

Framework zum Tracing von EMF-Modelltransformationen

Alexander Schulz-Rosengarten

Bachelorarbeit

eingereicht im Jahr 2014

Christian-Albrechts-Universität zu Kiel

Institut für Informatik

Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

Betreut durch: Dipl.-Inf. Christian Motika

Eidesstattliche Erklärung

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

Kiel,

Zusammenfassung

Im Bereich der Softwareentwicklung spielt der modellbasierte Entwurf eine immer wichtigere Rolle. Er bietet die Möglichkeit komplexe Strukturen und Abläufe anwenderfreundlich aufzubereiten, die somit leicht zu erfassen sind. Hierbei sind insbesondere Modell-zu-Modell-Transformationen ein hilfreiches Mittel, um die Struktur eines Modells zu vereinfachen oder es in einen anderen Modelltyp zu überführen. So können beispielsweise abstrakte Elemente in einem Modell durch eine Transformation in ihre feingranulareren Einzelteile aufgespalten werden, was eine programmatische Weiterverarbeitung erleichtert. Dabei ist es auch von Interesse, die Beziehungen der entstandenen Elemente zu ihren Quellen durch ein Mapping zu erhalten, um Informationen auf das Quellmodell übertragen zu können. Dieses Prinzip wird im Kontext der Modelltransformationen als Tracing bezeichnet.

Zu diesem Zweck beschäftigt sich diese Arbeit mit der Erstellung eines Frameworks zum Tracing von Transformationen EMF-basierter Modelle. Es wird eine Schnittstelle zur Erfassung der Entstehungsbeziehung zwischen Elementen während der Transformation vorgestellt, sowie die dazugehörige Datenstruktur, die es ermöglicht, verschiedene Ketten von Modelltransformationen zu erfassen und Mappings zwischen beliebigen Modellen zu ermitteln.

Dieses Framework wird dann im Anwendungsfall der von von Hanxleden et al. entwickelten SCCharts erprobt. Dies ist eine graphische synchrone Programmiersprache, die eine Reihe von Modelltransformationen zur Kompilierung und konservativen Erweiterung des Sprachumfangs nutzt.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Modelltransformationen	1
1.2	Tracing	2
1.3	SCCharts in KIELER	3
1.3.1	Modelltransformationen	4
1.4	Problemstellung	6
1.5	Aufbau	7
2	Verwandte Arbeiten	9
2.1	Tracing	9
2.1.1	Tracing von Modelltransformationen	9
2.2	Spezifikation von Mappings	13
3	Verwendete Technologien	15
3.1	Eclipse Entwicklungsumgebung	15
3.1.1	Eclipse Modeling Framework	15
3.1.2	Xtend	17
3.1.3	Guava	17
3.2	Kiel Integrated Environment for Layout Eclipse Rich Client	18
3.2.1	KIELER Lightweight Diagrams	19
4	Entwurf des Tracing-Frameworks	21
4.1	Anforderungen	21
4.1.1	Anwendungsfälle	22
4.1.2	Schnittstelle	22
4.1.3	Datenstruktur	24
4.1.4	Datenaufbereitung	24
4.1.5	Visualisierung	25
4.2	Schnittstelle	25
4.3	Datenstruktur	27
4.3.1	Entwürfe	27
4.3.2	Resultat	30
4.4	Datenaufbereitung	32
4.5	Visualisierung	36

Inhaltsverzeichnis

5 Implementierung des Tracing-Frameworks	39
5.1 Struktur	39
5.1.1 Tracing Funktionen	39
5.1.2 Visualisierung	40
5.1.3 Tests	40
5.2 Schnittstelle zum Erfassen von Mappings	41
5.3 Transformationsbaum als Tracing-Struktur	42
5.3.1 Erweiterte Funktionalität des Transformationsbaumes	42
5.4 Visualisierung der Tracing-Informationen	44
6 Ergebnisse der Experimente und Evaluation	49
6.1 Einbettung der Schnittstelle für Mappings	49
6.1.1 Evaluation	51
6.2 Tracing der ABO Transformation	52
6.3 Performance	54
6.3.1 Zeitverbrauch durch Mappinggenerierung	54
6.3.2 Zeitverteilung bei Transformationsketten	55
7 Fazit	57
7.1 Zusammenfassung	57
7.2 Zukünftige Einsatzbereiche	58
7.2.1 Simulation von SCCharts	58
7.2.2 Interaktive Timing-Analyse	58
Bibliografie	61

Abbildungsverzeichnis

1.1	Die Meta-Object Facility Architektur [Sch11]	2
1.2	Übersicht der konkreten graphischen Syntax für SCCharts [Han+14]	3
1.3	Abhängigkeitsgraph der Ext. SCCharts Transformationen [Han+14]	4
1.4	Übersicht über Kompilierungskette von SCCharts [Mot+13]	5
2.1	Metamodell für Tracing-Modelle in Kermeta [FHN06]	11
3.1	Konzept der Eclipse Plug-in-Architektur [Fuh11]	16
3.2	Vereinfachter Auszug aus dem <i>Ecore</i> Meta-Metamodell [Mot09] . .	16
3.3	Struktureller Aufbau des KIELER Projekts	18
3.4	Schematischer Ablauf der Diagrammerstellung durch KLighD [SSH13]	19
4.1	Anwendungsfalldiagramm der Frameworks	23
4.2	Mappingbeziehungen zwischen Modellelementen	25
4.3	Aufspalten einer Eins-zu-Viele-Beziehung in eine Menge von einzelnen Relationen	26
4.4	Minimaler Entwurf für das Metamodell	27
4.5	Erweiterter Entwurf für das Metamodell	28
4.6	Transformationszentrierter Entwurf für das Metamodell	29
4.7	Metamodell der Datenstruktur	30
4.8	Beispiel für ein Modell des Metamodells	31
4.9	Möglichkeiten für Pfade durch einen Transformationsbaum	32
4.10	Beispiel für transitives Zusammenführen von Mappings	36
4.11	Konzeptionelle Darstellung der Visualisierung des Mappings	37
5.1	Auszug aus Methoden der TransformationMapping Klasse	41
5.2	Auszug aus Methoden der TransformationTreeExtensions Klasse . .	42
5.3	Diagramm eines Transformationsbaumes	45
5.4	Visualisierung enthaltener Modelle im Transformationsbaum	46
5.5	Mapping zwischen Modellen in vereinfachter Diagrammform	47
6.1	ABO SCChart	52
6.2	Transformationsbaum der ABO Transformation	53
6.3	Mapping zwischen ABO als SCChart und SCG	53
6.4	Verteilung des Zeitverbrauchs bei den ABO Transformationen	55
7.1	Phasen der Kompilierung und Fluss der Timing-Informationen [Fuh+14]	59

Abkürzungsverzeichnis

API	Application Programming Interface
ATL	Atlas Transformation Language
EMF	Eclipse Modeling Framework
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEM	KIELER Execution Manager
KIML	KIELER Infrastructure for Meta Layout
KLighD	KIELER Lightweight Diagrams
M2M	Modell-zu-Modell
MoC	Model of Computation
MOF	Meta-Object Facility
OMG	Object Management Group
QVT	Query View Transformation
RCP	Rich Client Platform
SCG	Sequentially Constructive Graph
VM	Virtual Machine
WCET	Worst-Case Execution Time

Einleitung

Der modellbasierte Entwurf ist ein Ansatz der Softwareentwicklung, der in vielen Bereichen Anwendung findet und über das bewährte System der UML-Diagramme hinausgeht. Er ermöglicht eine Abstraktion von den meist komplexen Einzelabläufen hin zu einer visualisierbaren Struktur, die intuitiv und somit leicht zu erfassen ist. Dennoch setzen sich die graphischen und modellbasierten Werkzeuge nur zögerlich in der Wirtschaft durch, wie in der Erhebung von Deifel et al. [Dei+99] beschrieben wird.

1.1. Modelltransformationen

Ein wichtiges Instrument bei der modellbasierten Entwicklung sind die Modelltransformationen. Sie ermöglichen eine kontrollierte und strukturierte Veränderung eines Modells unter Zuhilfenahme vordefinierter Regeln. Diese Regeln sind meist auf dem Metamodell des eigentlichen Modells definiert, damit sie allgemeingültig für alle Modelle dieses Typs sind.

Der Begriff Metamodell bezieht sich dabei auf die Meta-Object Facility (MOF) Architektur¹, ein Standard der Object Management Group (OMG), dargestellt in Abbildung 1.1. Sie unterteilt die Modellierung in vier Metaebenen. Die unterste M0-Ebene beschreibt die konkrete Instanziierung eines Modells in der realen Welt. Das Modell ist die konzeptionelle Beschreibung eines solchen realen Objekts, basierend auf den Formalismen, die im Metamodell, der M2-Ebene, für das Modell spezifiziert wurden. Die M3-Ebene ist die abstrakteste Form der Modellierung, sie bietet die Definitionsbasis für die Metamodellierung.

Modelltransformationen bieten die Möglichkeit Modelle zu vereinfachen, zu erweitern oder in eine andere Form umzuwandeln, um so ein Modell auch auf andere Domänen zu übertragen. Die Implementierung der Transformation kann in einer beliebigen Programmiersprache umgesetzt werden, die Zugriff auf das Modell hat, aber auch in einer speziellen Modelltransformationssprache.

Weiterhin kann man Modelltransformationen in zwei Kategorien unterteilen.

¹<http://www.omg.org/spec/MOF/>

1. Einleitung

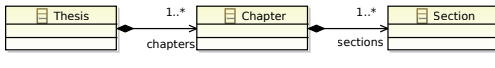

M3	Meta-Metamodell	MOF, Kermeta, KM3, Ecore
M2	Metamodell	UML, Petri-Netze, Xtext, DSLs
M1	Modell	
M0	Modellinstanzen	

Abbildung 1.1. Die Meta-Object Facility Architektur [Sch11]

Erstens die Modelltransformationen bei denen Text, beziehungsweise Code erzeugt wird und zweitens Modell-zu-Modell (M2M) Transformationen. Dies sind Modelltransformationen bei denen das Ergebnis erneut ein Modell ist. Hierzu gehören sowohl Transformationen, die einzelne Komponenten umwandeln, um ihre Komplexität zu reduzieren, als auch solche bei denen das Zielmodell auf einem anderen Metamodell basiert, um so das Quellmodell zu einer anderen Domäne kompatibel zu machen. Zudem wird in der Regel nicht nur eine einzelne Transformation durchgeführt, sondern ganze Ketten, mit denen ein Modell schrittweise in eine andere Struktur überführt wird.

Während der Modelltransformationen gehen meist die Informationen über die Beziehungen zwischen den Modellen und deren Elementen verloren. Dies erschwert die Verifikation und das Debugging, da Informationen nicht auf die Modellebene übertragen werden können, wie von Schmidt [Sch06] festgestellt. Um die entsprechenden Beziehungen während der Transformationen zu erhalten, bedarf es einer Ablaufverfolgung, dem Tracing, das Transformationsabläufe erfasst und rückverfolgen kann.

1.2. Tracing

Tracing ist im Kontext dieser Arbeit die Erfassung der Abfolgen von Modelltransformationen und deren Auswirkungen auf die Beziehungen zwischen Modellelementen. Kern der Tracings ist daher, die Erfassung des Verhältnisses von Quell- zu Zielmodell, also die Zuordnung der einzelnen Objekte des Quellmodells zu den daraus transformierten Objekten des Zielmodells.

Dies wird durch ein Mapping erreicht. Ein Mapping im Allgemeinen ist das Aufeinanderabbilden von Datenelementen aus zwei verschiedenen Datenmodellen. Man kann nun die Spezifikation eines Mappings, wie beispielsweise Lopes et al.

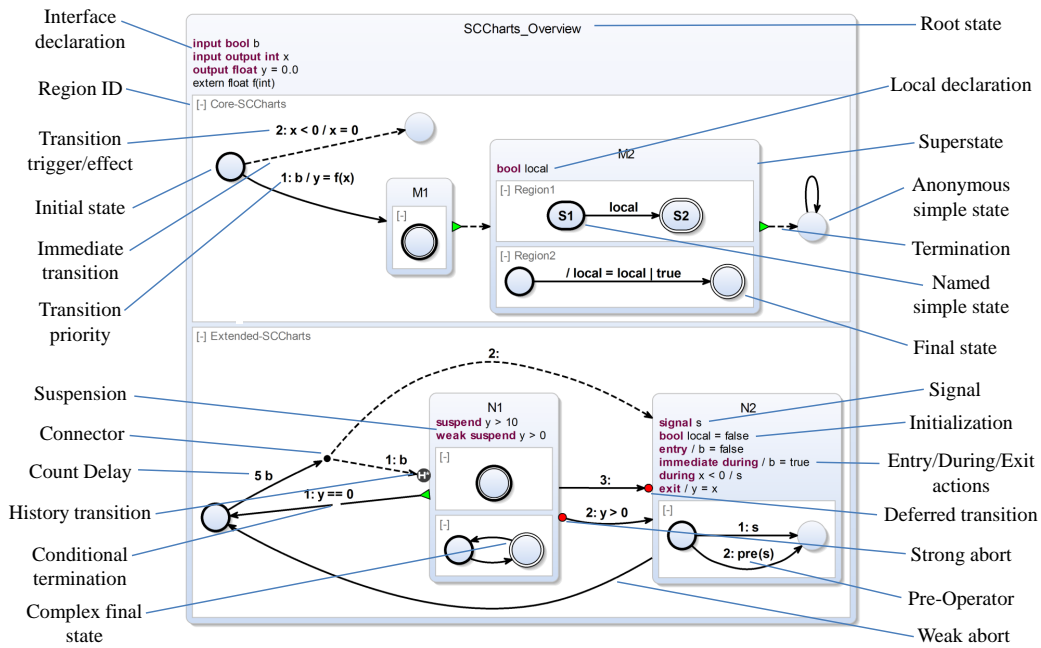


Abbildung 1.2. Übersicht der konkreten graphischen Syntax für SCCharts [Han+14]

[Lop+06] zeigt, an den Beziehungen der Elemente zweier Metamodelle festmachen. Diese Definition erschwert jedoch die Erstellung eines Mappings zwischen Modellen desselben Metamodells, wie es bei Transformationen zur modellinternen Abstraktion, also dem Vereinfachen oder Expandieren von Komponenten, der Fall ist. Aus diesem Grund wird hier ein Mapping als eine Menge von bidirektionalen Relationen zwischen konkreten Modellelementen eines beliebigen Metamodells definiert. Somit kann ein Mapping die Beziehungen zwischen den Elementen zweier beliebiger Modelle beschreiben. Jedoch ist dieses Mapping an eine konkrete Modelltransformation gebunden und kann nicht vom Metamodell auf alle möglichen Instanzen übertragen werden.

1.3. SCCharts in KIELER

Die von von Hanxleden et al. [Han+14] entwickelten Sequentially Constructive Charts (SCCharts) sind eine visuelle Programmiersprache für sequenziell konstruktive synchrone Programme. Sie sind angelehnt an das Konzept der Statecharts von Harel [Har87]. Bei SCCharts ist es möglich, komplexe Problemstellungen mithilfe sequenzieller, also intuitiver Programmierung beziehungsweise

1. Einleitung

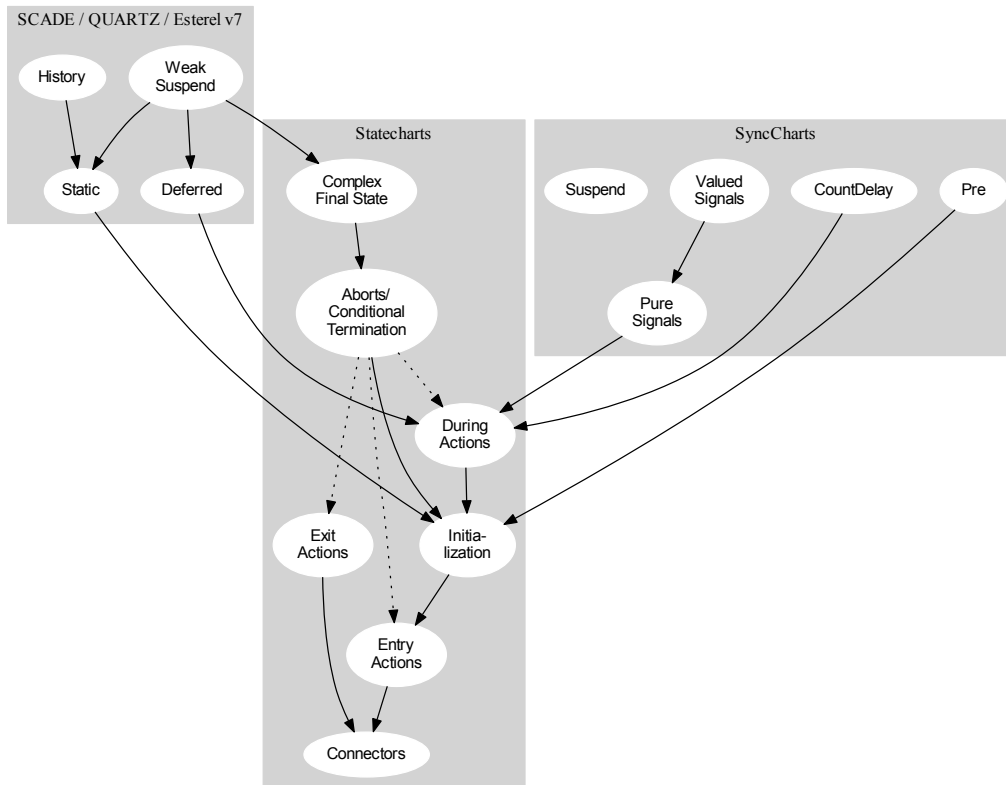


Abbildung 1.3. Abhängigkeitsgraph der Ext. SCCharts Transformationen [Han+14]

Modellierung zu lösen und dabei die Vorteile der deterministischen Nebenläufigkeit des synchronen Model of Computation (MoC) zu erhalten. Eine Referenzimplementierung der SCCharts wurde im Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) Projekt umgesetzt.

KIELER ist ein Forschungsprojekt, das von der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme entwickelt wird und sich mit der Entwicklung neuer Methoden für den graphischen modellbasierten Entwurf beschäftigt.

Bei der SCCharts Implementierung wurde das Eclipse Modeling Framework (EMF) genutzt, um die Modelle des SCCharts zu definieren. Abbildung 1.2 zeigt ein SCChart mit den einzelnen graphischen Syntaxelementen.

1.3.1. Modelltransformationen

Die semantische Definition der SCCharts basiert auf einer minimalen Menge von Sprach- beziehungsweise Modellkonstrukten, den Core SCCharts Elementen. Um dem Nutzer das Modellieren mit dieser Sprache zu erleichtern, gibt es zusätzlich weitere Elemente, die die Extended SCCharts bilden. In Abbildung 1.2 sind in

HW + SW Compiler Stack

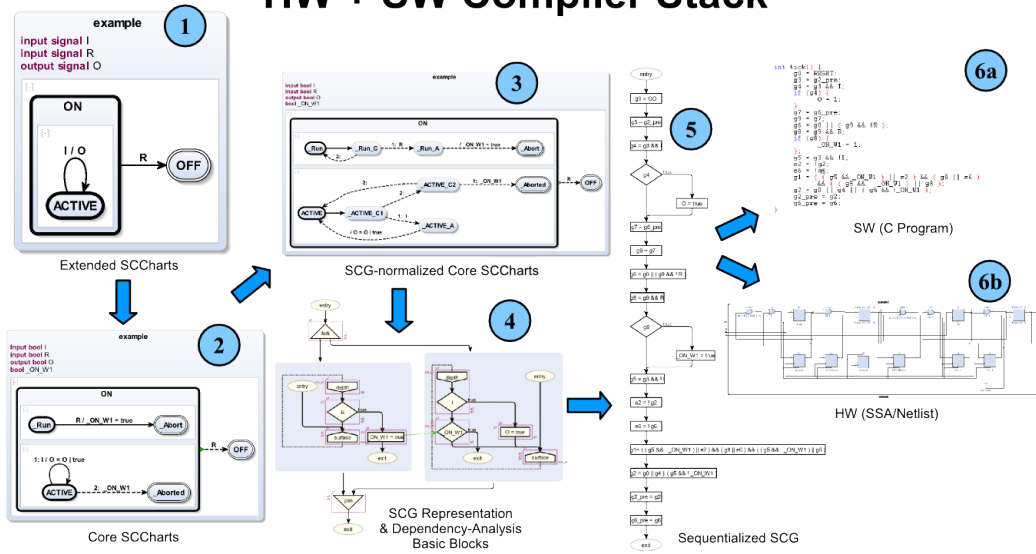


Abbildung 1.4. Übersicht über Kompilierungskette von SCCharts [Mot+13]

der oberen Region nur Core SCCharts Elemente darstellt und in der unteren Region auch die Extended SCCharts Konstrukte enthalten. Man bemerkt, dass auf diese Weise der Sprachumfang deutlich erhöht wurde und einzelne Ausdrücke auch kompliziertes Verhalten abstrakt beschreiben können. Dabei ist sichergestellt, dass diese Extended SCCharts immer in semantisch äquivalente Core SCCharts überführbar sind. Somit bilden diese Sprachkonstrukte eine Art syntaktischen Zucker, indem bereits auf Modellebene von komplexen Modellierungen abstrahiert wird. Einige Transformationen nutzen jedoch bei der Umwandlung von *extended* Features selbst andere *extended* Features. Dieses Vorgehen wurde gewählt, um die einzelnen Modelltransformationen zu vereinfachen und die Transformationslogik nicht vervielfachen zu müssen. Somit ergeben sich zwischen den Transformationen Abhängigkeiten in der Ausführungsreihenfolge, die in Abbildung 1.3 in Form eines Graphen dargestellt sind.

Weiterhin ist bei Core SCCharts auch eine Normalisierung möglich. Ein normalisierter Core SCChart hat meist mehrere Modellelemente, die dann jedoch in einer klareren und einfacheren Struktur angeordnet sind. Dies vereinfacht die Überführung der SCCharts in einen anderen Modelltyp.

Die Semantik der sequenziellen Konstruktivität ist mithilfe eines Sequentially Constructive Graph (SCG) definiert, der in seiner Struktur an einen Kontrollflussgraphen angelehnt ist. Ein solcher SCG kann, aufgrund der ähnlichen Struktur, leicht aus normalisierten Core SCCharts transformiert werden, damit ein entsprechender Scheduler und Compiler daraus ausführbaren Code erzeugen kann.

1. Einleitung

Zusammengefasst werden bei SCCharts viele und komplexe Strukturen von Modelltransformationen eingesetzt. Abbildung 1.4 fasst exemplarisch die Schritte der Kompilierungskette von Extended SCCharts hin zu einem lauffähigen Software- oder Hardware-Programm zusammen. Die einzelnen Kompilierungsschritte bestehen, wie bereits beschrieben, aus einer Abfolge von Transformationen. Daher ist es von Interesse diese komplexen Abläufe mit Bezug zu den Modellen, die kompiliert werden, genau zu erfassen und zu verstehen, vorzugsweise in einer visuellen Form. Da es sich bei SCCharts um eine Programmiersprache handelt, sind auch die Simulation und das Debugging wichtige Bereiche, wenn eine effiziente Modellierung mit SCCharts ermöglicht werden soll. Hierzu bedarf es Mappings, die eine Verbindung zwischen dem Quellmodell und dem ausführbaren Programmcode herstellen.

1.4. Problemstellung

Das Tracing der Modelltransformationen ist ein wichtiger Aspekt in der modellgetriebenen Entwicklung und löst einige der hier und von Schmidt [Sch06] festgestellten Probleme, die den Einsatz von Modellen zur Zeit noch erschweren. Jedoch wird das Tracing von den meisten Modellierungswerkzeugen nicht unterstützt, da die Modelltransformationen nicht in der Lage sind, automatisch ein Mapping zu generieren. Insbesondere ist dies bei der Implementierung der SCCharts-Transformationen in KIELER der Fall. Dies hat negative Auswirkungen in verschiedenen Ausprägungen:

- Entstehungsbeziehungen können nicht rückverfolgt werden.
- Informationen zu Modellelementen können nicht auf äquivalente Elemente anderer Modelle übertragen werden.
- Durchgeführte Transformationsabfolgen auf Modellen können nur schwer visualisiert und analysiert werden.

Es lässt sich feststellen, dass ohne Tracing-Informationen zwischen den einzelnen Modellen der Transformationsabfolgen keine nutzbaren semantischen Beziehungen bestehen. Insbesondere sind daher auch keine direkten Mappings zwischen beliebigen Modellen einer Transformationskette möglich, was die effiziente Übertragen von Informationen erleichtern würde. Weiterhin wird durch das Fehlen einer Visualisierung der Abläufe und Beziehungen das Verständnis der durchgeführten Transformationsabfolgen erschwert.

Das entwickelte Framework soll diese Probleme lösen und das Tracing von Modelltransformationen ermöglichen. Dabei soll es möglich sein, die Entstehungsbeziehungen zwischen Modellelementen während der Transformationen flexibel und generisch zu erfassen. Die auf diese Weise generierten Mappings

sollen zusammen mit den Transformationsabläufen in einer entsprechenden Datenstruktur gespeichert werden. Um möglichen Nutzern des Frameworks die Verwendung der Tracing-Informationen zu erleichtern, soll es weiterhin möglich sein, direkte Mappings zwischen beliebigen Modellen in einer Transformationskette zu ermitteln. Weiterhin soll eine Visualisierung der Transformationsabläufe und Beziehungen zwischen Modellen zur Verfügung stehen.

1.5. Aufbau

Zunächst werden in Kapitel 2 andere Arbeiten vorgestellt, die für ähnliche Problemstellungen Lösungen entwickelt haben, um diese Arbeit dagegen abzugrenzen. Kapitel 3 bietet einen kurzen Einblick in die Technologien, die für die Umsetzung des Frameworks verwendet wurden. Der Entwurf des Frameworks wird in Kapitel 4 präsentiert. Zunächst werden in Abschnitt 4.1 die Anforderungen definiert. Nachfolgend wird in Abschnitt 4.3 ein Metamodell entwickelt, das als Datenstruktur dient und mit dem insbesondere die im nachfolgenden Abschnitt vorgestellte Aufbereitung der Mappings möglich ist. Die eigentliche Implementierung des Frameworks als ein Eclipse Plug-in für das KIELER Projekt wird in Kapitel 5 vorgestellt. Die produzierten Ergebnisse werden anschließend in Kapitel 6 analysiert. Abschließend wird in Kapitel 7 ein Fazit zum entwickelten Framework gezogen und ein Ausblick auf dessen Einsatz gegeben.

Verwandte Arbeiten

In diesem Kapitel werden andere wissenschaftliche Arbeiten vorgestellt, die sich mit einer ähnlichen Problemstellung beschäftigt haben. Zunächst wird der Begriff Tracing gegen andere Themengebiete abgegrenzt. Anschließend werden Arbeiten vorgestellt, die ebenfalls Möglichkeiten zum Tracing von Modelltransformationen erarbeitet haben. Zuletzt wird eine Arbeit zur Spezifikation eines Mappings erörtert.

2.1. Tracing

Im Bereich der modellgetriebenen Softwareentwicklung tritt der Begriff Tracing in unterschiedlichen Ausprägungen auf. Galvão und Goknil [GG07] fassen in ihrer Arbeit die verschiedenen Ansätze zum Tracing zusammen. Dabei wurde zwischen drei Kategorien unterschieden.

- **Anforderungsgetriebene Ansätze:** In diesem Bereich ist das Ziel durch Tracing das Verständnis der Entwickler und Stakeholder für die Semantik der Abhängigkeiten des Systems zu verbessern.
- **Modellierungsansätze:** Hierbei handelt es sich um Ansätze, die den generellen Einsatz von Modellen und Metamodellierung für Schemata und zur Spezifikationen von Tracing bieten.
- **Transformationsansätze:** Die Ansätze in diesem Bereich beschäftigen sich mit Tracing von Modelltransformationen, um die Beziehungen von transformierten Modellen zu ihren Quellmodellen zu erfassen.

Diese Ansätze wurden dann anhand von Möglichkeiten zum Mapping, Skalierbarkeit und der Unterstützung durch Tools verglichen und die noch ungelösten Aspekte hervorgehoben.

Anhand der Einteilung beschreibt diese Arbeit primär einen Transformationsansatz für Tracing.

2.1.1. Tracing von Modelltransformationen

Verschiedene Arbeiten beschäftigen sich mit dem Thema des Tracings von Modelltransformationen, um eine bessere Anbindung der Modelle an Entwicklungs-

2. Verwandte Arbeiten

prozesse zu ermöglichen. In der Regel sind die Lösungen dabei an eine spezielle Transformationssprache gebunden. Einige dieser Sprachen werden im Folgenden vorgestellt und im Hinblick auf die Tracing-Eigenschaften mit dieser Arbeit verglichen.

Atlas Transformation Language

Die Atlas Transformation Language (ATL)¹ wurde von der ATLAS INRIA & LINA Forschungsgruppe [Jou+06] entwickelt und ist an das Query View Transformation (QVT) Prinzip der OMG angelegt. Sie ermöglicht die Transformation von Modellen, indem eine Menge von Regeln angegeben wird, die beschreibt, wie die einzelnen Elemente des Quellmetamodells in Zielelemente überführt werden sollen. Die Sprache selbst ist eine Mischung aus deklarativen und imperativen Programmierparadigmen und wird in einer eigenen ATL Virtual Machine (VM) ausgeführt. Listing 2.1 zeigt einen Ausschnitt aus einer ATL Transformation, bei der ein Buch in eine Publikation transformiert wird.

```
1 rule Book2Publication {
2   from
3     b : Book!Book
4   to
5     out : Publication!Publication (
6       title <- b.title,
7       nbPages <- b.getSumPages()
8     )
9 }
```

Listing 2.1. Teil einer Transformation in ATL²

Da Transformationen immer als ein Regelsatz für alle Metamodellelemente definiert werden müssen, ist die Sprache in Bezug auf den Einsatzbereich dieser Arbeit recht unflexibel. Michael Matzen [Mat10] bezeichnet ATL aus diesem Grund sogar als ein ungeeignetes Framework für strukturbasiertes Editieren von Modellen.

Tracing wird durch ATL in einer impliziten Form unterstützt, da bereits intern Relationen zwischen Elementen aus den Regeln abgeleitet werden. Den einzelnen Transformationsregeln werden dabei die Ausgaben anderer Regeln zur Verfügung gestellt, um Interaktionen zwischen diesen zu ermöglichen. Yie und Wagelaar [YW09] erweitern in ihrer Arbeit dieses Prinzip des impliziten Tracings, um es besser an die Anforderungen eines vollwertigen Tracing Frameworks anzupassen. Sie führen neue Funktionen ein, um die durch ATL zur Verfügung gestellten Relationen besser nutzen zu können. Zudem bieten sie einen Export der Relationen

¹www.eclipse.org/m2m/atl/

²[www.eclipse.org/atl/atlTransformations/Book2Publication/ExampleBook2Publication\[v00.02\].pdf](http://www.eclipse.org/atl/atlTransformations/Book2Publication/ExampleBook2Publication[v00.02].pdf)

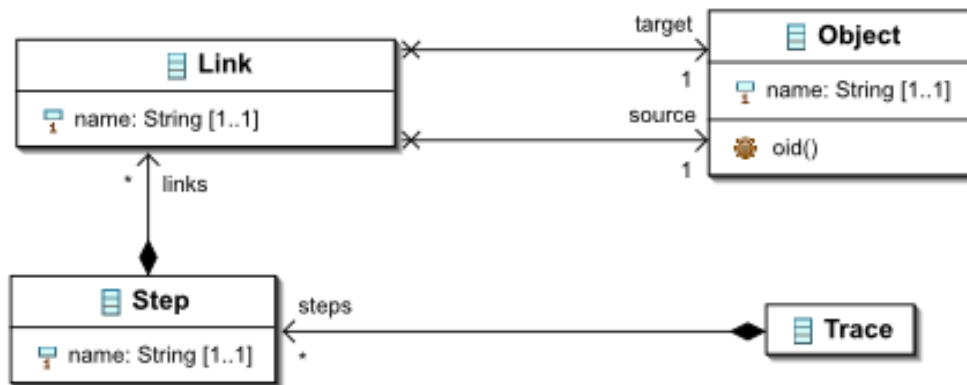


Abbildung 2.1. Metamodell für Tracing-Modelle in Kermeta [FHN06]

aus dem Bytecode der ATL-VM in ein EMF Modell an, um so eine Analyse der Tracinginformationen zu erleichtern.

Der Tracing-Ansatz von Yie und Wagelaar ist darauf angewiesen, dass alle Transformationen in ATL implementiert sind und von der ATL-VM ausgeführt werden. Im Gegensatz dazu steht in dieser Arbeit ein flexibler Ansatz im Vordergrund, der dem Programmierer der Transformationen die Möglichkeit gibt, ein Mapping zu generieren. Eine Übertragung dieses Prinzips auf ATL ist nicht möglich, da der deklarative regelbasierte Ansatz und das Fehlen von externen Prozeduraufrufen [Mat10] dies verhindern. Ein Export der Mappings in die Datenstruktur dieses Frameworks, ähnlich zu dem von Yie und Wagelaar, wäre wahrscheinlich möglich. Weiterhin fehlt es bei der Arbeit von Yie und Wagelaar aber an einer Unterstützung von Tracing bei Transformationsketten.

Kermeta

Kermeta³ ist eine Sprache zur Metamodellierung und zur aspektorientierten Programmierung von Modellerweiterungen und Transformationen. Moha et al. [Moh+10] präsentieren in ihrer Arbeit eine Evaluation von Kermeta in Bezug auf das Lösen von graphenbasierten Problemen. Dabei wurde festgestellt, dass sich Kermeta gut für Modellierung und das Implementieren insbesondere komplexer Algorithmen eignet, da das Framework performant arbeitet und leicht erweiterbar ist. Bei der Implementierung der Transformationen zeigten sich jedoch Schwierigkeiten durch die Umstellung auf die Kermeta Sprache und das Fehlen von statischen Analysen der Modelle.

Da Kermeta im Gegensatz zu ATL kein integriertes implizites Mapping besitzt, bedarf es eines Frameworks zum Tracing. Ein solches Framework haben Falleri et al.

³www.kermeta.org/

2. Verwandte Arbeiten

[FHN06] entwickelt. Dabei wurde ein ähnlicher Ansatz gewählt wie bei dieser Arbeit. Zunächst wurde ein Metamodell entworfen, dargestellt in Abbildung 2.1, das die Relationen zwischen Modellelementen darstellt und auch Transformationsketten unterstützt. Zur Erfassung der Relationen wird eine Schnittstelle angeboten, die vom Entwickler in die Implementierung der Transformationen eingebettet werden muss und die übergebenen Beziehungen in das Mapping aufnimmt. Auch wenn das Vorgehen von Falleri et al. vergleichbar ist mit dieser Arbeit, unterscheiden sich die Lösungen in der Ausprägung der Mappingschnittstelle und in den Eigenschaften des Tracingmodells. Die Schnittstelle ist sehr einfach gehalten, indem immer nur einzelne Relationen registriert werden können, wogegen in dieser Arbeit die Funktionalität umfangreicher angelegt ist, um sich besser an verschiedene Vorgehensweisen bei Transformationen anpassen zu können. Das Metamodell für das Tracing in Kermeta erlaubt nur eine ungeordnete Menge von Einzeltransformationen. In dieser Arbeit ist hingegen eine klarere Transformationskettenstruktur vorgesehen, die verschiedene Transformationsabfolgen in einem Tracingmodell erfassen kann. So ist es möglich, auch verschiedene Transformationspfade zu vergleichen, wie im Fall der Extended SCCharts.

Java

Eine allgemeine Hochsprache wie Java eignet sich ebenfalls für die Implementierung von Modelltransformationen, insbesondere wenn sie durch ein Transformations-Framework wie Xtend erweitert wird und zusätzliche Funktionen und Syntaxkonstrukte erhält, um eine modellnahe Programmierung und intuitive Implementierung von Transformationen zu ermöglichen.

Amar et al. [Ama+08] präsentieren in ihrer Arbeit ein Tool zum Tracing von Modelltransformationen in Java basierend auf EMF. Zur Erfassung der Relationen, die zwischen Quell- und Zielmodell entstehen, werden die einzelnen Transformationsereignisse erfasst und in ein Tracing-Modell übernommen. Dies wird erreicht, indem die Aufrufe an das Application Programming Interface (API) von EMF abgefangen werden und anhand von regulären Ausdrücken gefiltert werden. Jeweils vor und nach dem Ausführen einer Methode werden dann die erfassten Änderungen in ein dafür entworfenes Tracingmodell eingetragen.

Dieses Prinzip bietet einen hohen Grad an Automatisierung, jedoch keine Flexibilität in der Semantik des entstandenen Mappings, da man auf die Informationen der EMF-API Aufrufe und deren Interpretation durch das Tool beschränkt ist. Wenn beispielsweise eine Transformation als letzten Schritt eine Optimierung durchgeführt, also einige Modellelemente entfernt oder zusammenfasst, ist es fraglich, ob diese Änderungen automatisch und korrekt auf das bisherige Mapping übertragen werden können. Durch eine programmatische Registrierung der Relationen, wie in dieser Arbeit, kann sichergestellt werden, dass nur relevante und semantisch korrekte Relationen in das Mapping aufgenommen werden.

2.2. Spezifikation von Mappings

Das Paper von Lopes et al. [Lop+06] beschäftigt sich mit der allgemeinen Spezifikation von Mappings. Dabei werden zunächst die wichtigsten Eigenschaften von Mappings analysiert und anschließend ein Metamodell vorgestellt, mit dem ein Mapping spezifiziert werden kann. Die Spezifikation ist dabei auf Metamodellebene definiert, das heißt, es setzt die Elemente zweier Metamodelle in Beziehung und nicht die Modellelemente. Der Vorteil dieses Vorgehens wird anhand einer Implementierung als Eclipse Plug-in gezeigt, bei der sich aus einem derartig spezifizierten Mapping direkt die zugehörige Modelltransformation in ATL exportieren ließ.

Der vorgestellte Ansatz ist für die Spezifikation eines Mappings durchaus hilfreich, jedoch kann das Verfahren nur schwer auf die Problemstellung dieser Arbeit übertragen werden. Dies liegt hauptsächlich an der Definition des Mappings auf Metamodellebene, was zufolge hat, dass die Semantik einer jeden Modelltransformation in ein Mappingmodell übersetzt werden muss und dieses dann im Konflikt zu bestehenden Implementierungen der Transformationen stehen kann. Dabei ist fraglich, ob Transformationen die auf demselben Metamodell definiert sind, auch auf diese Art spezifizierbar sind.

Verwendete Technologien

In diesem Kapitel werden einige der Technologien vorgestellt, die im Kontext dieser Arbeit verwendet werden. In vorherigen Kapiteln wurde bereits vereinzelt Bezug auf diese Technologien genommen, daher steht hier eine etwas detailliertere Beschreibung im Vordergrund. Diese soll ein besseres Verständnis der Lösung und dessen Implementierung ermöglichen.

3.1. Eclipse Entwicklungsumgebung

Das Eclipse Projekt¹ besteht seit 2001 und wurde primär von IBM entwickelt. Es bietet eine quelloffene Java-basierte Entwicklungsumgebung für verschiedene Anwendungen, unter anderem dient es auch als Modellierungswerkzeug.

Das breite Feld von Einsatzbereichen² verdankt Eclipse seinem modularen Aufbau, anhand der OSGi-Spezifikationen³. Die von Eclipse implementierte Plug-in-Architektur ist in Abbildung 3.1 dargestellt. Dabei stellt die Eclipse Rich Client Platform (RCP) Kernkomponenten und Basis-Plug-ins zur Verfügung, die dann von weiteren und insbesondere auch durch eigene Plug-ins über ExtensionPoints erweitert werden können. Durch dieses Konzept ist es möglich neue Technologien in die Entwicklungsumgebung einzubinden oder eigene, auf Eclipse basierende Anwendungen zu erstellen, ohne dabei die gesamte Funktionalität einer Entwicklungsumgebung selbst implementieren zu müssen.

3.1.1. Eclipse Modeling Framework

Damit Eclipse auch für modellgetriebene Entwicklungsprozesse genutzt werden kann, wurde das Eclipse Modeling Framework (EMF) entwickelt. Dieses Framework ermöglicht die Spezifikation und automatische Implementierung von Modellen. Die Spezifikation von Modellen wird durch Metamodellierung auf Basis des zu diesem Zweck entwickelten *Ecore* Meta-Metamodells umgesetzt, wobei dem Anwender hierzu ein graphisches Modellierungswerkzeug zur Verfügung steht. Abbildung 3.2 zeigt einen Auszug aus dem *Ecore* Meta-Metamodell. Es ist

¹www.eclipse.org

²<http://heise.de/-1370644>

³www.osgi.org

3. Verwendete Technologien

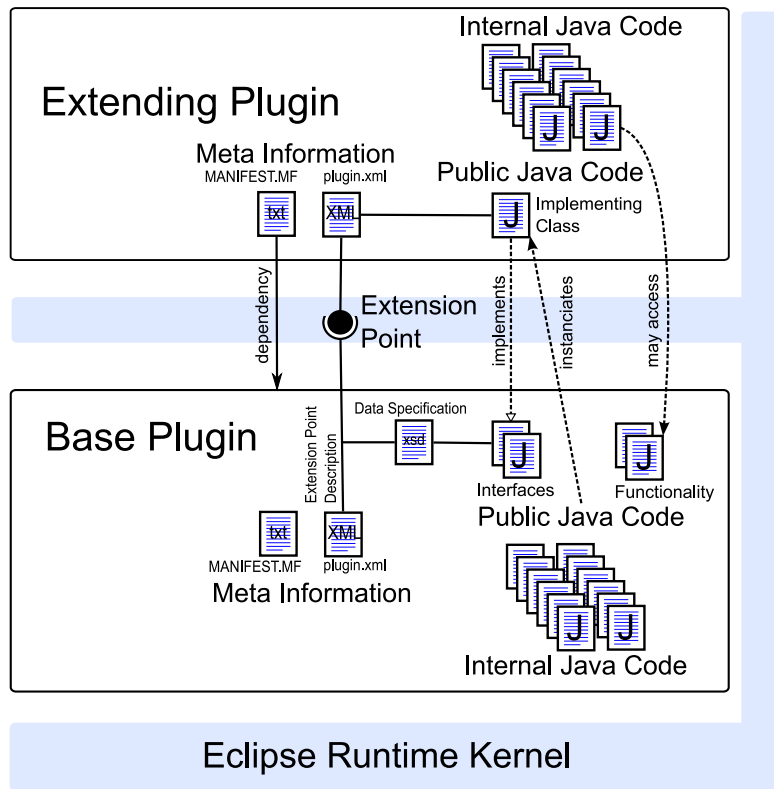


Abbildung 3.1. Konzept der Eclipse Plug-in-Architektur [Fuh11]

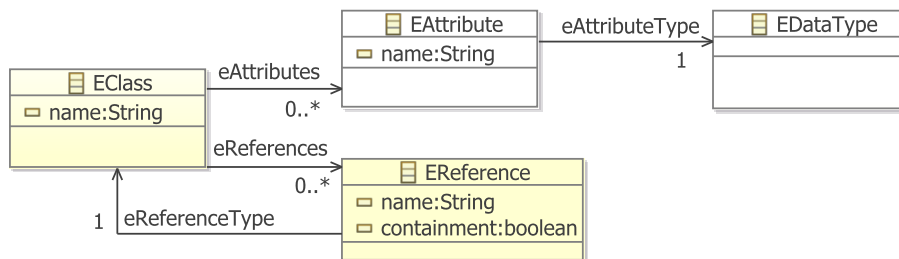


Abbildung 3.2. Vereinfachter Auszug aus dem Ecore Meta-Modell [Mot09]

zu erkennen, dass Modellelemente als Klassen spezifiziert sind, die getypte Attribute enthalten und mit anderen Klassen über Referenzen in Beziehung stehen. Anhand einer solchen auf Ecore basierenden Modellspezifikation in Form eines Metamodells ist es nun möglich, mit EMF eine Implementierung der Modelle in Java zu generieren. Weiterhin kann auch direkt ein Editor zur Modellerstellung in Form eines Eclipse Plug-ins generiert werden.

3.1.2. Xtend

Die Modelltransformationssprache Xtend⁴ wurde von der itemis AG⁵ entwickelt und bietet eine gute Integration von EMF. Ziel von Xtend ist es eine Programmiersprache zu bieten, die den bestehenden Funktionsumfang von Metamodellimplementierungen erweitert, um die Implementierung von Modelltransformationen zu vereinfachen. Xtend ist dabei stark an Java angelehnt und wird zur Ausführung vollständig in Java übersetzt, weshalb nativ die Nutzung von Java-Bibliotheken unterstützt wird. Zusätzlich zu den Konstrukten aus Java wurden im großen Umfang funktionale Programmierparadigmen und syntaktischer Zucker hinzugefügt. So bietet Xtend Lambda-Ausdrücke und Typinferenz, bei der die Typen von Variablen und Rückgabewerten aus dem Kontext inferiert werden. Zudem wurden *extensions* eingeführt, das sind Methoden wie *map*, *filter* oder *sort*, die direkt auf ihrem ersten Argument aufgerufen werden können, ohne es zu verändern. Dies ermöglicht eine kompakte und leserliche Aneinanderreihung von Funktionsaufrufen.

3.1.3. Guava

Google Guava⁶ ist eine quelloffene Sammlung von Programmbibliotheken für Java. Diese wurden von Google entwickelt, um eine erweiterte Funktionalitätsbasis für Java-Projekte zu bieten. Um dies zu erreichen wurden einige Standardbibliotheken im Funktionsumfang erweitert oder auf spezielle Einsatzgebiete angepasst. Zusätzlich wurden neue Programmierkonzepte, wie funktionale Paradigmen, hinzugefügt.

Dabei sind unter anderem die neuen Datenstrukturen für diese Arbeit von Interesse. Insbesondere die *MultiMap*, die eine dynamische Laufzeitdatenstruktur nach dem Prinzip einer *Map* darstellt. Sie ermöglicht es, unter einem Schlüssel mehrere Werte abzuspeichern und somit den Anforderungen an Mappings gerecht zu werden.

Auch die Xtend Sprache nutzt und unterstützt in einigen Bereichen den erweiterten Sprachumfang von Guava.

⁴www.eclipse.org/xtend/

⁵www.itemis.de/

⁶www.code.google.com/p/guava-libraries/

3. Verwendete Technologien

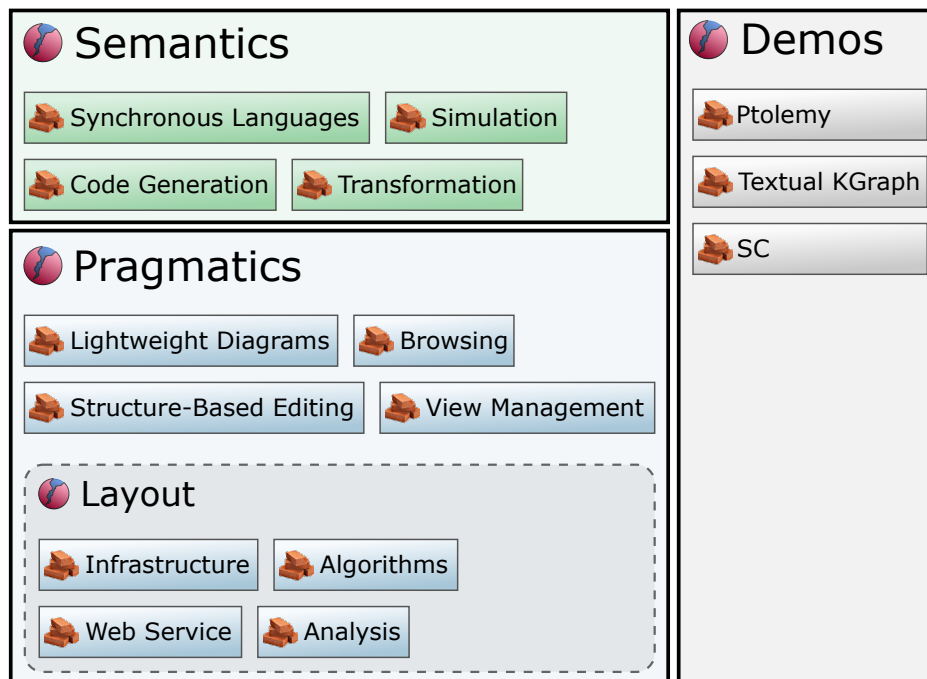


Abbildung 3.3. Struktureller Aufbau des KIELER Projekts⁷

3.2. Kiel Integrated Environment for Layout Eclipse Rich Client

Das Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)⁸ Forschungsprojekt, bekannt von den SCCharts aus Abschnitt 1.3, wird von der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme der Christian-Albrechts-Universität zu Kiel entwickelt. Der Fokus des Projekts liegt auf der Entwicklung neuer verbesserter Methoden für den graphischen modellbasierten Entwurf von Systemen. Abbildung 3.3 zeigt die strukturelle Aufteilung des Projekts.

Der Bereich Pragmatics [FH10] behandelt die praktischen Aspekte der modellgetriebenen Entwicklung. Dies umfasst unter anderem das strukturbasierte Editieren, sowie das Erstellen von Diagrammen. Dazu kommen noch die Aspekte des Layouts, die eine wichtige Rolle bei dem Verständnis von Modellen spielen. Dabei steht besonders ein automatisches Layout im Vordergrund, sowie effiziente Layout-Algorithmen.

Der Semantics Teil deckt den Bereich der Kompilierung und Simulation von Modellen ab, um auch die Semantik der Ausführung eines Metamodells definieren zu können.

⁷www.rtsys.informatik.uni-kiel.de/confluence/display/KIELER/0verview

⁸www.informatik.uni-kiel.de/rtsys/kieler

3.2. Kiel Integrated Environment for Layout Eclipse Rich Client

Die Demos, zu denen unter anderem die SCCharts aus Abschnitt 1.3 gehören, sind Technologien die entwickelt wurden, um die Prinzipien der anderen Bereiche zu testen und in realitätsnahen Anwendungsbeispielen umzusetzen.

3.2.1. KIELER Lightweight Diagrams

KIELER Lightweight Diagrams (KLighD) ist ein Framework für die Erstellung und das Anzeigen von leichtgewichtigen Diagrammen in Eclipse. Es ist Teil des KIELER Projekts und ist dem Bereich der Pragmatik zuzuordnen. Die Konzepte von KLighD und deren Details in Bezug auf die Anwendung werden in der Publikation von Schneider et al. [SSH13] beschrieben. Mithilfe von KLighD können Synthesen zu Modellen erstellt werden, die generisch ein Diagramm erzeugen, das dann angezeigt werden kann. Dabei wird zunächst ein automatisches Layout durchgeführt, um die Elemente anzuordnen. Auf diese Weise kann der Aufwand, der für eine effiziente Visualisierung nötig ist, reduziert werden und der Prozess automatisiert werden. Abbildung 3.4 zeigt den schematischen Ablauf der Diagrammerstellung durch KLighD. Dabei wird ein Eingabemodell mit der entsprechenden Diagrammsynthese zu einem Graphen transformiert, mit KIELER Infrastructure for Meta Layout (KIML) ein Layout angewendet und das fertige Diagramm dann angezeigt. Weiterhin ist KLighD gut an bestehende Konzepte und Werkzeuge zur Modellierung angebunden, so ist es beispielsweise möglich Synthesen komfortabel in Xtend zu implementieren. Ein weiterer Aspekt ist die Interaktivität der angezeigten Diagramme, die unter anderem Zoom und Highlighting unterstützen. Weiterhin können Inhalte von Diagrammkomponenten eingeklappt werden, um die Übersichtlichkeit zu erhöhen, oder interaktive Aktionsmöglichkeiten in ein Diagramm integriert werden.

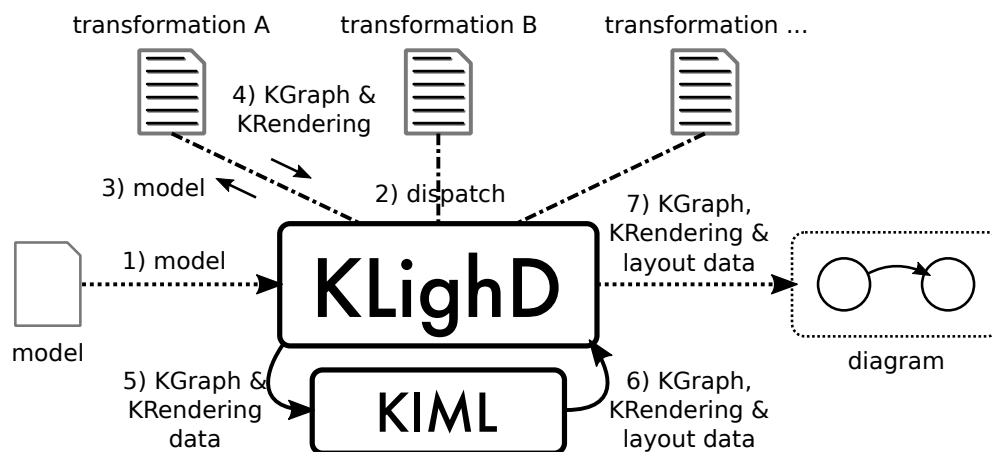


Abbildung 3.4. Schematischer Ablauf der Diagrammerstellung durch KLighD [SSH13]

Entwurf des Tracing-Frameworks

In diesem Kapitel wird der Entwurf der Infrastruktur des Tracing-Frameworks vorgestellt. Zunächst werden in Abschnitt 4.1 die Anforderungen festgelegt, anhand derer das Projekt in vier Teile gegliedert wurde. Die weiteren Abschnitte stellen detailliert das Konzept vor, das den entsprechenden Komponenten zugrunde liegt.

4.1. Anforderungen

Bevor mit dem eigentlichen Entwurf begonnen werden kann, müssen zunächst die einzelnen Anforderungen, ausgehend von der initialen Problemstellung, spezifiziert werden.

In Kapitel 1 wurden bereits verschiedene Aspekte der Problemstellung erwähnt. Diese lassen sich in einer abstrakten Problemstellung zusammenfassen.

Abstraktes Problem: Es fehlt die Möglichkeit zur flexiblen Erzeugung von Mappings während Modelltransformationen, sowie Strukturen zur Speicherung, Aufbereitung und Visualisierung der neu gewonnenen Informationen.

Um eine detailliertere Anforderungsstellung zu ermöglichen, wird die Problemstellung zunächst weiter eingeschränkt.

Dazu zählt, dass die Modelltransformationen auf M2M-Transformationen beschränkt werden. Dies vereinfacht die Spezifikation des Mappings, da Elemente über ihre generalisierten Basiselemente im Metamodell referenziert werden können. Damit fehlt zwar eine direkte Anbindung des Mappings an beispielsweise generierten Code oder andere textuelle Formen ohne Metamodell, jedoch kann dies mithilfe einer Modellrepräsentation des Textes gelöst werden. Dabei wird die textuelle Form durch ein Modell dargestellt, das es ermöglicht diese in das Mapping mit einzubeziehen. Im Anwendungsbeispiel der SCCharts existiert dieses Konzept bereits in Form der modellbasierte Zwischensprache *S*, die als Abstraktion der Programmiersprachen wie C und Java fungiert.

Weiterhin ist zu beachten, dass Mappings als Relationen zwischen Modellelementen definiert sind. Sie entstehen daher dynamisch während der Durchführung von Modelltransformationen auf einem konkreten Modell.

4. Entwurf des Tracing-Frameworks

Dies steht im Gegensatz zu dem Konzept von Lopes et al. [Lop+06], bei dem im Vorwege ein Mapping auf Metamodellebene erstellt wird, das dann auf alle Modelle übertragbar ist und aus dem die Modelltransformation abgeleitet werden.

Wenn man die abstrakte Problemstellung betrachtet, können vier Kernanforderungen gefolgert werden, die zu erfüllen sind.

1. Flexible minimalinvasive *Schnittstelle* zur Erzeugung von Mappings während Transformationen
2. *Datenstruktur* für Mappings und Transformationsabläufe
3. Hilfreiche Möglichkeiten zur *Datenaufbereitung*
4. *Visualisierung* der Informationen aus der Datenstruktur

In Unterabschnitt 4.1.1 wird zunächst auf die Anwendungsfälle eingegangen, die für das Framework auftreten können. Die Anforderungen dieser Kernbereiche, die sich unter anderem aus den Anwendungsfällen ergeben, werden in den Unterabschnitten 4.1.2 bis 4.1.5 näher spezifiziert.

4.1.1. Anwendungsfälle

Abbildung 4.1 zeigt ein Anwendungsfalldiagramm für das Framework. Es beschreibt den Fall der Nutzung des Framework zum Tracing bei Modellen in einer Anwendung. Dabei sind die Akteure sowohl die Entwickler als auch die Nutzer der Anwendung. Die Entwickler nutzen die Schnittstelle, die Datenstruktur und die Aufbereitungsfunktionalität des Frameworks, um in ihrer Anwendung ein Tracing von Modelltransformationen durchzuführen. Die Benutzer hingegen verwenden, aufbauend auf den zur Verfügung gestellten Funktionen der Anwendung, die Visualisierung der Modelltransaktionsketten, um diese besser zu verstehen und das Debugging von Modellen der Anwendung, das durch das Tracing ermöglicht wird.

Es zeigt sich, dass alle Anwendungsfälle durch die vier Kernbereiche abgedeckt werden können.

4.1.2. Schnittstelle

Die Schnittstelle zur Erstellung der Mappings soll Funktionen anbieten, die es einem Entwickler ermöglichen, die Registrierung von Entstehungsbeziehungen in Modelltransformationen einzubetten. Es soll sowohl möglich sein die Funktionalität während der Neuerstellung einer Modelltransformation einzubinden, als auch in eine bestehende Implementierung zu integrieren. Dabei sollen die Entstehungsbeziehungen der Elementinstanzen während der Ausführung der

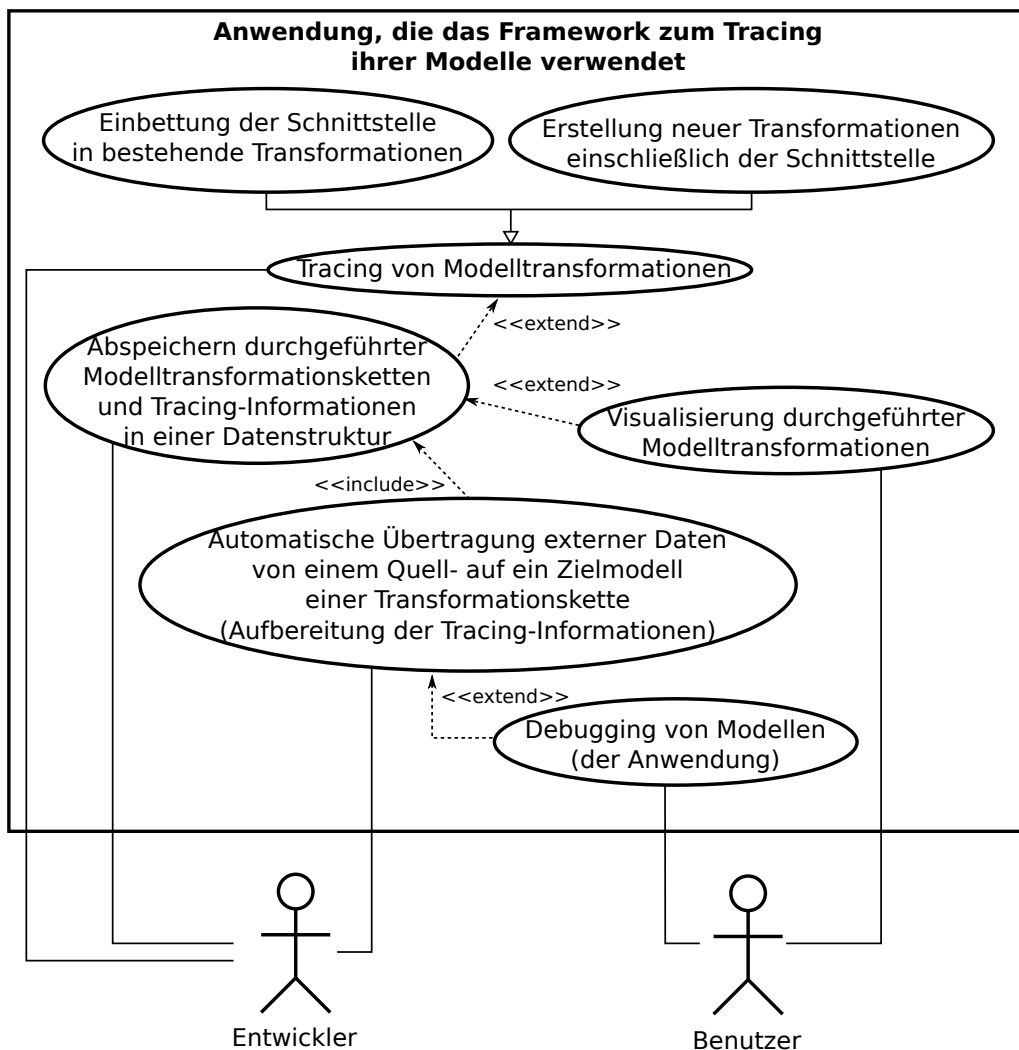


Abbildung 4.1. Anwendungsfalldiagramm der Frameworks

Modelltransformation erfasst und in eine passende Datenstruktur aufgenommen werden.

Weiterhin muss die Schnittstelle genug Funktionalität bieten, um sich an verschiedene Strategien der Implementierung von Modelltransformationen anpassen zu können. Gleichzeitig soll die Anzahl der Schnittstellenzugriffe, bei der Integration in die Transformationen möglichst gering gehalten werden.

4. Entwurf des Tracing-Frameworks

4.1.3. Datenstruktur

Kernaufgabe der Datenstruktur ist es, die Tracing-Informationen strukturiert zu repräsentieren und zu speichern.

Dazu muss sie zunächst in der Lage sein, mehrere korrespondierende Modelltransformationen zu erfassen. Denn meist werden mehrere Modelltransformationen hintereinander durchgeführt, um ein Modell zu transformieren. Da diese Reihenfolgeninformationen für mögliche Analysen und die Aufbereitung der Mappings relevant sind, muss diese Struktur der durchgeführten Transformationen in der Datenstruktur erfasst werden. Anlass dazu geben die Modelltransformationen zur Umwandlung von Extended SCCharts in Core SCCharts. Wie in Abbildung 1.3 zu erkennen ist, existieren mehrere Pfade durch den Abhängigkeitsgraph der *extended* Features, also existieren auch mehrere mögliche Transformationsreihenfolgen. Die Datenstruktur muss daher auch in der Lage sein, mehrere durchgeführte Transformationsabfolgen zu erfassen, damit ein Nutzer diese kompakt vergleichen kann.

Darüber hinaus muss die Datenstruktur die Mappings zu den einzelnen Modelltransformationen erfassen können. Dabei ist wichtig, dass alle Arten von Relationen, die zwischen den Elementen zweier Modelle durch eine Transformation auftreten können, dargestellt werden können.

Die Abbildung 4.2 zeigt die möglichen Relationen, in denen die Modellelemente stehen können. Die Eins-zu-Eins-Beziehung in Abbildungsteil 4.2a beschreibt beispielsweise eine einfache Umwandlung eines Elements in ein anderes Element im neuen Modell. In Teil 4.2b ist eine Eins-zu-Viele-Beziehung zu erkennen, die auftritt, wenn zum Beispiel ein abstraktes Element im Modell zu einer Darstellung mit mehreren einfacheren Konstrukten transformiert wird. Die Viele-zu-Eins-Beziehung, dargestellt in Abbildungsteil 4.2c, ist ein Beispiel für Relationen die bei Modelltransformationen entstehen und eine strukturelle Optimierung des Modells durchführen.

Zusammengefasst sollen beliebige Strukturen von durchgeführten Modelltransformationen und deren Mappings von der Datenstruktur erfasst werden können.

4.1.4. Datenaufbereitung

In diesem Bereich ist die Kernanforderung dem Nutzer die Möglichkeit zu bieten, auf die Datenstruktur der Mappings zuzugreifen und eine Datenaufbereitung durchzuführen. Der wichtigste Punkt ist dabei das Zusammenführen von Mappings bei Ketten von Transformationen. Wie am Beispiel der SCCharts aus Abschnitt 1.3 zu bemerken ist, werden meist mehrere Modelltransformationen hintereinander durchgeführt. Dabei sind die einzelnen Mappings meist nur noch von geringerem Interesse, da man ein Mapping über mehrere oder alle Transfor-

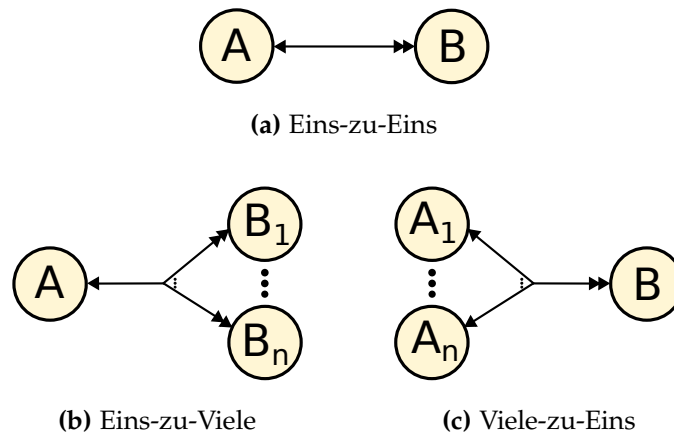


Abbildung 4.2. Mappingbeziehungen zwischen Modellelementen

mationsschritte hinweg benötigt, beispielsweise zwischen Start- und Endmodell der Kette. Genau dies soll die Aufbereitung bieten, indem sie zwischen beliebigen, in der Datenstruktur abgelegten Modellen die einzelnen Zwischenmappings zu einem direkten Mapping zusammenfasst.

4.1.5. Visualisierung

Die Visualisierung soll es ermöglichen, die in der Datenstruktur enthaltenen Informationen intuitiver zu erfassen. Hierzu gehört sowohl eine Visualisierung der Struktur der gespeicherten Modelltransformationen als auch der dazugehörigen Mappings.

Zudem sollte die bereits im Framework enthaltene Funktionalität zur Aufbereitung, die in Unterabschnitt 4.1.4 beschrieben wurde, interaktiv in die Visualisierung eingebunden werden, sodass dem Nutzer direkt eine Möglichkeit zur visuellen Analyse der Daten zur Verfügung steht.

4.2. Schnittstelle

Ein wichtiger Teil des Frameworks ist die Schnittstelle zur Erstellung der Mappings, da diese durch den Anwender genutzt werden muss, damit ein korrektes Mapping überhaupt entstehen kann. Da es das Ziel ist, mithilfe der Schnittstelle die Mapperstellung in bereits bestehende Implementierungen von Modelltransformationen einzubetten, muss dieses sich an die verschiedenen Transformationsstrategien anpassen können. Um diese Vorgehensweisen zu ermitteln, wurden die bestehenden Modelltransformationen von SCCharts betrachtet, und es zeigten sich zwei grundlegende Strategien.

4. Entwurf des Tracing-Frameworks

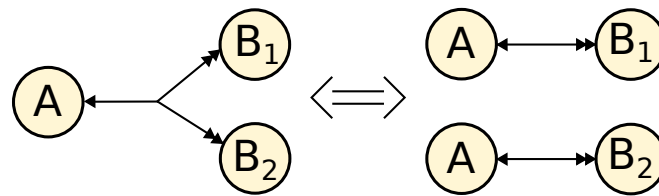


Abbildung 4.3. Aufspalten einer Eins-zu-Viele-Beziehung in eine Menge von einzelnen Relationen

Erstens die inkrementelle Erstellung des neuen Modells. Hierbei wird über die einzelnen Elemente des Quellmodells iteriert und zu jedem ein oder mehrere neue Elemente für das Zielmodell erstellt.

Die zweite Vorgehensweise ist ein kopiebasiertes Ändern. Diese Strategie wird genutzt, wenn nur einzelne Aspekte des Modells transformiert werden sollen. Dabei wird zunächst eine Kopie des Quellmodells erzeugt und anschließend verändert bis das fertig transformierte Zielmodell entstanden ist.

Somit ergeben sich die folgenden Anforderungen an die Funktionalität der Schnittstelle.

- Einfügen einer Eins-zu-Eins-Entstehungsbeziehung zwischen Elementen
- Einfügen einer Eins-zu-Viele-Entstehungsbeziehung zwischen Elementen
- Automatische Erstellung von Eins-zu-Eins-Beziehungen zu einer Menge von Elementen (z.B. beim Kopieren)
- Entfernen einer Entstehungsbeziehung
- Auslesen der Relationen zu einem Element

Auf diese Weise kann parallel zur Modelltransformation das Mapping erstellt werden, indem jeweils bei der Erstellung einer Instanz eines Modellelements ihre Entstehungsbeziehung registriert wird.

Dieses Vorgehen setzt jedoch voraus, dass während der inkrementellen Erstellung und Änderung des Mappings die Zwischenzustände der einzelnen Relationen in einer dynamischen Datenstruktur gespeichert werden und erst am Ende als fertiges Mapping zur Verfügung gestellt werden.

Weiterhin ist das Konzept der Registrierung der Mappingrelationen durch den Nutzer potenziell fehleranfällig, da leicht Elemente vergessen werden oder falsche Relationen erzeugt werden können. Um dies zu verhindern, ist eine Überprüfung der Vollständigkeit des Mappings notwendig. Hierzu kann am Ende einer Transformation das Quell- und Zielmodell mit dem Mapping abgeglichen werden, um Fehler oder Abweichungen zu erkennen.

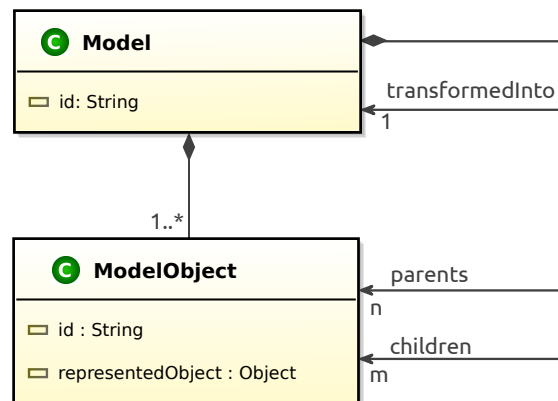


Abbildung 4.4. Minimaler Entwurf für das Metamodell

4.3. Datenstruktur

Eine weitere wichtige Komponente des Frameworks ist die Datenstruktur für die Mappings. Zunächst muss diese in der Lage sein, alle Arten der in den Anforderungen spezifizierten Beziehungen zwischen Modellelementen, dargestellt in Abbildung 4.2, zu erfassen. Hierzu bietet es sich an, die Relationen als eine Menge von Paaren aus Quell- und Zielelement zu betrachten, bei denen Elemente auch mehrmals als Quelle oder Ziel auftreten dürfen. Die Eins-zu-Eins-Beziehung ist bereits eine solche einelementige Menge. Die Abbildung 4.3 zeigt, wie eine Eins-zu-Viele-Beziehung in eine Menge von Einzelrelationen umgewandelt wird. Dieses Vorgehen ist entsprechend auch auf Viele-zu-Eins-Beziehung anwendbar. Zusätzlich zu den Mappings muss die Datenstruktur auch die Modelltransformationsabfolgen erfassen, die ausgehend von einem Quellmodell durchgeführt wurden.

Um anhand der Anforderungen die Datenstruktur korrekt zu definieren, wurde ein modellbasierter Ansatz gewählt.

4.3.1. Entwürfe

Zunächst wurden einige Entwürfe für das Metamodell der Datenstruktur angefertigt, um zu ermitteln, mit welchen Ansätzen die gegebenen Anforderungen am besten umzusetzen sind.

Minimaler Ansatz

Im ersten Entwurf, dargestellt in Abbildung 4.4, wurde der Ansatz verfolgt, das Metamodell möglichst minimal zu halten.

4. Entwurf des Tracing-Frameworks

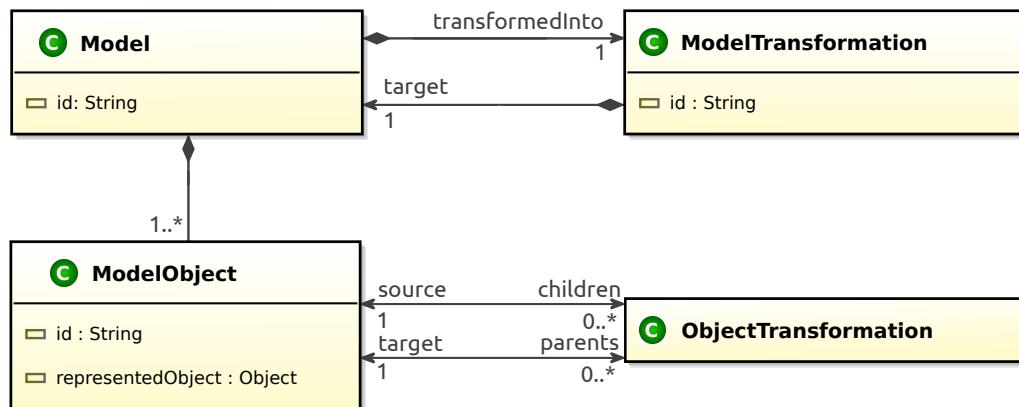


Abbildung 4.5. Erweiterter Entwurf für das Metamodell

Es enthält Modellelemente der Klasse `Model`, die ein transformiertes Modell repräsentieren und jeweils auf das Modell verweisen, das aus ihnen durch eine Transformation entstanden ist. Somit ergibt sich die Struktur einer verketteten Liste unter den repräsentierten Modellen. Weiterhin enthält jedes `Model` die Repräsentationen seiner Modellelemente in Form von mehreren `ModelObject`, die auf die konkreten Elementinstanzen verweisen.

Diese besitzen Assoziationen zu anderen `ModelObject`, um die einzelnen Relationen innerhalb des Mappings abzubilden.

Erweiterter Ansatz

In einem zweiten Entwurf wurde der minimale Ansatz aufgegriffen und um explizite Repräsentationen der Transformationen erweitert.

In Abbildung 4.5 ist zu erkennen, dass dieser weiterhin aus `Model` und `ModelObject` besteht, die auch ihre Bedeutung behalten haben. Jedoch wurden die Transformationen, die zuvor einfache Selbstassoziationen waren, als eigene Klassen umgesetzt. Somit tauchen auch Repräsentationen der Transformationen selbst im Metamodell auf, mit Verweisen auf ihre Quellen und Ziele.

Transformationszentrierter Ansatz

Beim dritten Entwurf wurde der Aspekt der Transformation als eigene Klasse aufgegriffen und ins Zentrum des Modells gestellt.

Wie in Abbildung 4.6 zu erkennen, enthält dieses Metamodell keine Repräsentation der Modelle mehr. Diese Informationen sind im zentralen Element `ModelTransformation` aufgegangen, das jeweils die Repräsentationen der Modellelemente des Quell- und Zielmodells enthält. Die `SourceModelObjects` und `TargetModelObjects` untereinander besitzen erneut mehrfache Assoziationen, um das Mapping abbilden

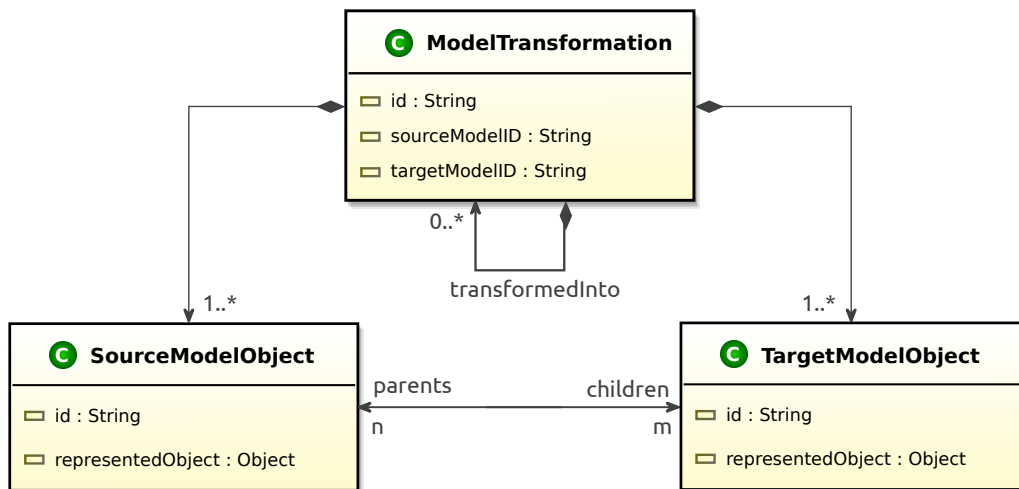


Abbildung 4.6. Transformationszentrierter Entwurf für das Metamodell

zu können. Weiterhin besitzt die `ModelTransformation`-Klasse eine Selbstassoziation, die auf mehrere direkt nachfolgende Transformationen verweisen kann, und so in der Lage ist auch verschiedene Abläufe von Transformationen zu erfassen. Dabei ergibt sich aufgrund der Aggregation und der Multiplizität eine Baumstruktur.

Vergleich

In Tabelle 4.1 werden einige charakteristische Eigenschaften der drei Entwürfe gegenüber gestellt.

Zunächst kann festgestellt werden, dass alle Entwürfe es ermöglichen ein Mapping der Modellelemente zu repräsentieren. Weiterhin sind die beiden ersten Entwürfe nicht in der Lage mehrere Ketten von Transformationen zu modellieren, da sie nur Listenstrukturen unterstützen. Dies resultiert aus der fehlenden Zuordnung von Einzelmappings zu einer Modelltransformation, was bei mehreren Nachfolgetransformationen eine Vermischung der Mappings zufolge hätte. Dies ist beim transformationszentrierten Ansatz nicht der Fall, da hier das Mapping mit Quell- und Zielmodell direkt an die Transformation gebunden ist und so auch bei mehreren Nachfolgern nicht vermischt wird.

Tabelle 4.1. Charakteristische Eigenschaften der Entwürfe

Eigenschaft	Minimal	Erweitert	Transformation
Transformationsstruktur	Liste	Liste	Baum
Repräsentation des Modells	ja	ja	ja
Repräsentation der Transformation	nein	nein	ja

4. Entwurf des Tracing-Frameworks

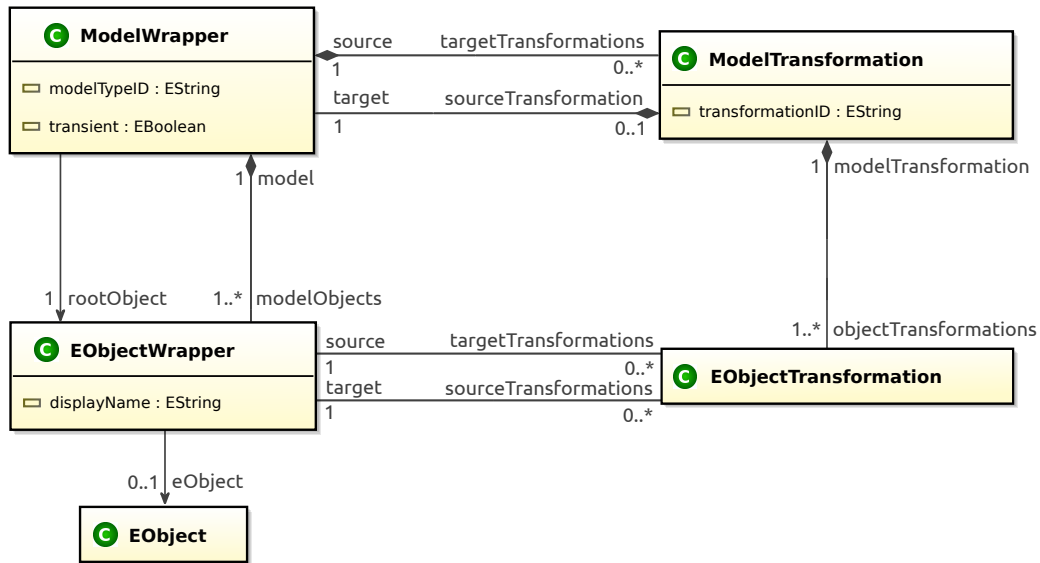


Abbildung 4.7. Metamodell der Datenstruktur

Der Nachteil bei diesem Entwurf ist jedoch, dass es die Modelle nicht als eigene Modellelemente repräsentiert, sondern immer nur als Teile der Transformation. Dies führt dazu, dass Modelle mehrfach repräsentiert werden müssen, wenn sie beispielsweise als Zielmodell und in einer darauf folgenden Transformation erneut als Quellmodell auftreten. Hierbei kann dann nicht auf die bereits erfassten Elemente zurückgegriffen werden. Dies ist wiederum beim ersten und zweiten Entwurf möglich.

Das Ergebnis des Vergleichs ist, dass kein Entwurf alle Anforderungen an die Datenstruktur erfüllen kann, jedoch die einzelnen Entwürfe jeweils unterschiedliche Aspekte ermöglichen.

4.3.2. Resultat

Auf Basis der Ergebnisse aus dem Vergleich der Entwürfe wurde das endgültige Metamodell, dargestellt in Abbildung 4.7, entwickelt. Dabei wurden die Aspekte des erweiterten und des transformationszentrierten Ansatzes aufgegriffen und in einem Modell vereint. So lassen sich sowohl korrekte Baumstrukturen für die Transformationen bilden, als auch die Modelle und Transformationen geeignet repräsentieren.

In der oberen Hälfte des Metamodells, aus Abbildung 4.7, sind die beiden Modellelemente dargestellt, mit denen die Transformationsstruktur modelliert werden kann. Wie im Entwurf aus Abbildung 4.5 sind dies Repräsentationen der Modelle und Modelltransformationen. Durch die Wahl der Multiplizitäten kann sich eine Baumstruktur ergeben, deren Zyklensfreiheit durch die Aggregation zwischen

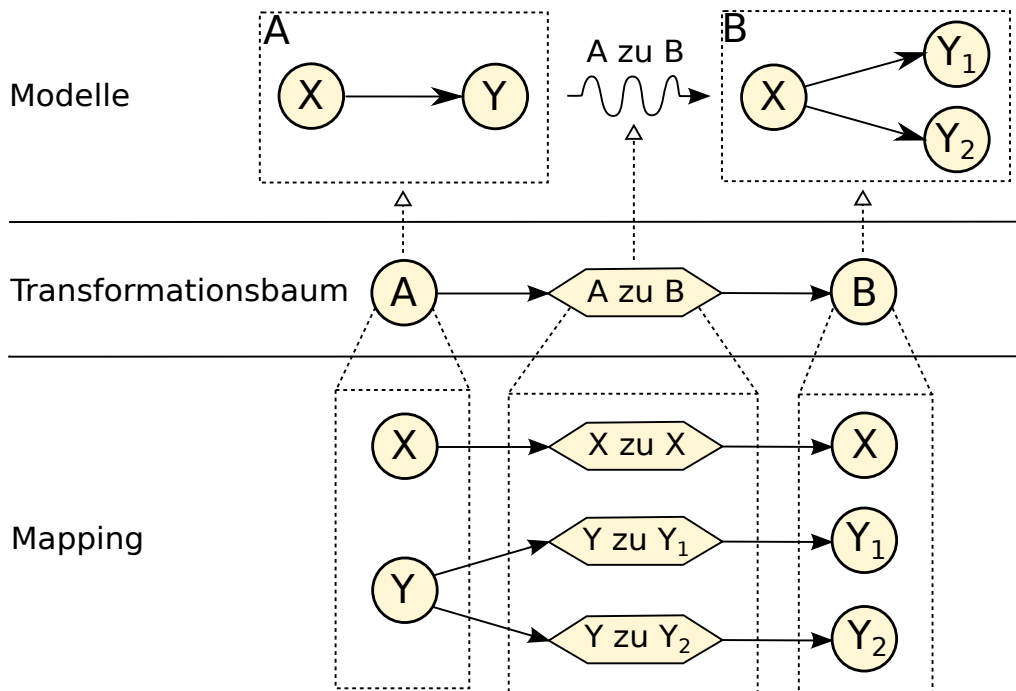


Abbildung 4.8. Beispiel für ein Modell des Metamodells

ModelWrapper und ModelTransformation sichergestellt ist. So können auch mehrere Transformationsabfolgen in der Struktur gespeichert werden.

Das Mapping kann mit den Elementen in der unteren Hälfte des Metamodells modelliert werden. Erneut ist die Struktur an den zweiten Entwurf angelehnt und repräsentiert die einzelnen Elemente der Modelle und deren Relationen. Die Relationen, die als EObjectTransformation bezeichnet werden, sind jeweils einer Modelltransformation zugeordnet. Dies ist ein Aspekt, der dem Entwurf aus Abbildung 4.6 folgt und sicherstellt, dass ein Mapping sich nicht mit anderen Mappings vermischt, die aus Transformationen desselben Quellmodells entstanden sind. Die einzelnen Modellelemente in den Mappings, hier EObjectWrapper, gehören zur den entsprechenden Repräsentation ihres Modells. Um die Navigation in der Datenstruktur zu erleichtern wurden viele Assoziationen bidirektional definiert und ein Verweis auf ein Wurzelement innerhalb eines Modells eingeführt. Die Assoziation des EObjectWrapper zu EObject stellt eine Referenz zu einer beliebigen Elementinstanz des EMF *Ecore* Meta-Metamodells dar. Dies ist auch der Grund für die Benennung der Metamodellelemente.

Somit ist es mit diesem Metamodell möglich, Modelle zu erstellen, die beliebige Strukturen von durchgeführten Modelltransformationen und deren Mappings erfassen.

4. Entwurf des Tracing-Frameworks

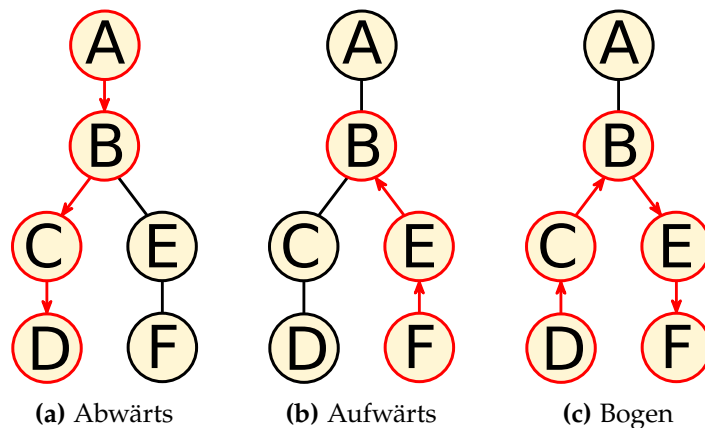


Abbildung 4.9. Möglichkeiten für Pfade durch einen Transformationsbaum

Um die Struktur eines solchen Modells zu verdeutlichen ist in Abbildung 4.8 ein Modell angegeben, welches das Mapping der Transformation des Modells A zu B erfasst hat. Es ist klar die Zweiteilung zu erkennen. Zunächst stellt der obere Bereich die transformierten Modelle dar. In der Mitte ist die Struktur der Transformationen zu erkennen, in diesem Fall ist es eine zweielementige Kette, jedoch ist hier auch eine Baumstruktur möglich. Dieser enthält auch die Repräsentation der Modelle A und B und der Transformation A-zu-B. Im unteren Teil ist das Mapping abgebildet. Es sind die Repräsentation der Elemente aus den Modellen A und B zu erkennen, sowie deren Relationen untereinander.

Transformationsbaum Da die entworfene Datenstruktur Transformationen und deren Mappings in der Form eines Baumes erfasst und darstellt, wird diese in der restlichen Arbeit als *Transformationsbaum* bezeichnet.

4.4. Datenaufbereitung

Der Kernpunkt der Datenaufbereitung ist das Zusammenführen von Mappings aus einem Transformationsbaum. Voraussetzung hierfür sind zunächst zwei Modelle, die beide im selben Transformationsbaum enthalten sein müssen. Es ist frei wählbar, welches der beiden Modelle das Quell- und welches das Zielmodell der zusammengeführten Mappings sein soll. In beiden Fällen ist es dann möglich einen gerichteten Pfad im Transformationsbaum zu finden. Die Abbildung 4.9 zeigt beispielhaft die drei Sorten solcher Pfade. Wobei zu beachten ist, dass diese Pfade auch gegen die Kantenrichtung im Baum verlaufen können, also aufwärts. Die verschiedenen dargestellten Pfade lassen sich durch Fragestellungen motivieren.

Pfad	Fragestellung
Abbildung 4.9a	Was ist aus den Elementen aus A in B entstanden?
Abbildung 4.9b	Aus welchen Elementen in B sind die Elemente in F entstanden?
Abbildung 4.9c	Welche Elemente in F sind entstehungsäquivalent zu den Elementen in D?

Dabei beschreibt die Fragestellung, was mit der Auswahl der Quell- und Zielmodelle des Pfades bezweckt werden soll. Die Antworten auf diese Fragen liefert das Mapping, das entsteht, wenn die einzelnen Mappings auf dem Pfad transitiv zusammengeführt werden.

Um ein solches Mapping zu erzeugen, muss man zunächst den Pfad bestimmen, der vom Quell- zum Zielmodell führt. Zu diesem Zweck wurde der Algorithmus 1 entwickelt, der den gewünschten Pfad ermittelt. Das Vorgehen ist dabei an die Pfade in Abbildung 4.9 angelehnt. Zunächst werden jeweils Pfade von Quell- und Zielknoten zur Wurzel des Baumes erstellt. Dann wird überprüft, ob es ein trivialer Pfad ist, also in einer Form wie in Abbildung 4.9a oder 4.9b. Wenn dies nicht der Fall ist, beinhaltet der Pfad eine Richtungsänderung im Baum, wie in Abbildung 4.9c. In diesem Fall wird der kleinste gemeinsame Oberknoten von Quell- und Zielknoten bestimmt, indem die Pfade auf gleiche Knoten abgeglichen werden. Anschließend werden die zwei Pfade an der ermittelten Stelle zusammengefügt. So kann für jede Auswahl von Modellen im Baum ein Pfad bestimmt werden.

Um anschließend aus dem Pfad die Mappings zusammenzuführen, wurde der Algorithmus 2 entwickelt. Dieser betrachtet der Reihe nach die einzelnen Mappings zwischen den Knoten im Pfad und fügt diese in das endgültige Mapping ein. Dabei werden beim Einfügen die Mappings nicht einfach ersetzt, sondern transitiv zusammengefügt. Abbildung 4.10 zeigt beispielhaft ein solches transitives Zusammenführen zweier Mappings. Aus A zu B und B zu C entsteht das Mapping A zu C.

4. Entwurf des Tracing-Frameworks

Algorithmus 1: Bestimmung eines Pfades im Baum

Eingabe :
Baum - Transformations-Baum mit Mappings
Quellmodell - Modellrepräsentation des Quellmodells als Knoten
Zielmodell - Modellrepräsentation des Zielmodells als Knoten

Daten : *Vorbedingung:*
Baum enthält Quellmodell und Zielmodell

Ausgabe : Pfad vom Quellmodell zum Zielmodell

- 1 Sei K ein Knoten im Baum
- 2 Sei QuellPfad ein leerer Pfad
- 3 Setze K zu Quellmodell
- 4 **solange** K hat Elternknoten **tue**
- 5 | Füge K an QuellPfad an
- 6 | Setze K auf Elternknoten von K
- 7 **Ende**
- 8 Sei ZielPfad ein leerer Pfad
- 9 Setze K zu Zielmodell
- 10 **solange** K hat Elternknoten **tue**
- 11 | Füge K an ZielPfad an
- 12 | Setze K auf Elternknoten von K
- 13 **Ende**
- 14 Sei EndPfad ein leerer Pfad
- 15 **wenn** Quellmodell liegt auf ZielPfad **dann**
- 16 | Verkürze ZielPfad bis Quellmodell
- 17 | Setze EndPfad zu ZielPfad in umgekehrter Richtung
- 18 **sonst wenn** Zielmodell liegt auf QuellPfad **dann**
- 19 | Verkürze QuellPfad bis Zielmodell
- 20 | Setze EndPfad zu QuellPfad
- 21 **sonst**
- 22 | Setze K als den ersten gemeinsamen Knoten von QuellPfad und ZielPfad
- 23 | Verkürze QuellPfad bis einschließlich K
- 24 | Verkürze ZielPfad bis ausschließlich K
- 25 | Setze EndPfad zu QuellPfad
- 26 | Hänge an EndPfad den ZielPfad in umgekehrter Richtung an
- 27 **Ende**

Algorithmus 2: Zusammenführen von Mappings

Eingabe :

Pfad - Pfad von einem Quellmodell zu einem Zielmodell

Baum - Transformations-Baum mit Mappings

Daten : *Vorbedingungen:*

Pfad hat mindestens die Länge 2

Pfad ist im Baum

Ausgabe : Map als Mapping zwischen Quellmodell und Zielmodell

```

1 Sei Map ein leeres Mapping
2 Sei MapQuelle das erste Modell im Pfad
3 Sei MapZiel das zweite Modell im Pfad
4 Fülle Map mit dem Mapping zwischen MapQuelle und MapZiel
5 solange MapZiel hat Nachfolger im Pfad tue
6   | Setze MapQuelle auf MapZiel
7   | Setze MapZiel auf Nachfolge von MapZiel im Pfad
8   | Sei ZwischenMap das Mapping zwischen MapQuelle und MapZiel
9   | für alle Relationen Rel in Map tue
10  | | Entferne Rel aus Map
11  | | Sei UrQuelle die erste Komponente (Quelle) in Rel
12  | | Sei Schlüssel die zweite Komponente (Ziel) in Rel
13  | | Sei NeueRels die Menge der Relationen in ZwischenMap mit
14  | | Schlüssel als erster Komponente (Quelle)
15  | | für alle Relationen NeuRel in NeueRels tue
16  | | | Ersetze erste Komponente (Quelle) von NeuRel durch UrQuelle
17  | | | Füge NeuRel in Map ein
18  | Ende
19 Ende

```

4. Entwurf des Tracing-Frameworks

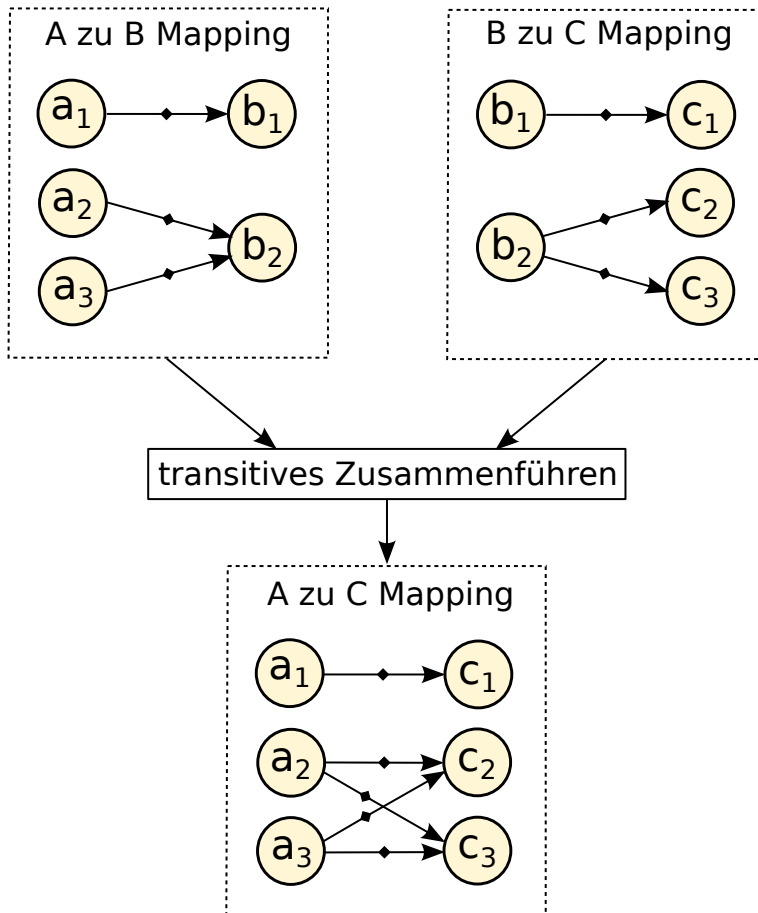


Abbildung 4.10. Beispiel für transitives Zusammenführen von Mappings

4.5. Visualisierung

Die entwickelte Datenstruktur ermöglicht zunächst nur das Erstellen eines Modells auf Basis des Metamodells. Der Nutzer kann zwar diesen Transformationsbaum programmatisch weiterverarbeiten, jedoch ist es auch eine Anforderung, eine graphische Darstellung bereitzustellen. Dies geschieht, wie im Anhang A der UML Spezifikation¹ beschrieben, durch Diagramme. Für die Baumstruktur des Transformationsbaumes ist es naheliegend, ein Knoten-Kanten-Diagramm zur Visualisierung zu nutzen, um so eine schnelle Übersicht über die enthaltenen Modelle und durchgeführten Transformationen zu geben.

Die Diagramme für Mappings können zunächst auch mit einem Knoten-Kanten-Diagramm dargestellt werden, wobei die Menge der Knoten die Elemente des

¹www.omg.org/spec/UML/2.5/Beta2/

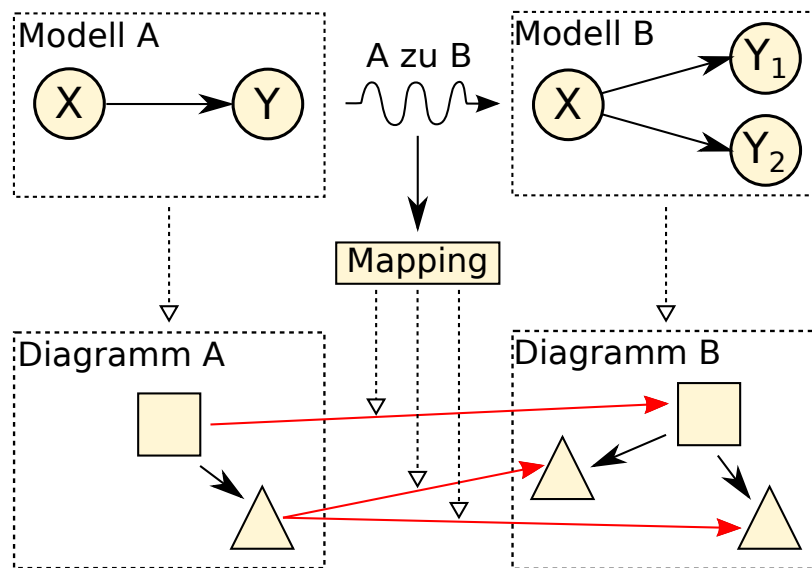


Abbildung 4.11. Konzeptionelle Darstellung der Visualisierung des Mappings

Quell- und Zielmodells enthält und Kanten die jeweiligen Relationen angeben. Dabei ist es wichtig, dass Elemente von Quell- und Zielmodell unterscheidbar sind, beispielsweise durch entsprechende Gruppierung. Wenn sowohl für das Quell- als auch das Zielmodell bereits eine eigene Synthese zu einem Diagramm besteht, kann diese sinnvollerweise genutzt werden. Dann werden beide Modelle nebeneinander angezeigt und die Relationen direkt durch hervorgehobene Pfeile markiert.

Dieses Prinzip ist angelehnt an die „Graphische Zuordnung von Elementen einer Modelltransformation“ von Stanislaw Nasin [Nas13].

Abbildung 4.11 zeigt wie eine solche Visualisierung der Relationen konzeptionell aussieht. Es sind die Modelle aus der Abbildung 4.8 zu erkennen, zusammen mit dem aus der Transformation entstandenen Mapping. Auf Basis dieses Mappings können nun die Relationen der Modellelemente in den Diagrammen als rote Pfeile dargestellt werden.

Um die Menge der angezeigten Informationen zu reduzieren, hat der Nutzer die Möglichkeit, die Mappings selektiv zu filtern und mit Rückgriff auf die Datenaufbereitung auch direkt zusammengeführte Mappings angezeigt zu bekommen.

Implementierung des Tracing-Frameworks

Aufbauend auf dem Entwurf des Frameworks aus Kapitel 4 wird in diesem Kapitel dessen Implementierung vorgestellt. Dabei werden die einzelnen Designentscheidungen aufgegriffen und ihre technische Umsetzung im KIELER Projekt erläutert. Zunächst wird in Abschnitt 5.1 die Plug-in Struktur des Frameworks erklärt, nachfolgend wird auf die Implementierung der Schnittstelle eingegangen und in Abschnitt 5.3 wird die erzeugte Datenstruktur und dessen Aufbereitung beschrieben. Zuletzt wird in Abschnitt 5.4 die Umsetzung der Visualisierung vorgestellt.

5.1. Struktur

Das Framework wurde in Java implementiert und mithilfe von Plug-ins in Eclipse integriert. Es wurden drei Plug-ins erstellt, die unterschiedliche Teile der Implementierung enthalten.

- (i) `de.cau.cs.kieler.ktm`
- (ii) `de.cau.cs.kieler.ktm.klighd`
- (iii) `de.cau.cs.kieler.ktm.test`

Der Aufbau und Inhalt dieser Plug-ins wird in den folgenden Unterabschnitten näher beschrieben. Zunächst wird auf das Kern-Plug-in (i) eingegangen, das die Implementierung der Schnittstelle, Datenstruktur und Aufbereitungsfunktionen enthält. Anschließend wird das Plug-in (ii), das die Visualisierung enthält, in Unterabschnitt 5.1.2 beschrieben und abschließend die Tests aus dem Plug-in (iii) erläutert.

5.1.1. Tracing Funktionen

Das Plug-in (i) enthält die Schnittstelle zum Erzeugen von Mappings. Diese ist in der Klasse `TransformationMapping` umgesetzt und in der Xtend Sprache programmiert, damit sie komfortabler in die Transformationen in KIELER eingebettet

5. Implementierung des Tracing-Frameworks

werden kann. In Abschnitt 5.2 wird näher auf die Details der Implementierung der Schnittstelle eingegangen.

Weiterhin enthält es die Implementierung der Datenstruktur basierend auf EMF, bezeichnet als `TransformationTree`. Als Erweiterung der Datenstruktur dient die Klasse `TransformationTreeExtensions`. Sie enthält Methoden zur Erstellung und Veränderung von Transformationsbäumen, sowie Möglichkeiten zur Aufbereitung der Daten. Die Details werden in Abschnitt 5.3 erläutert. Die `TransformationTreeExtensions` Klasse wurde ebenfalls in Xtend programmiert, da so eine komfortable Nutzung der EMF-API möglich ist.

5.1.2. Visualisierung

Im Plug-in (ii) ist die Visualisierung der Tracing-Informationen mithilfe von `KLighD` umgesetzt worden. Es enthält die Diagrammsynthese für den Transformationsbaum, sowie die Klassen zur Verarbeitung der diagrammspezifischen Nutzerinteraktionen. Weiterhin sind einige Hilfsklassen enthalten, die für die Synthese der Diagramme intern genutzt werden. In Abschnitt 5.4 werden die erzeugten Diagramme und Interaktionsmöglichkeiten näher beschrieben.

Die Visualisierung wurde in einem separaten Plug-in umgesetzt, da sie von den `KLighD` Plug-ins abhängig ist, die jedoch nicht für das Tracing notwendig sind. Daher wurde diese Struktur gewählt, damit es möglich ist, das Tracing mit dem Plug-in (i) zu nutzen, ohne `KLighD` in ein Projekt einbinden zu müssen.

5.1.3. Tests

Das Test-Plug-in (iii) enthält Testklassen, mit denen die Funktionalität aus dem Plug-in (i) überprüft wird.

Für die `TransformationMapping` und `TransformationTreeExtensions` Klassen wurden JUnit-Tests angelegt, die jede Methode anhand eines minimalen Testmodells auf ihr erwartetes Verhalten überprüft. Es sind Klassen für minimale Transformationen mit eingebetteter Erstellung von Mappings enthalten, sowie Hilfsklassen die Testmodelle erstellen.

Weiterhin ist eine Testklasse vorhanden, die den Einsatz der Frameworks im Anwendungsfall der `SCCharts` überprüft. Dazu wurde die Mappingschnittstelle in einige Transformationen der `SCCharts` eingebettet und damit ein Test-`SCChart` schrittweise transformiert. Gleichzeitig wurde der dazugehörige Transformationsbaum generiert, der die entsprechenden Tracing-Informationen enthält. Die Artefakte, die aus diesem Experiment entstanden sind, werden in Kapitel 6 evaluiert.

5.2. Schnittstelle zum Erfassen von Mappings

Die `TransformationMapping` Klasse ist die Umsetzung der Schnittstelle aus Abschnitt 4.2 des Entwurfs. Dort wurden die folgenden Anforderungen an die Funktionalität festgelegt.

- Einfügen einer Eins-zu-Eins-Entstehungsbeziehung zwischen Elementen
- Einfügen einer Eins-zu-Viele-Entstehungsbeziehung zwischen Elementen
- Automatische Erstellung von Eins-zu-Eins-Beziehungen zu einer Menge von Elementen (z.B. beim Kopieren)
- Entfernen einer Entstehungsbeziehung
- Auslesen der Relationen zu einem Element

Abbildung 5.1 zeigt die wichtigsten Methoden der `TransformationMapping` Klasse. Die Methoden deren Methodenname mit „map“ beginnt, ermöglichen das Einfügen von Relationen, wobei die Benennung an den Eltern- und Kind-Part in einer Entstehungsbeziehung angelegt ist. Mit `mappedParents` und `mappedChildren` können die Relationen zu einem Element ausgelesen werden und mit `unmap` einzelne oder mehrere Beziehungen aus dem Mapping entfernt werden. So wird es möglich, auch bereits registrierte Relationen nachträglich zu verändern. Dies ist insbesondere sinnvoll, wenn zuerst mit `mappedCopy` eine Kopie eines Modells erzeugt wurde, wobei automatisch die Relationen zwischen Original und Kopie in das Mapping eingetragen werden und dieses dann verändert wird.

Alle diese Methoden setzen eine dynamische Laufzeitdatenstruktur für das Mapping voraus. Die Mappingstruktur im Transformationsbaum ist für diesen Zweck zwar geeignet, jedoch nicht effizient genug, daher wird eine `HashMultiMap` aus der Guava Bibliothek verwendet. Größtenteils bildet die Klasse `TransformationMapping` somit einen Wrapper, um die `HashMultiMap`, der an die Besonderheiten beim Ein-

```

● mapParent(EObject, EObject) : boolean
● mapParents(EObject, List<EObject>) : boolean
● mapChild(EObject, EObject) : boolean
● mapChildren(EObject, List<EObject>) : boolean
● mappedChildren(EObject) : List<EObject>
● mappedParents(EObject) : List<EObject>
● unmap(EObject, EObject) : boolean
● unmapAll(EObject) : boolean
● mappedCopy(T) : T
● extractMappingData() : ImmutableMultimap<EObject, EObject>
● checkMappingCompleteness(EObject, EObject) : Pair<ImmutableSet<EObject>, ImmutableSet<EObject>>

```

Abbildung 5.1. Auszug aus Methoden der `TransformationMapping` Klasse

5. Implementierung des Tracing-Frameworks

satz als Modelltransformationsmapping angepasst ist. Eine solche MultiMap kann am Ende einer Transformation mit `extractMappingData` extrahiert werden und enthält dann sämtliche während der Transformation erfassten Relationen, die anschließend in den Transformationsbaum eingefügt werden können.

Weiterhin erfüllt die Funktion `checkMappingCompleteness`, die Anforderung an eine Überprüfungsfunktion für die Vollständigkeit des Mappings, indem sie die symmetrischen Differenzen der Menge der Elemente im Quell- beziehungsweise Zielmodell zu der Menge der Elemente in den entsprechenden Relationskomponenten bildet. So können Abweichungen und fehlende Elemente erkannt werden.

5.3. Transformationsbaum als Tracing-Struktur

Die in Abschnitt 4.3 entworfene Datenstruktur wurde mithilfe von EMF implementiert. Zunächst wurde das Metamodell, bekannt aus Abbildung 4.7, mit *Ecore* modelliert. Anschließend wurde mit der EMF Codegenerierung eine Java-Implementierung erzeugt, die das Erzeugen von Modellen in Form eines Transformationsbaumes erlaubt.

5.3.1. Erweiterte Funktionalität des Transformationsbaumes

Um den Umgang mit der Datenstruktur zu erleichtern, dient die Klasse `TransformationTreeExtensions`. Ein Auszug der Methoden dieser Klasse ist in Abbildung 5.2 dargestellt. Diese erweitert das Metamodell des Transformationsbaumes nach dem Prinzip der *extensions* von Xtend, um einen besseren Zugriff zu ermöglichen. Sie bietet eine Reihe von Funktionen zur Navigation in der Baumstruktur,

- `TransformationTreeExtensions`
 - `root(ModelWrapper) : ModelWrapper`
 - `parent(ModelWrapper) : ModelWrapper`
 - `children(ModelWrapper) : List<ModelWrapper>`
 - `succeedingTransformations(ModelWrapper) : List<ModelTransformation>`
 - `succeedingModelWrappers(ModelWrapper) : List<ModelWrapper>`
 - `depth(ModelWrapper) : int`

 - `initializeTransformationTree(...) : ModelWrapper`
 - `addTransformationToTree(...) : ModelWrapper`
 - `removeModelFromTree(ModelWrapper) : ModelWrapper`
 - `makeTransient(ModelWrapper) : ModelWrapper`

 - `findModelInTree(ModelWrapper, EObject, String) : ModelWrapper`
 - `joinMappings(ModelWrapper, EObject, ModelWrapper, EObject) : Multimap<EObject, EObject>`

Abbildung 5.2. Auszug aus Methoden der `TransformationTreeExtensions` Klasse

5.3. Transformationsbaum als Tracing-Struktur

beispielsweise Methoden zur Bestimmung von Kindknoten oder der Wurzel des Baumes.

Weiterhin ist mit dieser Klasse eine komfortable Erstellung und Veränderung eines Transformationsbaumes möglich. So können transformierte Modelle zusammen mit ihren Mappings direkt in den Baum eingefügt oder entfernt werden, ohne den Transformationsbaum über die generierte EMF-API verändern zu müssen. Dabei wird automatisch das Mapping in Form einer MultiMap in die Modellstruktur übernommen. Außerdem wird auch geprüft, ob das Modell an der korrekten Stelle eingefügt wurde, sodass sichergestellt wird, dass eine Transformationskette nicht unterbrochen wird.

Repräsentation von Modellen

Im Metamodell aus Abbildung 4.7 ist definiert, dass EObjectWrapper Elemente in einem Modell repräsentieren, indem sie eine Referenz auf dessen Instanz (EObject) besitzen. Es ist davon auszugehen, dass Quellmodelle nach ihrer Transformation noch verändert oder angepasst werden. Wenn im Transformationsbaum Referenzen auf eine Modellinstanz, die verändert wird, existieren, hätte dies zufolge, dass die EObjectWrapper nicht mehr ihre ursprünglichen Elemente repräsentieren und das Mapping damit fehlerhaft wäre. Aus diesem Grund wird beim Einfügen eines Modells in den Transformationsbaum nicht die Originalinstanz verwendet, sondern eine Kopie erzeugt. So kann sichergestellt werden, dass ein Mapping für ein Modell innerhalb des Transformationsbaumes immer konsistent bleibt, auch wenn das Quellmodell geändert wird. Zudem würde eine Änderung des Quellmodells sowieso eine erneute Transformation nach sich ziehen, woraus auch ein neuer Transformationsbaum entstehen würde.

Da die erzeugte Kopie durch die Referenzen an den Transformationsbaum gebunden ist, erhöht dies die Größe einer Modellinstanz des Transformationsbaumes deutlich. Um einem Nutzer die Möglichkeit zu geben die Größe bei Bedarf zu reduzieren, können einzelne Modelle als transient markiert werden. Die Klasse TransformationTreeExtensions besitzt dafür die entsprechende Methode makeTransient und das Metamodell sieht das Attribut transient im ModelWrapper als Markierung für solche Modelle vor. Die EObjectWrapper eines solchen flüchtigen Modells, besitzen dann keine Referenzen mehr auf die EObject Instanzen und die interne Kopie des Modells wird gelöscht. Daher sind diese dann keine geeigneten Repräsentationen der Modellelemente mehr, da sie ihnen nicht mehr zuzuordnen sind. Auf diese Weise kann die Größe einer konkreten Modellinstanz des Transformationsbaumes reduziert werden, ohne das interne Mappings verloren gehen. Da keine Reidentifikation transienter Modellrepräsentationen mit ihren Modellen möglich ist, schränkt dies die Zugriffs- und Verarbeitungsmöglichkeiten ein, weshalb nur unwichtige Zwischenmodelle als transient gesetzt werden sollten.

5. Implementierung des Tracing-Frameworks

Zusammenführen von Mappings

Weiterhin enthält die `TransformationTreeExtensions` Klasse auch die Methoden zur Aufbereitung des Mappings, bekannt aus Abschnitt 4.4. Diese kann in einer Anwendung, die zum Tracing das Framework verwendet, eingesetzt werden, um ein Mapping zwischen zwei beliebigen Modellen im Transformationsbaum zu erzeugen.

Zur Umsetzung des transitiven Zusammenführens von Mappings wurden in der Methode `joinMappings` die Algorithmen 1 und 2 in Xtend implementiert. Da diese die Mappings im Transformationsbaum verarbeiten, ist das Resultat ein Mapping aus `EObjectWrapper` mit Verweisen auf die internen Kopien der Modelle. Ein Nutzer kann mit einem solchen Mapping wenig anfangen, da ihm eine andere Modellinstanz zur Verfügung steht. Deshalb überträgt die Implementierung zusätzlich noch die Mappinginformationen auf die Modellinstanz des Nutzers und gibt so ein Mapping aus `EObject` zurück, was komfortabler zu nutzen ist. Im Fall, dass Ziel- oder Endmodell im Transformationsbaum als transient markiert sind, ist ein solches Übertragen nicht möglich. Ein transientes Modell in einem Zwischenmapping beeinflusst den Prozess des Zusammenführens jedoch nicht.

Die Methode `joinMappings` setzt voraus, dass die Repräsentationen des Ziel- oder Endmodells im Transformationsbaum bekannt sind. Da dem Nutzer jedoch nur eigene Modellinstanzen und der Baum als ganzes zur Verfügung stehen, wird zusätzlich die Funktion `findModelInTree` zum Finden von Modellrepräsentationen bereitgestellt. Diese findet anhand der `TypID` des Modells und der Modellinstanz den zugehörigen `ModelWrapper` im Transformationsbaum. Dies wird über einen strukturellen Vergleich mittels EMF Compare erreicht, bei dem die Modellinstanz des Nutzers mit den internen Kopien der Modelle im Transformationsbaum abgeglichen werden. Für transiente Modelle ist ein solches Auffinden nicht möglich.

5.4. Visualisierung der Tracing-Informationen

Die Visualisierung wurde mithilfe des `KLighD` Frameworks implementiert. Dabei wurden die Konzepte aus Abschnitt 4.5 umgesetzt.

Dem Nutzer stehen drei Diagramm-Optionen zur Verfügung, um die angezeigten Diagramme zu beeinflussen.

- Model visualization
- EObject attributes
- Selective mapping edges

5.4. Visualisierung der Tracing-Informationen

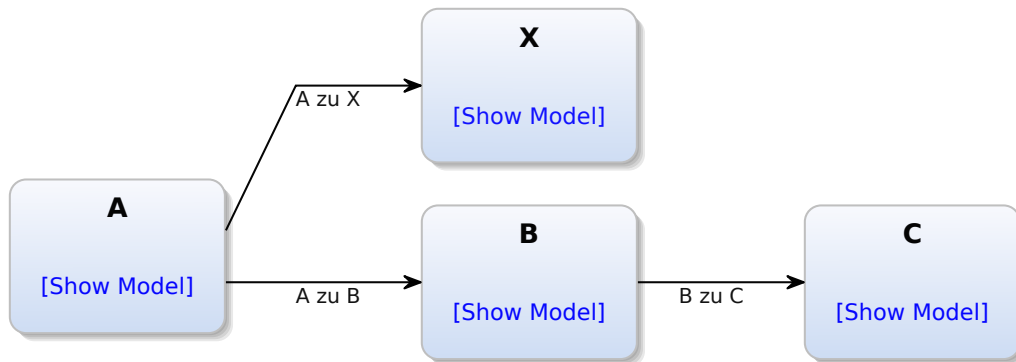


Abbildung 5.3. Diagramm eines Transformationsbaumes

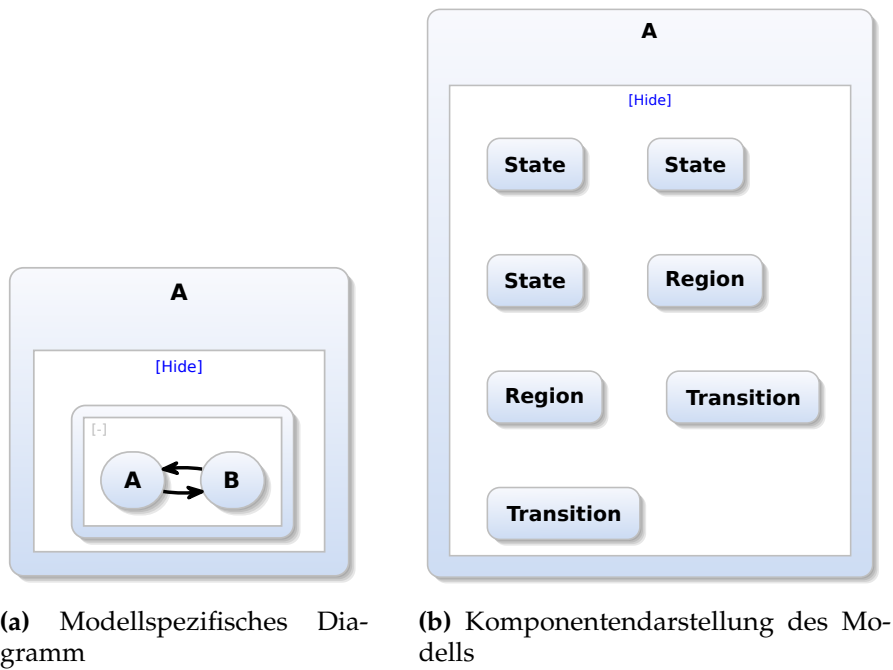
Zunächst wird in der Visualisierung eines Transformationsbaumes ein einfaches Knoten-Kanten-Diagramm angezeigt, das durch eine Diagrammsynthese mit KLightD erzeugt wird. Es bietet eine schnelle Übersicht über die durchgeführten Transformationen, die erfasst wurden. Abbildung 5.3 zeigt ein solches Diagramm eines Transformationsbaumes. Es ist zu erkennen, dass eine Kette von Transformationen vom Quellmodell A zu Modell D durchgeführt wurde, sowie eine zusätzliche Transformation die A zu X transformiert hat.

Weiterhin bietet das Diagramm die Möglichkeit, die einzelnen Modelle im Baum ebenfalls mit anzuzeigen. Hierzu kann der untere Bereich im Knoten einer Modellrepräsentation aufgeklappt werden. Wenn die Option Model visualisation gesetzt ist, wird hier das Diagramm des enthaltenen Modells angezeigt. Abbildung 5.4a zeigt eine solche modellspezifische Visualisierung des enthaltenen Modells.

Sollte keine Diagrammsynthese zur Verfügung stehen, das Modell transient sein oder die Option nicht gesetzt sein, wird auf eine Komponentendarstellung zurückgegriffen. Diese zeigt die einzelnen Modellelemente ungeordnet als Knoten an, wie es in Abbildung 5.4b dargestellt ist. Es ist zu erkennen, dass die Knoten die Klassenbezeichnungen der Modellelemente tragen. Dies ist wenig aussagekräftig, da diese öfter auftreten können. Aus diesem Grund kann mit der Option EObject attributes auch die Attribute der Modellelemente angezeigt werden, was zu einer aussagekräftigeren Darstellung, wie in Abbildung 5.4c führt.

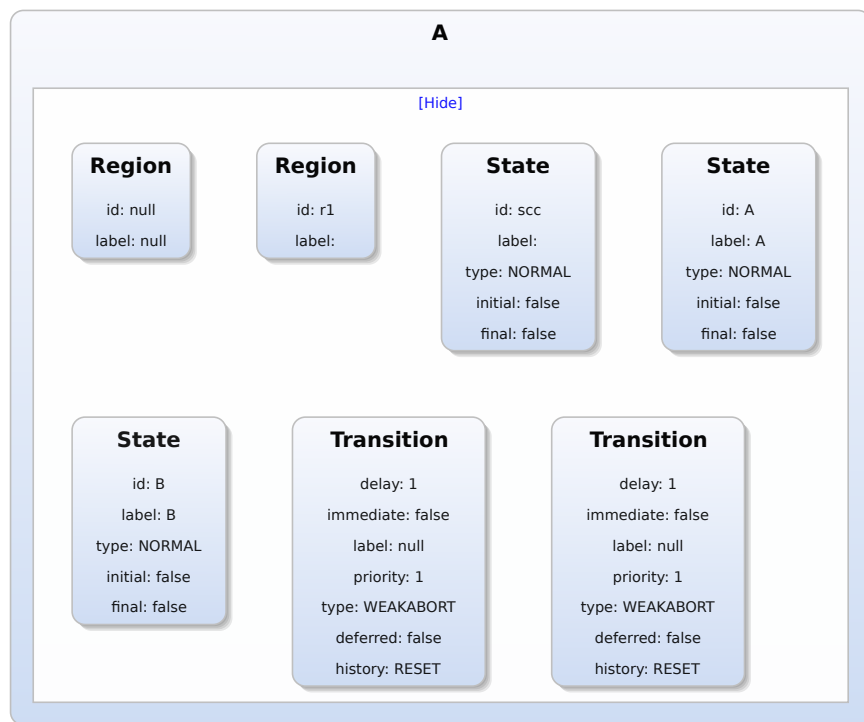
Anhand des Diagramms kann, durch Anwählen der Knoten, ein Quell- und Zielmodell ausgewählt werden, zwischen denen ein Mapping angezeigt wird. Hierzu wurde ein entsprechender ActionListener registriert, der diese Ereignisse verarbeiten und das Diagramm anpasst. Sollten die Modellknoten nicht benachbart sein, wird automatisch ein transitiv zusammengeführtes Mapping berechnet.

5. Implementierung des Tracing-Frameworks



(a) Modellspezifisches Diagramm

(b) Komponentendarstellung des Modells



(c) Komponentendarstellung des Modells mit Attributen

Abbildung 5.4. Visualisierung enthaltener Modelle im Transformationsbaum

5.4. Visualisierung der Tracing-Informationen

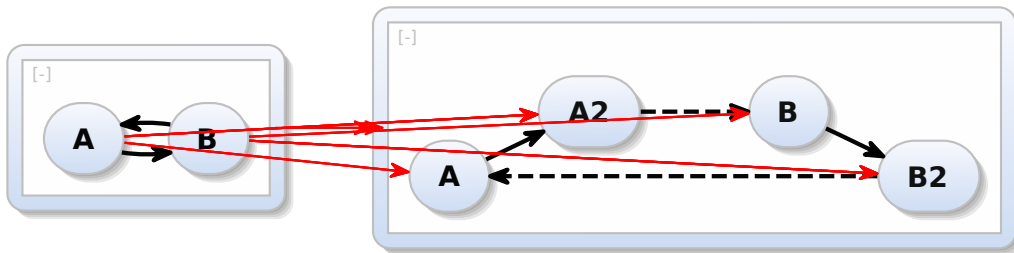


Abbildung 5.5. Mapping zwischen Modellen in vereinfachter Diagrammform

Das Resultat wird dann im Transformationsbaum angezeigt, wobei sowohl die Komponentendarstellung, als auch eine Visualisierung mit der modellspezifischen Synthese möglich ist. Die Beziehungen aus dem Mapping werden dabei durch Pfeile zwischen den Diagrammelementen dargestellt. Abbildung 5.5 zeigt in vereinfachter Form ohne Transformationsbaum, eine solche Visualisierung eines Mappings zwischen zwei Modellen. In diesem Fall ist das rechte Modell durch eine fiktive Transformation entstanden, die A und B aufgespalten hat.

Momentan werden von KLightD nur Pfeile zwischen Modellelementen, die als Knoten dargestellt werden, unterstützt. Weiterhin ist zu erkennen, dass sich die Pfeile kreuzen können und das Mapping somit unübersichtlich wird. Um dem entgegen zu wirken, gibt es die Option *Selective mapping edges*. Diese blendet alle Pfeile zunächst aus und zeigt nur jene an, die zu Elementen gehören, die durch den Nutzer angewählt wurden.

Ergebnisse der Experimente und Evaluation

Dieses Kapitel stellt die Ergebnisse vor, die aus der experimentellen Nutzung des Tracing-Frameworks gewonnen wurden. Dabei werden die verschiedenen Aspekte des Frameworks betrachtet und evaluiert. Basis dieser Ergebnisse ist ein Experiment, das beispielhaft an SCCharts durchgeführt wurde.

6.1. Einbettung der Schnittstelle für Mappings

Um die Anwendbarkeit des Tracing-Frameworks zu überprüfen, bedarf es Modelltransformationen, die ein Mapping produzieren. Aus diesem Grund wurde die Schnittstelle zum Erzeugen der Mappings (TransformationMapping) in ausgewählte SCCharts Transformationen eingebettet.

Dies sind TriggerEffect und SurfaceDepth, mit denen Core SCCharts normalisiert werden und SCC2SCG, die einen Normalized Core SCChart in einen SCG transformiert.

Diese Modelltransformationen wurden ausgewählt, da sie einerseits eine sinnvolle Transformationskette von Core SCCharts nach SCG bilden, die dem Vorgehen bei der Kompilierung von SCCharts aus Abbildung 1.4 entspricht. Andererseits werden verschiedene Bereiche abgedeckt, die in Abschnitt 4.1 als Anforderungen definiert wurden, so kann getestet werden, ob diese auch erfüllt sind. TriggerEffect ist eine Modelltransformation von SCCharts nach SCCharts, die auf dem selben Metamodell bleibt und dabei die Strategie des kopiebasierten Änderns nutzt, um das Modell umzuwandeln. Listing 6.1 zeigt einen Ausschnitt aus der Implementierung der Transformation.

```

1 def Region transformTriggerEffect(Region rootRegion) {
2   clearMapping; //NEU
3
4   // Clone the complete SCCharts region
5   var targetRootRegion = rootRegion.mappedCopy.fixAllPriorities; //MODIFIZIERT
6   //ZUVOR: var targetRootRegion = rootRegion.copy.fixAllPriorities;
7
8   // Traverse all transitions
9   for (targetTransition : targetRootRegion.getAllContainedTransitions) {
10    targetTransition.transformTriggerEffect(targetRootRegion);

```

6. Ergebnisse der Experimente und Evaluation

```
11     }
12     targetRootRegion;
13 }
14 def void transformTriggerEffect(Transition transition, Region targetRootRegion) {
15     ...
16     for (effect : transition.effects.immutableCopy) {
17         ... // split trigger and effects into single states
18         val effectState = parentRegion.createState(GENERATED_PREFIX + "S")
19         effectState.mapParents(transition.mappedParents); //NEU
20         effectState.uniqueName
21         val effectTransition = createImmediateTransition.addEffect(effect)
22         effectTransition.mapParents(transition.mappedParents); //NEU
23
24         effectTransition.setSourceState(effectState)
25         lastTransition.setTargetState(effectState)
26         lastTransition = effectTransition
27     }
28     ...
29 }
30 }
```

Listing 6.1. Auszug aus TriggerEffect Transformation

Die SCC2SCG Transformation erzeugt hingegen einen SCG, der auf einem anderen Metamodell basiert als die SCCharts. Zudem wird der SCG nach dem Prinzip der inkrementellen Erstellung erzeugt, also zu jedem SCCharts-Element wird ein entsprechendes SCG-Element erzeugt. Einige Code-Auszüge der Transformation sind in Listing 6.2 aufgeführt.

```
1 def SCGraph transformSCG(Region rootRegion) {
2     clearMapping; //NEU
3     ...
4     // Create a new SCGraph
5     val sCGraph = ScgFactory::eINSTANCE.createSCGraph;
6     rootRegion.mapChild(sCGraph); //NEU
7     rootRegion.rootState.mapChild(sCGraph); //NEU
8     ...
9     // Generate nodes and recursively traverse model
10    for (region : rootRegion.rootState.regions) {
11        region.transformSCGGenerateNodes(sCGraph);
12    }
13    ...
14    // Fix superfluous exit nodes
15    sCGraph.trimExitNodes.trimConditionNodes;
16
17    //val completeness = checkMappingCompleteness(rootRegion, sCGraph); //NEU (Fehlersuche)
18    return sCGraph;
19 }
20
21 ...
22 }
```

6.1. Einbettung der Schnittstelle für Mappings

```
23 // If two exit nodes follow each other, remove the first one.
24 def SCGraph trimExitNodes(SCGraph sCGraph) {
25     val exitNodes = sCGraph.eAllContents.filter(typeof(Exit)).toList;
26     val superfluousExitNodes = exitNodes.filter(e|e.next != null && e.next.target
27         instanceof Exit).toList;
28     for (exitNode : superfluousExitNodes.immutableCopy) {
29         ...
30         exitNode.next.target.mapParents(exitNode.mappedParents); //NEU
31         ...
32         sCGraph.nodes.remove(exitNode);
33         exitNode.unmapAll; //NEU
34     }
35     sCGraph
36 }
```

Listing 6.2. Auszug aus SCC2SCG Transformation

6.1.1. Evaluation

Es ist gelungen, eine minimalinvasive Schnittstelle zu definieren, die sich auch bei verschiedenen Vorgehensweisen nutzen lässt, wie beispielsweise in Listing 6.1 zu erkennen ist. Initial wird in Zeile 2 sichergestellt, dass die Transformation mit einem leeren Mapping startet. In Zeile 5 wurde ein einzelner Befehl angepasst, sodass beim Kopieren direkt ein Mapping zwischen Original und Kopie registriert wird. Nachfolgend werden dann, bei der Änderung des Zielmodells, wie dem Hinzufügen neuer Elemente, das Mapping um die neuen Relationen erweitert. In diesem Fall werden in den Zeilen 19 und 22 zu den neuen Elementen die gleichen Eltern eingetragen, wie bei den Elementen, aus denen sie abgespalten wurden.

Beim iterativen Erstellen, wie in Zeile 6 in Listing 6.2, reicht es aus, die neu erstellten Elemente direkt als Kinder ihrer Quelle einzutragen.

Jedoch hat sich beim Einbetten der Mappinggenerierung gezeigt, dass es nicht immer leicht ist, alle Stellen im Code ausfindig zu machen, die das Mapping beeinflussen. Insbesondere da die Methode `checkMappingCompleteness` aus Zeile 17 in Listing 6.2 nur die fehlerhaft gemappten Modellelemente auflistet und nicht angibt, wo sie entstanden sind.

Hier würde eine stärker automatisierte Erfassung, wie im Ansatz von Amar et al. [Ama+08], Zeit bei der Implementierung und Fehlersuche sparen. Jedoch ist es fraglich, ob ein Tracing Framework, das nur EMF-API Aufrufe analysiert, komplexe Zusammenhänge wie das Mapping in Zeile 29 des Listings 6.2 erfassen könnte. Dort werden zwei redundante ExitNodes zusammengefasst und die Beziehungen des Einen auf den Anderen übertragen, bevor er gelöscht wird. Hierbei zeigt sich der Vorteil, dass die Semantik des Mappings durch die Art der Registrierung, also durch den Nutzer, festgelegt wird.

6. Ergebnisse der Experimente und Evaluation

Dennoch bleibt das Problem, dass der Transformationscode durch die Instruktionen der Mappinggenerierung 'verschmutzt' wird. Bei der Nutzung von Xtend besteht die Möglichkeit die Erstellung der Mappings, in die *extensions* der transformierten Modelle zu verlagern, was durch den flexiblen Aufbau der Schnittstelle ermöglicht und unterstützt wird, um so den eigentlichen Transformationscode 'sauber' und verständlich zu halten. Es wäre beispielsweise möglich, die Registrierung der Relation in Zeile 19 in den darüberliegenden *extension* Aufruf `createState` zu verlegen.

6.2. Tracing der ABO Transformation

Aufbauend auf den Modelltransformationen aus Abschnitt 6.1, wurde das Tracing anhand einer Beispieltransformation eines SCCharts getestet.

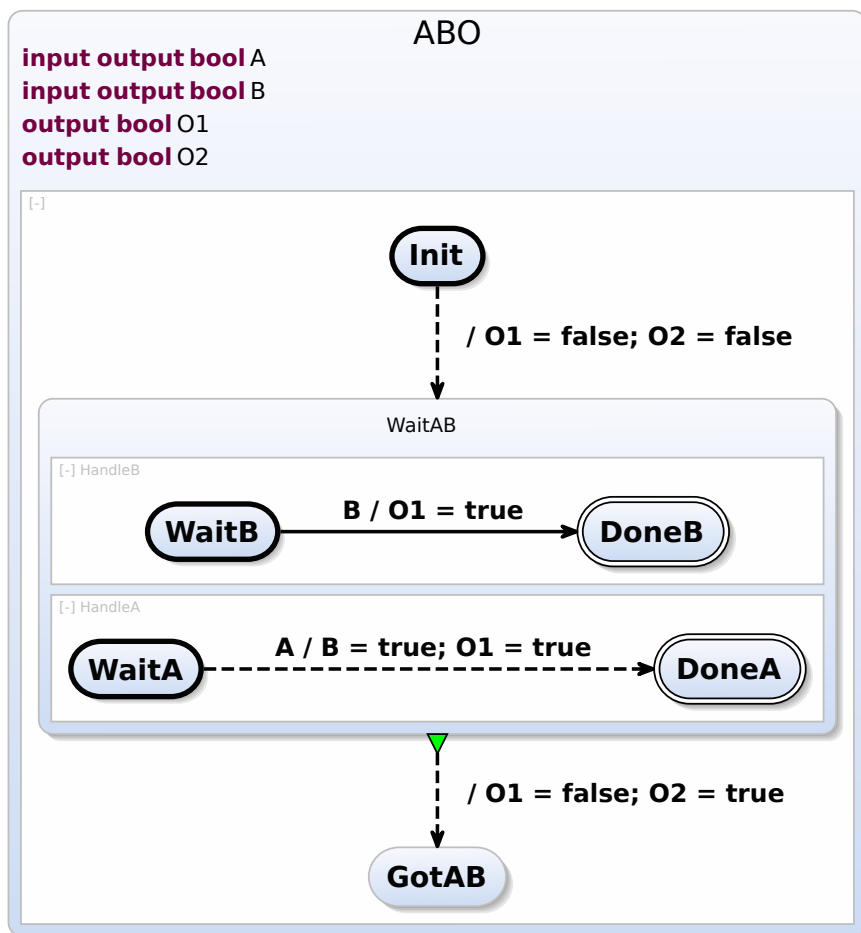


Abbildung 6.1. ABO SCChart

6.2. Tracing der ABO Transformation

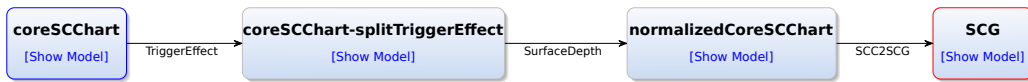


Abbildung 6.2. Transformationsbaum der ABO Transformation

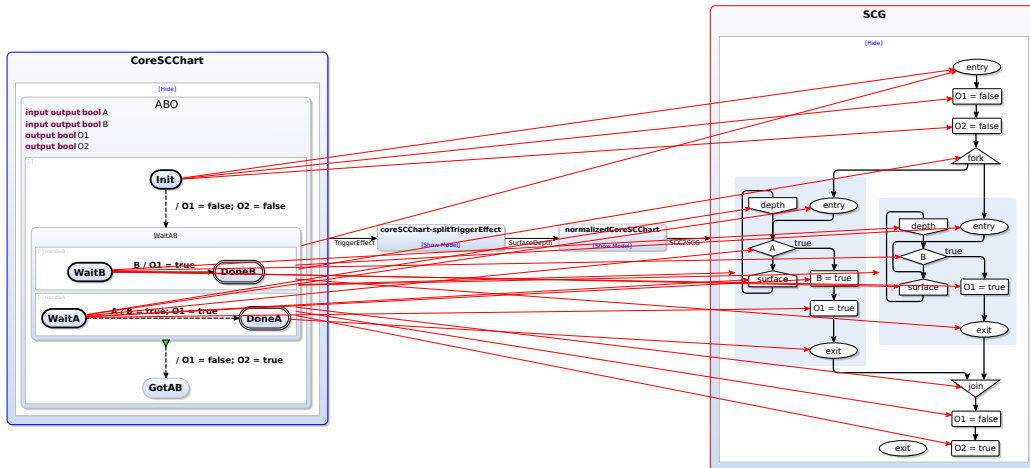


Abbildung 6.3. Mapping zwischen ABO als SCChart und SCG

Hierzu wurde der SCChart ABO, dargestellt in Abbildung 6.1, verwendet. Er ist das „Hallo Welt!“ der SCCharts, indem er ein minimales Beispiel für das sequenziell-konstruktive MoC der SCCharts darstellt.

Da ABO bereits ein Core SCChart ist, war es möglich ihn direkt mit der bestehenden Transformationskette in einen SCG zu transformieren. Dabei wurde nach jeder Transformation das Zwischenmodell und das Mapping in den Transformationsbaum eingetragen. Das Resultat zeigt die Abbildung 6.2.

Nun ist es auch möglich ein Mapping vom initialen ABO Core SCChart, über die Normalisierung hinweg, zum SCG zu ermitteln. Abgesehen von dem programmatischen Zugriff auf diese Tracing-Informationen, können diese auch visualisiert werden. In Abbildung 6.2 ist zu erkennen, dass das Quellmodell blau und das Zielmodell rot markiert ist. Abbildung 6.3 zeigt das Mapping zwischen den beiden Modellen. Es wird deutlich, aus welchen Teilen im SCChart, die Elemente im SCG entstanden sind, auch wenn im mittleren Teil die Pfeile sehr unübersichtlich werden. Hier schafft jedoch die Option zur selektiven Anzeige Abhilfe.

Weiterhin ist es hinderlich, dass nicht das vollständige Mapping angezeigt wird, sondern nur die Pfeile angezeigt werden können, die zwischen Diagrammknoten existieren. Dies verschleiert eine Reihe von Beziehungen, die eigentlich im Mapping enthalten sind.

6. Ergebnisse der Experimente und Evaluation

Abschließend ist zu bemerken, dass es möglich war, mit dem entwickelten Tracing-Framework erfolgreich eine Ablaufverfolgung für das ABO Beispiel von Core SCCharts zu SCG zu schaffen, was auch auf andere SCCharts und Transformationen übertragbar ist. Die Visualisierung trägt dabei deutlich zum Verständnis der durchgeführten Modelltransformationen und erfassten Entstehungsbeziehungen bei.

6.3. Performance

Anhand der Transformationskette aus Abschnitt 6.2 wurden Zeitmessungen durchgeführt, um zu ermitteln, wie das Framework die Performance des Systems beeinflusst, in dem es zum Tracing eingesetzt wird.

Um den zusätzlichen Zeitverbrauch durch die Tracing-Komponenten zu ermitteln, wurde jeweils der Zeitbedarf der einzelnen Schritte in der Transformationsabfolge aus Abschnitt 6.2 gemessen. Der Transformationsverlauf baut sich dabei wie folgt auf:

1. Transformation des ABO-Modells mit TriggerEffect zu ABO_{TE}
2. Initiale Erstellung des Transformationsbaumes mit ABO, ABO_{TE} und dem entstandenen Mapping
3. Umwandlung von ABO_{TE} in ABO_{TE+SD} durch die SurfaceDepth Transformation.
4. Anfügen von ABO_{TE+SD} mit Mapping an den Transformationsbaum
5. Transformation des normalisierten ABO_{TE+SD} in einen SCG (ABO_{SCG}), mittels SCC2SCG
6. Anfügen von ABO_{SCG} mit Mapping an den Transformationsbaum

Diese Transformationskette wurde zu Messzwecken um die entsprechenden Zeitabfragen erweitert und mehrfach ausgeführt, um durchschnittliche Zeitwerte zu erhalten. Die Zeitmessungen wurden mit Abfragen an die Systemzeit im Javacode umgesetzt und unterliegen somit den natürlichen Verzögerungen bei nebenläufigen Systemen, wie beispielsweise durch den Java Garbage Collector.

6.3.1. Zeitverbrauch durch Mappinggenerierung

Wie bereits in Abschnitt 6.1 beschrieben, wurden einige SCCharts Transformationen um die Schnittstelle zur Mappinggenerierung erweitert. Mithilfe des ABO SCChart wurden zuerst mehrfach die Modelltransformationen mit Mappinggenerierung und anschließend die alten Transformationen ohne Mapping durchgeführt.

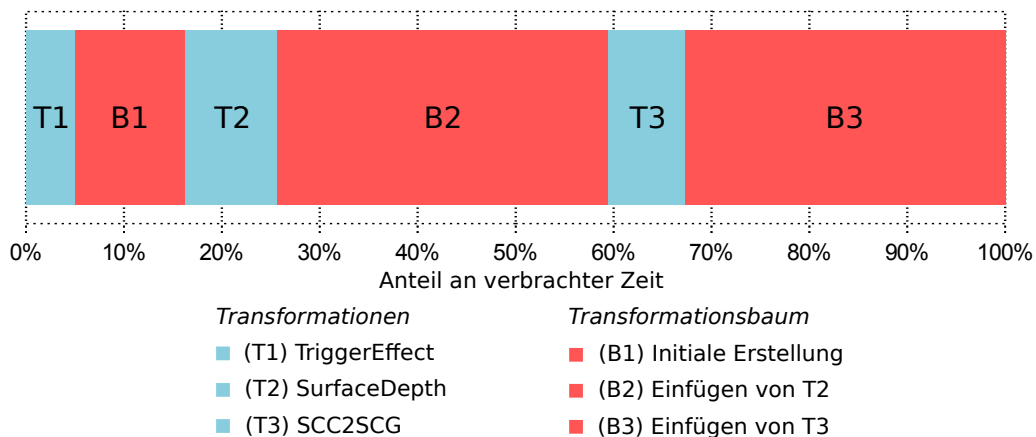


Abbildung 6.4. Verteilung des Zeitverbrauchs bei den ABO Transformationen

Die Ergebnisse der Zeitmessungen zeigten trotz starker Abweichungen, dass durch die zusätzlichen Instruktionen zur Erstellung des Mappings, die Transformationen 10 – 30% langsamer wurden.

Dies ist primär auf die Datenstruktur in der TransformationMapping Klasse zurückzuführen. Die HashMultiMap ermöglicht ein Ausführen der wichtigsten Operationen, wie Einfügen, Auslesen und Entfernen, in konstanter Zeit, da sie ihrerseits auf einer HashMap< K, HashSet<V> > basiert. Da Guava leider keine bidirektionale MultiMap mit konstanten Zugriffszeiten bietet, sind jedoch einige Zugriffe nur in $O(n)$, mit n als Anzahl der Einträge in der Map, möglich. Hierzu gehören Zugriffe, wie mappedParents, unmap und unmapAll, da bei diesen die Schlüssel (K) zu Werten (V) erfragt werden und dabei über alle Einträge iteriert werden muss, um diese zu ermitteln. Weiterhin wird die Ausführung verzögert, wenn die aktuelle Kapazität der HashMultiMap überschritten wird und der Speicher erweitert wird, um die Kapazität zu erhöhen. Dabei tritt ein sogenanntes *rehashing* auf, das zusätzlich Zeit verbraucht.

6.3.2. Zeitverteilung bei Transformationsketten

Die Ergebnisse der Zeitmessungen der einzelnen Transformationsschritte mit Mappinggenerierung und Baumerstellung sind in Abbildung 6.4 dargestellt. Sie zeigt die Anteile der einzelnen Schritte am Zeitverbrauch der gesamten Transformationskette.

Es ist deutlich zu erkennen, dass ein Großteil der Zeit (ca. 80%) durch die Erstellung und Erweiterung des Transformationsbaumes verbraucht wird. Dies liegt an dem Kopieren des Modells für den Transformationsbaum und dem Übertragen des Mappings aus der MultiMap in die Modellstruktur des Transformationsbaumes.

6. Ergebnisse der Experimente und Evaluation

Entscheidender ist jedoch der Einsatz von EMF Compare beim Einfügen neuer Modelle in den Baum. Hierbei wird zunächst ein struktureller Vergleich gemacht, um zu überprüfen, ob das Quellmodell des einzufügenden Mappings auch mit dem Modell übereinstimmt, an dessen Repräsentation es angehängt werden soll. Zudem wird so auch ein Mapping erzeugt, das es ermöglicht, die Relationen aus dem Mapping der Transformation auf das Modell im Baum zu übertragen. Weiterhin erklärt die Nutzung von EMF Compare auch den zeitlichen Unterschied zwischen der Erstellung des Baumes (B1), bei der kein Vergleich notwendig ist und dem Einfügen (B2,B3).

Diese Ergebnisse zeigen, dass die Transformationsabläufe durch die Erstellung des Transformationsbaumes zur Erfassung der Tracing-Informationen deutlich verlangsamt werden. Da der zusätzliche Zeitaufwand hauptsächlich außerhalb der Transformationen verursacht wird, besteht für den Nutzer die Möglichkeit, die Erzeugung des Transformationsbaumes optional zu gestalten. Somit würde die Performance bei Transformationsabfolgen, bei denen kein Bedarf an Tracing besteht, beschleunigt werden, da beispielsweise im Experiment die Bereiche B1, B2 und B3 wegfallen würden. Nur im Falle von explizitem Bedarf an Tracing, wie beim Debugging, würde dann ein Transformationsbaum erzeugt werden.

Fazit

Dieses abschließende Kapitel fasst die zuvor präsentierte Lösung zusammen und gibt einen Ausblick auf zukünftige Einsatzbereiche.

7.1. Zusammenfassung

In dieser Arbeit wurde ein Framework vorgestellt, das Tracing von Modelltransformationen in EMF ermöglicht. Umgesetzt wurde das System in Form eines Eclipse Plug-ins als Teil des KIELER Projektes.

Wie in der Problemstellung aus Abschnitt 1.4 festgestellt, bestand Bedarf an der Erfassung von Entstehungsbeziehungen in Transformationen. Daher wurde eine Schnittstelle definiert und implementiert, mit der es möglich ist, innerhalb einer Transformation die Beziehungen zwischen transformierten Modellelementen zu registrieren und ein Mapping zu erzeugen. Diese Mappings können dann in eine dafür entworfene Datenstruktur, dem Transformationsbaum, eingetragen werden. Dieser unterstützt das Tracing auch bei Ketten von Transformationen und bei verschiedenen Transformationspfaden, ausgehend von einem Quellmodell. Zudem wurde eine Möglichkeit zum Aufbereiten der Informationen entwickelt, das Zusammenführen von Mappings. Dabei werden einzelne Mappings in einem Transformationspfad transitiv zusammengeführt und so ein direktes Mapping zwischen einem Quellmodell und endgültigem Zielmodell erstellt. Auf diese Weise wird es möglich Informationen zu Modellelementen auf äquivalente Elemente anderer Modelle zu übertragen.

Weiterhin wurde für den Transformationsbaum und die Mappings eine Visualisierung mit KLighD erstellt. Diese bietet einerseits eine Übersicht über die durchgeführten Transformationen und ist andererseits in der Lage interaktiv Mappings zwischen beliebigen Modellen im Baum zu visualisieren, wobei die modellspezifischen Diagramme genutzt werden können.

Die Evaluation der Testergebnisse zeigte, dass das Framework zum Tracing der SCCharts Transformationen erfolgreich eingesetzt werden kann und die Visualisierung geeignet ist, die durchgeführten Modelltransformationen und erfassten Entstehungsbeziehungen, zu veranschaulichen.

7.2. Zukünftige Einsatzbereiche

Die Anwendungstests des Frameworks haben gezeigt, dass es flexibel in eine Anwendung zu integrieren ist und das Tracing der Modelltransformationen ermöglicht. Auch wenn die hier vorgestellte Implementierung auf EMF und Java beziehungsweise Xtend basiert, lässt sich das Konzept auch auf andere Technologien übertragen. So kann das Framework in verschiedenen Einsatzbereichen verwendet werden.

Im KIELER Projekt sind im Kontext der SCCharts die folgenden zwei Einsatzbereiche geplant. Beide setzen eine vollständige Einbettung der Mappinggenerierung in die SCCharts Modelltransformationen voraus.

7.2.1. Simulation von SCCharts

Als eine graphische Programmiersprache bieten die SCCharts selbstverständlich die Möglichkeit ausführbaren Code zu erzeugen. Dazu wird der SCChart zunächst zu einem Core SCChart transformiert, dann normalisiert und in einen SCG umgewandelt. Auf diesem wird ein statisches Schedule durchgeführt und dann in sequenzieller Form in die Zwischensprache S übersetzt. Daraus kann dann der ausführbare C oder Java Code erzeugt werden.

Ziel der Simulation ist es, die Laufzeitdaten während der Ausführung auf die SCCharts zu übertragen und diese zu visualisieren. Dazu gehören Variablenwerte und aktive Zustände. Das KIELER Projekt enthält das KIELER Execution Manager (KIEM) Plug-in von Christian Motika [Mot09]. Dieses ermöglicht die automatische schrittweise Ausführung und bietet Zugriff auf zuvor festgelegte Laufzeitdaten. Es ist bereits möglich, ein SCChart auf oberster Ebene mit Markierungen zu versehen, die es ermöglichen diesem die Laufzeitdaten zuzuordnen, jedoch ist dieses Vorgehen nur eine Einzellösung für den Quell-SCChart. Durch die Nutzung des Tracing Frameworks ist es ausreichend, wenn die Laufzeitdaten mittels einer KIEM Anbindung auf das S Modell übertragen werden. Von dort können die Daten und Zustandsinformationen mittels eines Mappings auf den Quell SCChart übertragen werden. Weiterhin bietet es die Möglichkeit, die Laufzeitdaten auch auf andere Zwischenmodelle wie den SCG zu übertragen und so die Simulation auf verschiedenen Ausprägungsformen des Modells zu betrachten.

7.2.2. Interaktive Timing-Analyse

In einer aktuell laufenden Arbeit befassen sich Fuhrmann et al. [Fuh+14] mit einer Interaktive Timing-Analyse im Anwendungsfall der SCCharts. Dabei geht es um das Anzeigen der Worst-Case Execution Time (WCET) in SCChart, um dem Programmierer damit die Möglichkeit zu geben, sein Programm hinsichtlich

7.2. Zukünftige Einsatzbereiche

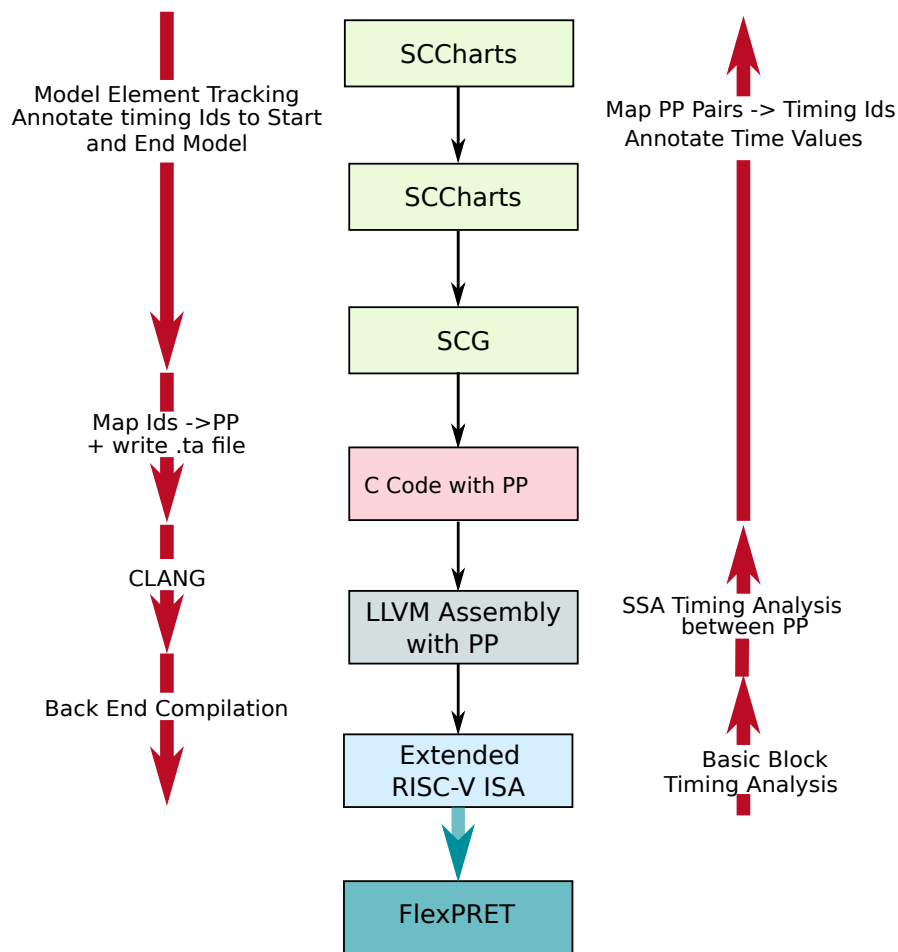


Abbildung 7.1. Phasen der Kompilierung und Fluss der Timing-Informationen [Fuh+14]

der Ausführungszeit zu optimieren. Da eine WCET Analyse nur auf Code-Ebene möglich ist, bedarf es eines entsprechenden Mappings, um die Ausführungszeiten korrekt im SCChart anzeigen zu können. Abbildung 7.1 zeigt die Phasen der Kompilierung hin zu messbaren Code, sowie den Ablauf von der Erstellung der Messpunkte hin zum Rückfluss der Timing-Informationen. Das entwickelte Tracing-Framework soll dabei für das Mapping von SCCharts bis SCG eingesetzt werden. Dabei geht es darum, die Abschnitte im Code zu bestimmen, die zu den einzelnen Regionen in SCCharts gehören, um passende Messpunkte setzen zu können. Für das eigentliche Übertragen der Timing-Informationen wird kein Tracing mehr benötigt, da diese dann bereits auf die Regionen gemapped sind.

Literatur

- [Ama+08] Bastien Amar, Hervé Leblanc, Bernard Coulette und Université Paul Sabatier. A Traceability Engine Dedicated to Model Transformation for Software Engineering. ECMDA Traceability Workshop. 2008.
- [Dei+99] Bernhard Deifel, Ursula Hinkel, Barbara Paech, Peter Scholz und Veronika Thurner. „Die Praxis der Softwareentwicklung: Eine Erhebung“. German. In: *Informatik-Spektrum* 22.1 (1999), S. 24–36. ISSN: 0170-6012.
- [FH10] Hauke Fuhrmann und Reinhard von Hanxleden. Taming Graphical Modeling. Technical Report 1003. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Mai 2010.
- [FHN06] Jean-Rémy Falleri, Marianne Huchard und Clémentine Nebut. „Towards a traceability framework for model transformations in Kermet“. In: *ECMDA-TW Workshop*. 2006.
- [Fuh11] Hauke Fuhrmann. „On the Pragmatics of Graphical Modeling“. Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.
- [Fuh+14] Insa Fuhrmann, David Broman, Steven Smyth und Reinhard von Hanxleden. Towards Interactive Timing Analysis for Designing Reactive Systems. *Workshop on Reconciling Performance with Predictability (RePP'14)*, ETAPS 2014. Grenoble, France, Apr. 2014.
- [GG07] Ismenia Galvao und Arda Goknil. „Survey of traceability approaches in model-driven engineering“. In: *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*. EDOC '07. IEEE Computer Society, 2007, S. 313–. ISBN: 0-7695-2891-0.
- [Han+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer und Owen O'Brien. „SCCharts: Sequentially Constructive Statecharts for safety-critical applications“. In: *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*. Edinburgh, UK: ACM, Juni 2014.
- [Har87] David Harel. „Statecharts: A visual formalism for complex systems“. In: *Science of Computer Programming* 8.3 (Juni 1987), S. 231–274.

Literatur

- [Jou+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev und Patrick Valduriez. „Atl: a qvt-like transformation language“. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. ACM. 2006, S. 719–720.
- [Lop+06] Denivaldo Lopes, Slimane Hammoudi, Jean Bézivin und Frédéric Jouault. „Mapping Specification in MDA: From Theory to Practice“. In: *In: Konstantas, D., Bourrières, J.-P., Léonard, M., Boudjlida, N. (Eds). Interoperability of Enterprise Software and Applications - INTEROP-ESA*. Springer, 2006, S. 253–264.
- [Mat10] Michael Matzen. „A Generic Framework for Structure-Based Editing of Graphical Models in Eclipse“. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mim-dt.pdf>. Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, März 2010.
- [Moh+10] Naouel Moha, Sagar Sen, Cyril Faucher, Olivier Barais und Jean-Marc Jézéquel. „Evaluation of Kermeta for solving graph-based problems“. In: *International Journal on Software Tools for Technology Transfer* 12.3-4 (2010), S. 273–285.
- [Mot09] Christian Motika. „Semantics and Execution of Domain Specific Models—KlePto and an Execution Framework“. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf>. Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dez. 2009.
- [Mot+13] Christian Motika, Steven Smyth, Reinhard von Hanxleden und Michael Mendler. *Sequentially Constructive Charts (SCCharts)*. Poster presented at 10th Biennial Ptolemy Miniconference (PTCONF'13), Berkeley, CA, USA. Juli 2013.
- [Nas13] Stanislaw Nasin. „Graphische Zuordnung von Elementen einer Modelltransformation“. Bachelor Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2013.
- [Sch06] Douglas C. Schmidt. „Guest editor’s introduction: model-driven engineering“. In: *Computer* 39.2 (Feb. 2006), S. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2006.58.
- [Sch11] Christian Schneider. „On Integrating Graphical and Textual Modeling“. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/chsch-dt.pdf>. Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Feb. 2011.

- [SSH13] Christian Schneider, Miro Spönemann und Reinhard von Hanxleden. „Just model! – Putting automatic synthesis of node-link-diagrams into practice“. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*. With accompanying poster. San Jose, CA, USA, 15–19 09 2013.
- [YW09] Andrés Yie und Dennis Wagelaar. „Advanced Traceability for ATL“. In: *1st International Workshop on Model Transformation with ATL (MtATL 2009)*, Nantes, France. Springer, 2009, S. 78–87.