereich der *Demonstrators* stellt Editoren zur Verfügung, mit den die Technologien der anderen Bereiche getestet werden können. Der Bereich der *Semantics* bietet eine Infrastruktur, mit der es möglich ist, die Semantik der Ausführung eines Metamodells zu definieren, um gewünschte Abläufe zu simulieren. Der *Layout*-Bereich beschäftigt sich mit der graphischen Darstellung an sich, zum Beispiel dem Verteilen der Knoten, um die Anzahl der Kanten zu minimieren. Zum Schluss ist da noch der Bereich der *Pragmatics*. Dieser Bereich beschäftigt sich mit der täglichen Arbeit der Modellentwickler im Bezug auf modellgetriebene Entwicklung. Darunter zählen das Erstellen und Bearbeiten von Modellen, Verbinden der Modelle mit verschiedenen Ansichten, um bestimmte Informationen zu visualisieren, und vieles mehr. Diese vier Bereiche ermöglichen dem Modellentwickler, seine Arbeit enorm zu vereinfachen.

2.2.1 KLighD

KLighD[5], also *KIELER Lightweight Diagrams*, ist ein Projekt, das in dem Pragmatics-Bereich von KIELER angesiedelt ist. Das Projekt beschäftigt sich mit leichter vorübergehender Darstellung von Modellen oder Teilen davon, ohne komplexe Bearbeitungswerkzeuge wie graphische Editoren einzubeziehen. Dafür bietet KlighD eine Infrastruktur zum Erstellen und Aktualisieren von dynamischen Ansichten verschiedener Datenmodelle auf einer abstrakten Ebene. Zusätzlich ermöglicht KLighD die Interaktion mit diesen Ansichten. Um diese Infrastruktur zur Verfügung zu stellen, wird das Problem auf eine dynamische Kombi-

[5]http://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=328115

nation von den Modell-zu-Modell-Transformationen und der Betrachter der Datenmodelle abstrahiert. Dabei wird für die Darstellung eines Datenmodells eine Ansicht erzeugt, die anhand der entsprechenden Transformation gefüllt wird. Unter diesen Modelltransformationen sind Transformationen gemeint, die aus einem Datenmodell ein graphisches Modell erzeugen. Die einzelnen Transformationen für die verschiedenen Typen der Datenmodelle müssen von der im KLighD-Projekt zur Verfügung gestellten *AbstractTransformation* abgeleitet sein und als Extension an dem entsprechenden Extension-Point registriert werden. Dadurch ist es möglich, verschiedene Ansichten von Datenmodellen zu erzeugen, ohne jedes mal eine weitere Ansicht implementieren zu müssen.

2.2.2 KGraph

KGraph[6], auch *KIELER Graph*, ist die zentrale Datenstruktur zur Darstellung von Graphen, die in KIELER verwendet wird. Diese Struktur wurde unter Verwendung von EMF erzeugt und besitzt daher ebenfalls alle Fähigkeiten der EMF-Modelle. Somit können die einzelnen Instanzen eines KGraphen

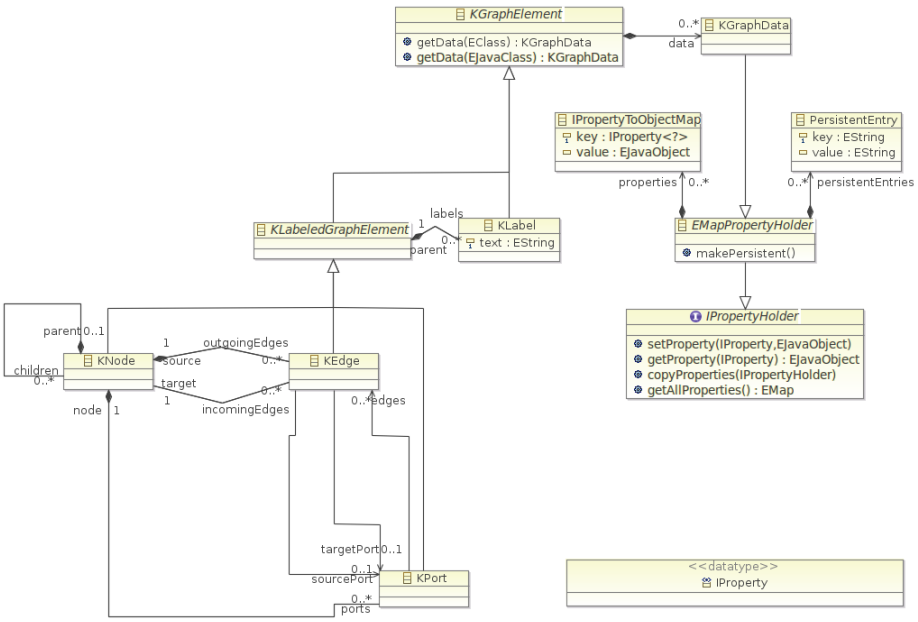

Abbildung 11. Das Metamodell des KGraphs

[6]http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KGraph+Meta+Model

mithilfe eines XML-Formats gespeichert, transformiert und validiert werden. Der KGraph wird in den verschiedenen Projekten von KIELER für verschiedene Zwecke verwendet. In *KIML*, *KIELER Infrastructure for Meta Layout*, wird KGraph als eine Datenstruktur verwendet, über die Layoutinformationen zwischen den graphischen Darstellungen und den Graph-Layout-Algorithmen ausgetauscht wird. In *KWebS*, *KIELER Web Services*, wird KGraph als ein Dateiformat für Graphen verwendet. Und letztendlich wird KGraph als die Basisstruktur für die darstellenden Modelle in KLighD verwendet.

Die Verwendung in KLighD steht in diesem Fall im Vordergrund. Hierbei wird jeder Graph durch einen Knoten, *KNode*, auf der höchsten Ebene dargestellt. Das bedeutet, dieser hat keinen Elternknoten, in dem er sich befindet. Dieser Knoten beinhaltet alle weiteren Knoten, die die Teile des Graphen repräsentieren. Hierbei kann jeder Knoten weitere enthalten. Dadurch kann ein verschachtelter Untergraph dargestellt werden. Der Graph kann auch Kanten, *KEdges*, enthalten, die verschiedene Knoten miteinander verbinden. Dabei haben die Kanten einen Bezug zu dem Quell- und dem Zielknoten. Im Gegensatz dazu haben die Knoten Bezug auf die von ihm ausgehende und eingehende Kanten. Zusätzlich besteht die Möglichkeit Anschlüsse, *KPorts*, zu den Knoten hinzuzufügen. Diese dienen als Anschlüsse für die Kanten und ermöglichen zum Beispiel eine Reihenfolge festzulegen. Die einzelnen Ports an einem Knoten und die Kanten, die von diesem Knoten ausgehen, werden in dem Knoten gespeichert. Diese drei Elemente können auch noch Beschriftungen, *KLabels*, erhalten, die in den meisten Fällen textuell sind. Hauptsächlich mithilfe dieser vier Elemente werden die einzelnen Graphen erzeugt, die die Datenmodelle repräsentieren.

3 Konzepte zur Visualisierung von Modelltransformationen

In diesem Kapitel werden die verschiedenen Konzepte vorgestellt, die die graphische Zuordung von Elementen einer Modelltransformation ermöglichen sollen. Zuerst ist es jedoch wichtig zu erläutern, wie das Projekt grundsätzlich funktionieren soll.

Die grundsätzliche Idee dabei besteht darin, eine Erweiterung für Eclipse zu erstellen, welche die Funktionalität bietet, verschiedene Modell-zu-Modell-Transformationen graphisch darzustellen. Dabei soll diese Erweiterung in Form von Plug-ins erzeugt werden, damit diese ohne großen Aufwand zu anderen bereits bestehenden Projekten hinzugefügt werden kann. Im Weiteren soll diese Erweiterung mit Transformationen zwischen beliebigen EMF-basierten Modellen funktionieren, ohne die Modelle oder die Erweiterung ergänzen zu müssen. Also darf die Erweiterung weder von Typen noch von Namen oder sonstigen Attributen der Modelle abhängig sein. Die Modelltransformationen, die dabei verwendet werden, sollen Xtend als Transformationssprache verwenden. Diese Programmiersprache bietet einige Vorzüge und eignet sich daher bestens für die Erstellung der Transformationen zwischen Modellen. Auch die Transformationen der einzelnen Modelle zu ihren graphischen Darstellungen sollen mittels Xtend realisiert sein. Diese müssen als *modelTransformations*-Extensions bei KLighD regestriert werden. Auf diese Weise kann KLighD für jedes Modell die passende Transformation aus den Extensions auswählen und auf das Modell anwenden. Für die Visualisierung der einzelnen Modelltransformationen soll ebenfalls KLighD verwendet werden. Dadurch wird dieser Prozess enorm vereinfacht, indem das Erstellen einer Ansicht und das Hinzufügen des Graphen zu dieser Ansicht von KLighD automatisch durchgeführt wird. Daraus folgt, dass der Anwender für die Umsetzung dieser Idee, einiges zur Verfügung stellen muss. Darunter fallen die Metamodelle, welche die einzelnen Modellinstanzen beschreiben, und die den Metamodellen entsprechende Modelltransformationen, sowohl zwischen einzelnen Modellen, als auch zwischen Modellen und ihren graphischen Darstellungen.

Als Nächstes ist es nun sinnvoll, sich die verschiedenen Ansätze anzuschauen, die das Visualisieren der Modelltransformationen umsetzen sollten. Dafür werden zuerst nochmal die Schwierigkeiten behandelt, die bei der Zuordnung der Elemente einer Transformation entstehen. Danach werden die verschiedenen Lösungsansätze und die Probleme, woran diese scheiterten, vorgestellt. Zum Schluss wird dann der Lösungsansatz erklärt, der die gewünschte graphische Zuordnung ermöglicht und die gegebenen Voraussetzungen erfüllt.

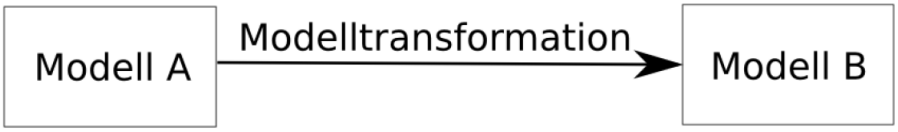

Abbildung 12. Transformation zwischen den einzelnen Modellen

3.1 Probleme

Bei der graphischen Zuordnung von Elementen einer Modelltransformation unter Verwendung der obengenannten Voraussetzungen ergeben sich einige Probleme.

Zuerst einmal müssen die Beziehungen der einzelnen Elemente bei der Transformation der Modelle gespeichert werden (siehe Abbildung 12). Da die Visualisierung der Modelltransformation nicht unbedingt im gleichen Moment wie die Transformation erfolgt, sondern möglicherweise erst im Nachhinein, ist es notwendig, die Beziehungen der Elemente persistent zu speichern. Dafür gibt es mehrere Lösungen. Doch weil die einzelnen Modelle anhand der entsprechenden Metamodelle erstellt werden und diese nicht verändert werden sollen, ist es notwendig, die Beziehungen der Elemente in einer zusätzlichen Datei zu speichern. Dazu ist es wichtig zu verstehen, wie die Beziehungen angelegt werden sollen. Dabei können die Modellelemente eine eins-zu-eins- oder eine eins-zu-viele-Beziehung eingehen. Aus diesem Grund ist es wichtig, genau zu überlegen, was für eine Struktur erzeugt werden soll, um die Beziehungen der einzelnen Modellelemente zu speichern. Bei einer Struktur, die nur das Speichern einer eins-zu-eins-Beziehung ermöglicht, erhöht sich das Volumen des benötigten Speicherplatzes, da für das Speichern einer eins-zu-viele-Beziehung mehrere Einträge erzeugt werden müssen. Im Gegensatz dazu erschwert sich bei einer Struktur zum Speichern von eins-zu-viele-Beziehungen der Zugriff auf die einzelnen Elemente einer Beziehung.

Als Nächstes ist es für die Visualisierung der Modelltransformationen wichtig, die einzelnen Elemente der Modelle zu den entsprechenden Elementen ihrer graphischen Darstellung zuzuordnen (siehe Abbildung 13). Dafür müssen auch die Beziehungen zwischen den Elementen der Modelle und den Elementen ihrer graphischen Darstellung gespeichert werden. Im Gegensatz zu Beziehungen zwischen den einzelnen Modellelementen besteht hier die Möglichkeit, die Beziehungen der Elemente bei der Visualisierung der Modelle nicht in einer externen Datei zu speichern. Das wird dadurch ermöglicht, dass die visuellen Darstellungen der Modelle keine eigenständigen Dateien bilden, sondern falls erwünscht neu erzeugt werden. Aus diesem Grund müssen die

Abbildung 13. Transformation zwischen dem Modell und der graphischen Darstellung

Beziehungen der Elemente bei der Visualisierung nur eine bestimmte Zeit gemerkt werden. Dieses Problem kann ebenfalls mit verschiedenen Mitteln gelöst werden. Zum Beispiel bietet KLighD eine Funktionalität, die es erlaubt, die Beziehungen der Elemente während der Visualisierung zu merken. Falls diese Beziehungen doch persistent gespeichert werden, ist es notwendig diese bei jeder Visualisierung neu anzulegen, da jedes mal neue graphische Darstellungen und neue Beziehungen erzeugt werden.

Zum Schluss ist es notwendig, die Beziehungen der Elemente der Modelltransformation auszulesen und den Modellelementen entsprechende graphische Elemente miteinander visuell zu verbinden. Dabei muss darauf geachtet werden, dass auch Kanten eines Graphen Quell- oder Zielelemente sein können. Da eine Kante nicht als Quell- oder Zielelement einer anderen Kante dienen kann, muss der Graph insofern erweitert werden, dass ein Zwischenknoten in die Kante eingefügt wird, der als Quell- oder Zielelement dient.

3.2 Lösungsansätze

Nun ist es an der Zeit, sich die verschiedenen Lösungsansätze anzuschauen, die für die Umsetzung der graphischen Zuordnung der Elemente einer Modelltransformation ausprobiert wurden.

Als Erstes entstand der Gedanke, eine Java-Klasse zu erzeugen, deren Instanz als Speichermedium für die einzelnen Beziehungen benutzt wird. Dabei entstand die Klasse `SaveTable`. Diese beinhaltet drei unabhängige `Hashtables`, die dafür benutzt werden sollen, die Beziehungen zwischen den Elementen des Quell- und des Zielmodells, zwischen den Elementen des Quellmodells und der entsprechenden graphischen Darstellung und zwischen den Elementen des Zielmodells und den entsprechenden graphischen Elementen zu speichern (siehe Abbildung 14). Die einzelnen `Hashtables` sollten dabei Elemente des Typs `Object` als Schlüssel und auch als Werte verwenden. Um die Möglichkeit einer eins-zu-viele-Beziehung zu gewährleisten, wobei keine Kollisionen beim

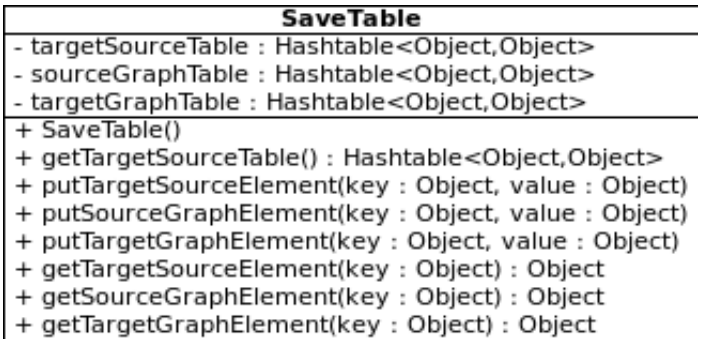

Abbildung 14. Die SaveTable-Klasse

Speichern in die `Hashtable` entstehen, werden beim Speichern der Beziehungen zwischen den Modellelementen die Elemente des Zielmodells als Schlüssel verwendet. Auf diese Weise bleiben die Schlüssel einzigartig, auch wenn die Werte der entsprechenden Einträge gleich sind. Dieser Ansatz wurde jedoch aufgrund von Problemen verworfen. Das hauptsächliche Problem bestand in dem von EMF generierten Quellcode, der für das Erzeugen und Bearbeiten des Modells und der einzelnen Modellelemente benötigt wird. In diesem Quellcode werden die Modellelemente durch Schnittstellen realisiert, die in anderen Klassen implementiert werden. Deswegen können die Schnittstellen nicht instanziiert werden und repräsentieren nicht die Modellelemente. Durch diese Aufteilung entstand das Problem, dass beim Speichern der Elemente eines Modells nur die von EMF generierten Schnittstellen gespeichert wurden. Durch sie konnten lediglich die Typen der einzelnen Elemente einer Beziehung zugeordnet werden. Die Elemente selbst konnten dadurch nicht zugeordnet werden.

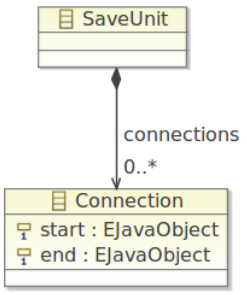

Abbildung 15. Das Metamodell des ersten EMF-Ansatzes

Als Nächstes entstand die Idee ein EMF-basiertes Modell als Speichereinheit zu verwenden, um die Beziehungen der Modellelemente zu speichern. Dabei sollte für jede Beziehung zwischen zwei Elementen ein neues `Connection`-Objekt innerhalb des Modells erzeugt werden, das die Beziehung repräsentiert. Dabei sollten die einzelnen Modellelemente als Attribute von Typ `Object` in den `Connections` abgelegt werden (siehe Abbildung 15). Dadurch entfiel das Problem mit den Kollisionen. Die Modellelemente wurden in diesem Ansatz nicht in einer Tabelle, sondern einzeln gespeichert. Dadurch wurde es möglich, Beziehungen zu speichern, die sowohl gleiche Quellmodellelemente, als auch gleiche Zielmodellelemente besitzen. Leider bestand bei diesem Ansatz wie zuvor das Problem, dass beim Speichern der Elemente nur die Schnittstellen der Modellelemente gespeichert wurden. Somit wurde auch dieser Ansatz verworfen, wobei er der Lösung ziemlich nah war. Aus diesem Ansatz entstand auch die Idee, die alle Voraussetzungen erfüllte und die im Folgenden beschrieben wird.

3.3 Eigener Lösungsweg

Letztendlich entstand das Konzept, welches die graphische Zuordnung der Elemente einer in Xtend geschriebenen Modelltransformation ermöglichte. Dabei wird ebenfalls ein EMF-basiertes Modell als Speichereinheit für die Beziehungen der einzelnen Elemente verwendet. Im Gegensatz zu den anderen Ansätzen werden bei diesem Modell jedoch nicht die einzelnen Modellelemen-

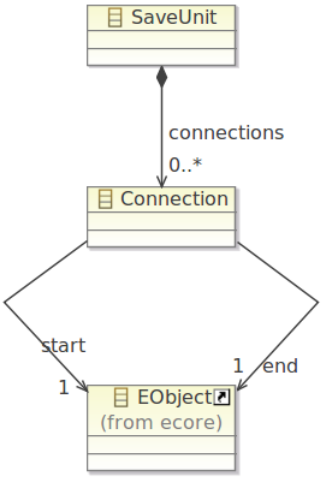

Abbildung 16. Das SaveUnit-Metamodell

te, sondern Referenzen zu diesen Modellelementen gespeichert. Dafür bietet EMF die Funktionalität, welche das Referenzieren beliebiger Modellelemente erlaubt. Diese können auch Teil eines externen Modells sein. Auf diese Weise ermöglicht EMF, dass die Beziehungen der Speichereinheit Modellelemente referenzieren, die sich in externen Dateien befinden. Für das Verwenden beliebiger Modelle verweisen diese Referenzen auf das entsprechende `EObject` in den Modellen (siehe Abbildung 16).

Wie in dem Ansatz davor wird für jede Beziehung zwischen zwei Elementen einer Modelltransformation ein `Connection`-Objekt angelegt. Diesem Objekt werden dann zwei Referenzen, `start` und `end`, zugewiesen. Jede von diesen Referenzen verweist dabei auf ein `EObject`, also ein beliebiges EMF-basiertes Modellelement, welches in einer externen Datei gespeichert ist. Auf diese Weise lassen sich die einzelnen Modellelemente eindeutig zuordnen. Zusätzlich besteht wie in dem Ansatz zuvor kein Problem, mehrere Beziehungen mit dem gleichen Quell- oder Zielelement zu speichern, weil die Beziehungen einzelne `Connection`-Elemente bilden. Dabei wird in Kauf genommen, dass für das Bilden einer eins-zu-viele-Beziehung manche Modellelemente mehrmals referenziert werden, da dadurch die Leistung beim Erstellen der Beziehungen und dem Auslesen einzelner Elemente erhöht wird.

Für das Zuweisen der einzelnen graphischen Elemente den entsprechenden Modellelementen wird bei diesem Ansatz die von KLighD bereitgestellte Funktion `putToLookUpWith()` verwendet. Diese legt während der Transformation eines Modells zu seiner graphischen Darstellung eine `MultiMap` an, in der die Beziehungen der Modellelemente zu den graphischen Elementen gespeichert werden. Diese `MultiMap` wird jedoch nicht in einer externen Datei gespeichert und existiert nur in dem durch die Visualisierung entstandenen Kontext. Da bei jeder graphischen Zuordnung der Elemente einer Modelltransformation die Visualisierung erneut vorgenommen wird, ist diese Funktion für den Zweck ausreichend. Als Nächstes ist es nun sinnvoll, sich anzuschauen, wie das Konzept im Einzelnen umgesetzt wurde.

4 Implementierung

In diesem Kapitel wird die Umsetzung der graphischen Zuordnung der Elemente einer Modelltransformation genauer beschrieben. Dabei wird die entstandene Eclipse-Erweiterung sowohl aus der Sicht des Programmierers als auch aus der Sicht des Anwenders betrachtet. Zuerst wird dafür die Struktur und der Aufbau der Erweiterung erklärt. Dann wird anhand von Beispielen erläutert, wie diese Erweiterung zu verwenden ist. Und zum Schluss werden die Vor- und Nachteile dieser Erweiterung dargestellt.

4.1 Aufbau und Struktur

Bei der Umsetzung des oben vorgestellten Konzepts zur graphischen Zuordnung der Elemente einer Modelltransformation entstanden die folgenden zwei Plug-ins:

- `de.cau.cs.rtprak.sna.saveunit`
- `de.cau.cs.rtprak.sna.transformationgraph`

Das `saveunit`-Plug-in enthält das EMF-Modell und die daraus generierten Klassen und Schnittstellen, die für das Erstellen und Bearbeiten der Speichereinheit `SaveUnit` notwendig sind. Das `transformationgraph`-Plug-in enthält neben dem `Activator` für das Plug-in drei weitere Dateien. Erste Datei ist die Klasse `TransformationModel`. Sie wird dafür verwendet, die einzelnen Modelle und die Speichereinheit zu speichern und an KLighD weiterzugeben. Auf diese Weise ist es möglich, die für die Modelltransformation notwendigen Daten an KLighD zu übergeben, ohne KLighD zu erweitern. Mit diesen Daten als Parameter ruft KLighD anschließend die entsprechende Modelltransformation auf und stellt den in der Transformation erzeugten Graph in einer Ansicht dar. Das `TransformationModel` enthält folgende Attribute mit den zugehörigen `get`- und `set`-Methoden:

- `private EObject source;`
- `private EObject target;`
- `private SaveUnit unit;`

Die Attribute `source` und `target` können beliebige EMF-basierte Modelle enthalten. Das Attribut `unit` enthält die Speichereinheit vom Typ `SaveUnit`. Diese drei Attribute werden im Weiteren dazu benutzt, die einzelnen Elemente der Transformation graphisch zuzuordnen. Eine weitere Klasse dieses Plug-ins ist `TransformationGraphHandler`. Diese Java-Klasse wird aufgerufen, falls eine graphische Zuordnung der Elemente einer Modelltransformation erfolgen soll. Dafür lädt die Klasse die einzelnen Modelle und die Speichereinheit, packt diese in das `TransformationModel` und ruft die entsprechende Transformation

auf, die die graphische Zuordnung von Elementen einer Modelltransformation vornimmt. Diese Transformation ist in der letzten Datei dieses Plug-ins enthalten, `TransformationGraph.xtend`. Diese Datei sorgt dafür, dass KLighD die Transformationen der beiden Modelle zu ihren graphischen Darstellungen ausführt, packt die daraus entstandene Graphen zusammen und verbindet die einzelnen Elemente anhand der `SaveUnit` mit Pfeilen. Bei der Verbindung zweier Elemente wird darauf geachtet, dass weder das Quell- noch das Zielelement eines Pfeils Kanten sind. In dem Fall, dass eines der Elemente eine Kante ist, wird automatisch ein Zwischenknoten erzeugt und in diese Kante eingefügt. Dieser Zwischenknoten dient dann als Quelle bzw. Ziel für den genannten Pfeil.

Für den Anwender dieser Eclipse-Erweiterung sind nur wenige Dinge wichtig. Zuerst muss der Anwender für die Visualisierung einzelner Modelle KLighD verwenden. Das heißt, dass die Transformationen der einzelnen Modelle zu ihren graphischen Darstellungen als `modelTransformations`-Extensions bei KLighD registriert sind. In diesen Transformationen muss beim Erstellen eines graphischen Elements die Beziehung zwischen diesem und dem ursprünglichen Modellelement durch die Funktion `putToLookUpWith()` hergestellt werden. Auf diese Weise ist es möglich, nachzuvollziehen, welches Modellelement vom graphischen Element repräsentiert wird.

Für die Verwendung der `SaveUnit` sind folgende Funktionen wichtig:

1. Für das Erzeugen einer neuen `SaveUnit`-Instanz:
   `SaveunitFactory.eINSTANCE.createSaveUnit()`

2. Für das Speichern einer Elementbeziehung in der `SaveUnit`-Instanz:
   `connect(EObject source, EObject target)`

Als Nächstes muss vor der Transformation eines Modells eine neue Instanz von `SaveUnit` erzeugt werden. Diese Instanz wird mit den für die Transformation notwendigen Daten an die Transformation übergeben. In der Transformation müssen die Beziehungen der Modellelemente, die im Nachhinein einander zugeordnet werden sollen, in der `SaveUnit`-Instanz gespeichert. Dafür muss für jedes Elementenpaar die Funktion `connect(EObject source, EObject target)` auf der `SaveUnit`-Instanz aufgerufen werden. Als ersten Parameter erhält diese Funktion das Quellmodellelement und als zweiten Parameter das Zielmodellelement. Auf diese Weise werden alle Beziehungen zwischen Modellelementen in der `SaveUnit`-Instanz gespeichert. Anschließend muss die `SaveUnit`-Instanz als eigenständige Datei gespeichert werden. Das Ergänzen der Modelltransformationen durch diese Befehle muss vom Anwender persönlich vorgenommen und kann nicht automatisiert werden, da die mit Xtend entwickelten Transformationen nicht standardisiert sind. Möglicherweise möchte der Anwender auch nicht alle, sondern nur bestimmte Beziehungen herstellen.

Zum Schluss sollte noch die Namenskonvention beachtet werden, bei der der Name des Zielmodells der Name des Quellmodells ergänzt mit einem Punkt und einer entsprechenden Endung ist. Der Name der `SaveUnit`-Datei entspricht dem Namen des Zielmodells ergänzt mit „`.saveunit`“. Zum Beispiel bei der Transformation eines Quellmodells mit dem Namen „`model.regexp`“ sollte ein Zielmodell mit dem Namen „`model.regexp.nfa`“ und eine Speichereinheit mit dem Namen „`model.regexp.nfa.saveunit`“ entstehen. Im Weiteren wird anhand von Beispielen erklärt, wie diese Erweiterung zu benutzen ist.

4.2 Beispiele

Um diese Eclipse-Erweiterung zu benutzen, muss zuerst beim Aufrufen einer Transformation eine Instanz von `SaveUnit` erstellt werden. Diese wird dann als Parameter an die eigentliche Transformation übergeben. In dem folgenden Beispiel wird zuerst eine `SaveUnit`-Instanz für die Transformation von einem regulären Ausdruck zu einem nichtdeterministischen endlichen Automaten erzeugt. Als nächstes wird eine Instanz der entsprechenden Transformation erstellt. Zum Schluss wird dann die Transformation aufgerufen, wobei sie den regulären Ausdruck und die `SaveUnit`-Instanz erhält und einen Automaten zurückliefert.

```
SaveUnit unit = SaveunitFactory.eINSTANCE.createSaveUnit();

Injector injector = Guice.createInjector();
TransformRegexpToNfa transformation = injector
                .getInstance(TransformRegexpToNfa.class);

Automaton autom = transformation
                .transformRegexpToNfa(regexp, unit);
```

Innnerhalb der Modelltransformation werden die einzelnen Beziehungen der Elemente in der Speichereinheit abgespeichert. Im folgenden Beispiel wird gezeigt, wie die Beziehung zwischen einem regulären Ausdruck `exp` und einem Zustand des entstandenen Automaten `state` in der `SaveUnit`-Instanz `unit` abgespeichert wird.

```
unit.connect(exp, state)
```

Das muss bei allen erzeugten Modellelementen in der Transformation vorgenommen werden, falls diese ihren Quellelementen zugeordnet werden sollen. Hierbei ist der erste Parameter das Quell- und der zweite Parameter das Zielelement. Ähnliches Verfahren muss auch bei den Transformationen der Modelle zu ihren graphischen Darstellungen angewandt werden. Hierbei wird auf jedem erzeugten Element die Funktion `putToLookUpWith()` mit dem

Quellelement als Parameter aufgerufen, um die Beziehung zwischen diesen Elementen herzustellen. Zum Beispiel wird für eine Transition des Modells eine neue Kante im Graphen erzeugt.

```
createEdge() => [
        it.putToLookUpWith(trans);
        ...
]
```

Sind diese Voraussetzungen und die Namenskonvention erfüllt und sind die Transformationen der einzelnen Modelle zu ihren graphischen Darstellungen als Extensions bei KLighD registriert, ist die visuelle Darstellung der Modelltransformation ganz einfach. Ein Klick mit der rechten Maustaste auf das Zielmodell zeigt die verschiedenen Menüeinträge. Unter diesen befindet sich der Eintrag "`Show Transformation Graph`". Mit einem Klick auf diesen Eintrag öffnet sich automatisch eine Ansicht, welche die gewünschte graphische Zuordnung der Elemente einer Modelltransformation anzeigt.

4.3 Vor- und Nachteile

Die oben vorgestellte Eclipse-Erweiterung für die graphische Zuordnung der Elemente einer Modelltransformation besitzt einige Vor- und Nachteile sowohl für den Programmierer, als auch für den Anwender. Diese werden im Folgenden beschrieben.

Einer der Vorteile der Erweiterung für den Programmierer ist der einfache Aufbau. Es ist ohne viel Aufwand möglich, die einzelnen Plug-ins zu erweitern, um bestimmte Funktionalitäten hinzuzufügen. Für den Anwender bietet die Erweiterung ebenfalls einige Vorteile. Die Erweiterung funktioniert auf beliebigen EMF-basierten Modellen, die mit Xtend transformiert werden. Dabei muss weder an den Modellen, noch an der Erweiterung etwas verändert werden.

Der hauptsächliche Nachteil dieser Erweiterung besteht darin, dass die einzelnen Transformationen durch einige Befehle ergänzt werden müssen. Dieser Prozess kann nicht automatisiert werden, da es beim Anwender liegt, die Beziehungen welcher Elemente er darstellen möchte. Zusätzlich ist es bis jetzt noch nicht möglich, auf Veränderungen des Quellmodells zu reagieren. Dabei muss erst die Modelltransformation erneut durchgeführt werden, um die Zuordnung zu aktualisieren.

5 Evaluation

In diesem Kapitel wird beschrieben, welchen Umfang die entstandene Eclipse-Erweiterung hat und wie diese getestet wurde.

Die im Rahmen dieser Arbeit entstandene Eclipse-Erweiterung ist recht übersichtlich. Sie besteht aus zwei Plug-ins. Das erste Plug-in besteht aus dem EMF-Modell der verwendeten Speicherstruktur und dem daraus generierten Quellcode, der für das Erstellen und Bearbeiten dieser Struktur benötigt wird. Das zweite Plug-in beinhaltet den `Acivator` für das Plug-in und drei weitere Dateien. Diese Dateien haben zusammen ungefähr 500 Zeilen Quellcode und Kommentare. Zusätzlich werden zum Ausführen der Erweiterung einige weitere Plug-ins verwendet. Darunter fallen hauptsächlich verschiedene Komponenten von KIELER, zum Beispiel KLighD oder KGraph. Das Aussehen der graphischen Darstellung der Modelltransformation hängt von dem Layout-Algorithmus ab, der dabei verwendet wird. Dieser Algorithmus wird in der `TransformationGraph.xtend`-Datei des `de.cau.cs.rtprak.cs.sna.transformationgraph`-Plug-ins bestimmt.

Als Nächstes werden der Umgang mit der Erweiterung und die verschie-

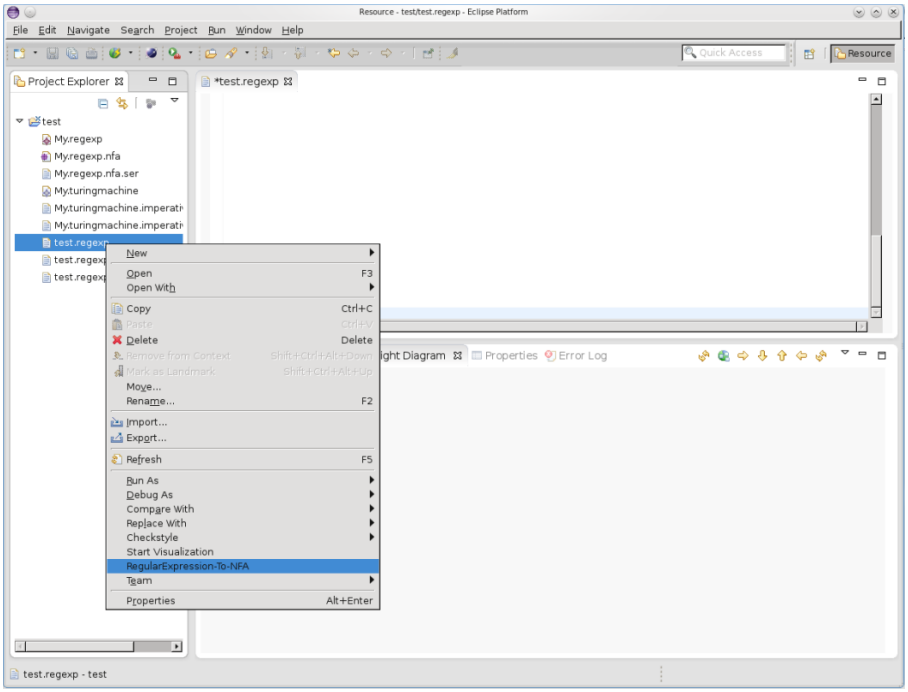

Abbildung 17. Auswählen der Modelltransformation

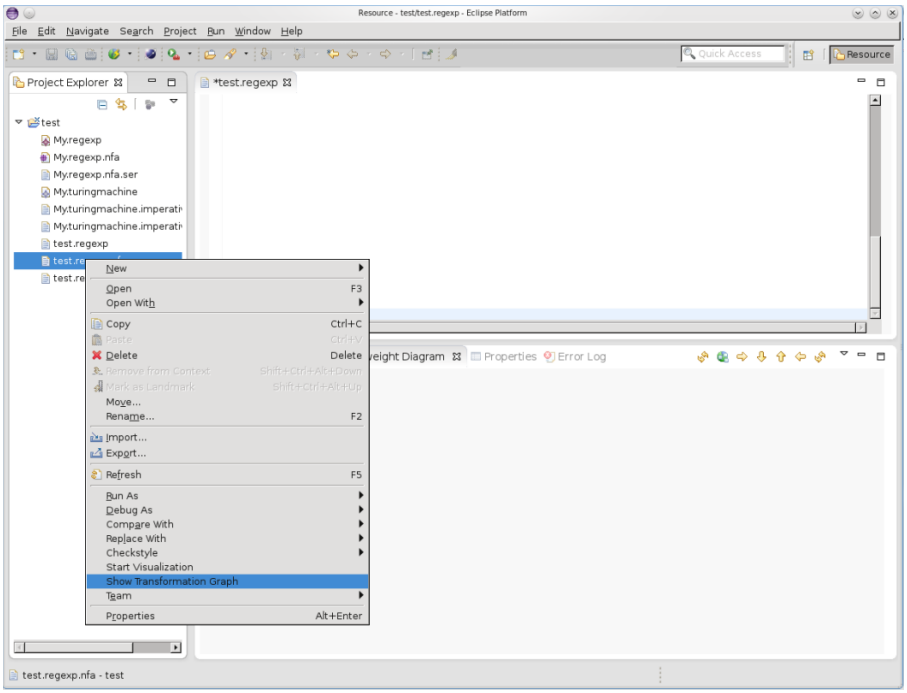

Abbildung 18. Auswählen der Visualisierung der Modelltransformation

denen Versuche beschrieben, die zum Testen der Erweiterung durchgeführt wurden.

5.1 Fallstudie

Der Umgang mit der Erweiterung ist sehr einfach. Nach dem Hinzufügen der notwendigen Plug-ins und dem Ergänzen der Modelltransformation durch die Funktion zum Erstellen der Beziehungen zwischen den Elementen sind nur wenige Klicks mit der Maus dazu notwendig, um die graphische Zuordnung der Elementen einer Modelltransformation anzuzeigen. Wie von mehreren Kommilitonen getestet und bestätigt, ist die Verwendung der Erweiterung intuitiv und braucht keine besondere Erklärung. Nach dem Erstellen und Speichern eines gültigen Quellmodells wird mit einem Rechtsklick auf der erstellten Datei ein Menü aufgerufen, worin sich der Transformationsbefehl, zum Beispiel „`RegularExpression-To-NFA`“, befindet (siehe Abbildung 17). Durch diesen Befehl wird das entsprechende Zielmodell und die entsprechende Speichereinheit erzeugt. Bei einem Rechtsklick auf das Zielmodell erscheint ein Menü, welches den Befehl „`Show Transformation Graph`“ enthält (siehe Abbil-

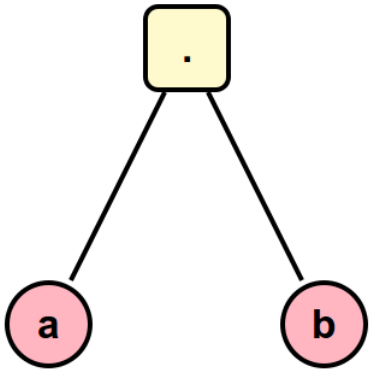

Abbildung 19. Visualisierung des Quellmodells

dung 18). Mit einem Klick auf diesen Befehl erscheint eine Ansicht, die die grafische Zuordnung der Elemente der entsprechenden Modelltransformation enthält.

5.2 Tests

Die in Verbindung mit dieser Arbeit entstandene Eclipse-Erweiterung wurde mithilfe von verschiedenen Modellen getestet. Dabei wurden Transformationen zwischen regulären Ausdrücken und nichtdeterministischen endlichen Automaten und zwischen Turingmaschinen und imperativen Programmen verwendet und graphisch dargestellt. Um die Erweiterung zu testen, wurden verschiedene Modellinstanzen der regulären Ausdrücke und Turingmaschinen erstellt und zu entsprechenden nichtdeterministischen endlichen Automaten und imperativen Programmen transformiert. Zum Beispiel wird der reguläre Ausdruck `a` konkateniert mit `b` zu dem entsprechenden nichtdeteministischen endlichen Automaten transformiert. Die graphischen Darstellungen dieser

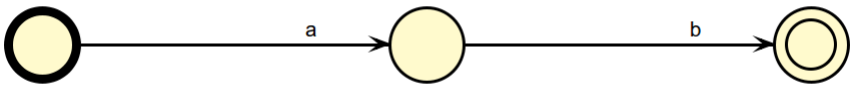

Abbildung 20. Visualisierung des Zielmodells

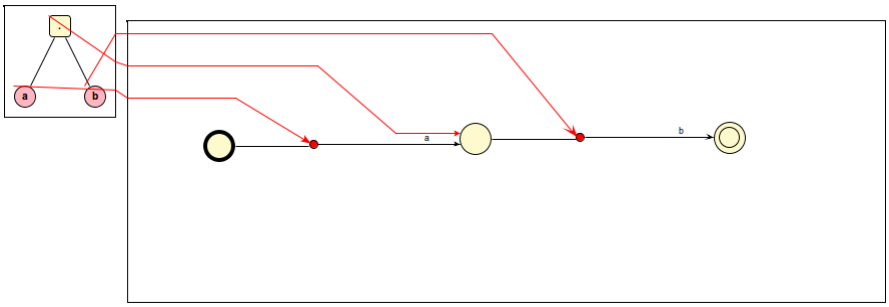

Abbildung 21. Visualisierung der Modelltransformation

Modelle sind in Abbildung 19 und Abbildung 20 zu sehen. Die entsprechende graphische Zuordnung der Elemente dieser Transformation ist in Abbildung 21 zu sehen.

6 Fazit und Ausblick

In dieser Arbeit wurde eine Eclipse-Erweiterung vorgestellt, die die graphische Zuordnung der Elemente einer Modelltransformation ermöglicht. Dabei kann die Erweiterung mit beliebigen EMF-basierten Modellen umgehen. Dafür muss lediglich die Transformation zwischen den Modellen ergänzt werden, um die Verbindung zwischen den einzelnen Elementen herzustellen. Zusätzlich müssen bei den Transformationen der Modelle zu ihren graphischen Darstellungen einige Aspekte bedacht werden, um die Voraussetzungen für die graphische Darstellung der Modelltransformationen zu erfüllen. Für die modellgetriebene Softwareentwicklung bietet diese Erweiterung eine gute Möglichkeit, bessere Übersicht über die transformierten Modelle und die Abstammung der einzelnen Modellelemente zu gewinnen. Dabei erleichtert sie auch die Fehleranalyse für die einzelnen Transformationen, indem sie die Transformationen graphisch darstellt. Im Weiteren könnte diese Eclipse-Erweiterung um die Möglichkeit der graphischen Zuordnung der Modellelemente zu den entsprechenden Metamodellelementen ergänzt werden (siehe Abbildung 22).

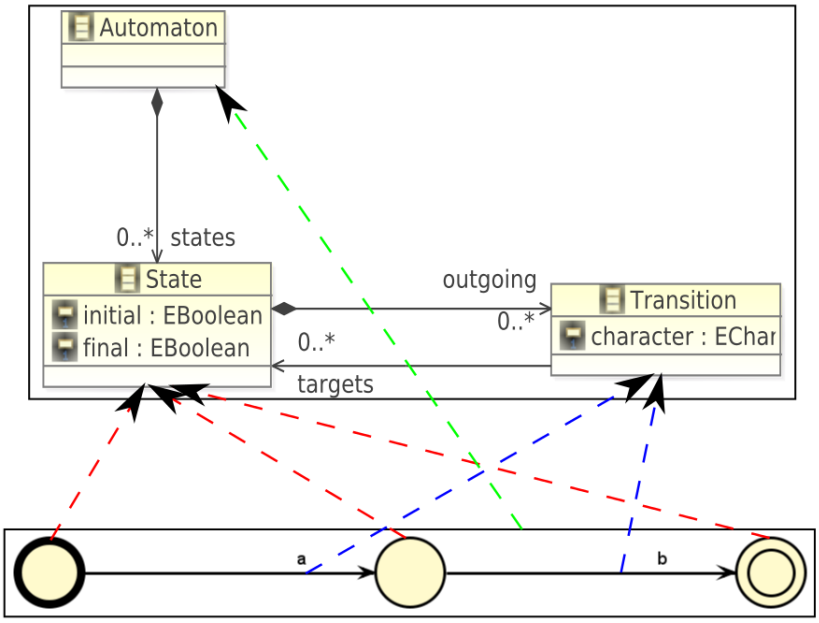

Abbildung 22. Zuordnung von Modellelementen zu Metamodellelementen

Zusätzlich wäre es nötig zu überlegen, ob die notwendige Ergänzung der Modelltransformationen durch die Befehle zur Speicherung der Elementbeziehungen nicht automatisiert werden kann. Auf diese Weise könnte der Umgang mit der Erweiterung enorm vereinfacht werden. Da die modellgetriebene Softwareentwicklung immer mehr an Bedeutung gewinnt, ist die in dieser Arbeit vorgestellte Eclipse-Erweiterung zur graphischen Zuordnung der Elemente einer Modelltransformation ein nützliches Werkzeug zur Verbesserung der Qualität und Analyse nach Fehlern.

Literatur

[1] Brockhaus-enzyklopädie online, 2005. Druckausgabe u.d.T.: Brockhaus. Enzyklopädie in 30 Bänden (2006).

[2] Vincent Aranega, Anne Etien, and Jean-Luc Dekeyser. Using an alternative trace for QVT. *Electronic Communications of the EASST*, 42, 2011.

[3] Nahum Gershon, Stephen G Eick, and Stuart Card. Information visualization. *interactions*, 5(2):9–15, 1998.

[4] Olivier Gruber, BJ Hargrave, Jeff McAffer, Pascal Rapicault, and Thomas Watson. The Eclipse 3.0 platform: adopting OSGi technology. *IBM Systems Journal*, 44(2):289–299, 2005.

[5] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 719–720. ACM, 2006.

[6] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[7] Object Technology International, Inc. Eclipse Platform Technical Overview. *Technical Report*, http://www.eclipse.org/whitepapers/eclipse-overview.pdf, 2003.

[8] B.A. Price, I.S. Small, and R.M. Baecker. A taxonomy of software visualization. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume ii, pages 597–606, 1992.

[9] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[10] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[11] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[12] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[13] Andrés Yie and Dennis Wagelaar. Advanced traceability for ATL. In *1st International Workshop on Model Transformation with ATL*, pages 78–87, 2009.