Inkrementelles Update von Knoten-Kanten-Diagrammen mit EMF Compare 2.1

Carsten Sprung

Bachelorarbeit
eingereicht im Jahr 2014

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe für Echtzeitsysteme und Eingebettete Systeme
Prof. Dr. Reinhard von Hanxleden

Betreut durch: Dipl.-Inf. Christian Schneider

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

Der modellbasierte Entwurf spielt in der Systementwicklung eine immer größere Rolle. Das Eclipse Modeling Framework (EMF) bietet dazu komfortable Möglichkeiten zur Erstellung und Verarbeitung von Modellen und Modellinstanzen. Mit EMF Compare steht außerdem ein Framework zur Verfügung, mit dem sich EMF Modelle vergleichen und zusammenführen lassen. In der aktuell erschienenen Version stellt EMF Compare 2.1 eine komplette Neuimplementierung der vorherigen Version 1.3 dar. Änderungen im API erfordern grundlegende Anpassungen in der Verwendung von EMF Compare.

Diese Bachelorarbeit soll einen Einblick in die notwendigen Änderungen geben, die sich bei Verwendung von EMF Compare mit spezialisierten *Ecore*-Modellen durch den Versionssprung ergeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

API	Appliaction Programing Interface
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
IDE	Integrated Development Environment
KGraph	KIELER Graph
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KLay	KIELER Layouters
KLighD	KIELER Lightweight Diagrams
KSbasE	KIELER Structure-based Editing
MDSE	Model-Driven Software Engineering
MOF	Meta-Object Facility
OMG	Object Management Group
RCA	Rich Client Appilcation
RCP	Rich Client Platform
SCChart	Sequentially Constructive Chart
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Kapitel 1

Einleitung

Modellbasierte Softwareentwicklung ist aus der Informatik nicht mehr wegzudenken. Die Abstraktion von der realen Problemstellung, die mit der Modellierung einhergeht, spiegelt die intuitive Problemlösung des Menschen wieder. Sobald Menschen etwas untersuchen oder erklären, bilden sie ein Modell davon im Kopf.

Stellen Sie sich vor, Sie müssten jemandem ein Auto beschreiben. Vier Räder, ein Motor, ein Lenkrad, vielleicht noch Türen oder Scheinwerfer. Eine solche reduzierte Form des Autos stellt bereits ein Modell desselben dar. Es beinhaltet nur die wichtigsten Aspekte und lässt damit die Komplexität der Realität hinter sich.

Doch wenn Sie das Modell mit anderen diskutieren oder bearbeiten wollen, kommt es fast unweigerlich zu Kommunikationsproblemen. Der eine fasst möglicherweise LKW und PKW zusammen, ein anderer möchte hier gerne näher unterscheiden. Es bedarf also einer gemeinsamen Grundlage, auf der die Syntax der Modelle definiert wird.

1.1. Modellgetriebene Software-Entwicklung

Modelle abstrahieren von der realen Problemstellung. Graphisch aufbereitet bieten sie eine für den Menschen einfach lesbare und verständliche Form der Darstellung.

In der modellgetriebenen Softwareentwicklung (*Model-Driven Software Engineering* (MDSE)) werden Modelle genau aus diesen Gründen zur Darstellung komplexer oder komplizierter Zusammenhänge verwendet. Einfach zu definierende Semantiken ermöglichen es auch informatikfernen Fachkräften, domänenspezifische Zusammenhänge eindeutig zu formulieren.

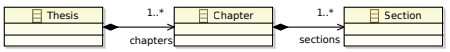
M3	Meta-Metamodell	MOF, Kermeta, KM3, Ecore
M2	Metamodell	UML, Petri-Netze, Xtext, DSLs
M1	Modell	
M0	Modellinstanzen	

Abbildung 1.1. Die Meta-Object Facility Architektur (Schneider [17])

1.1.1. Metamodellierung

Um Modellierungssprachen einheitlich zu definieren, hat die Object Management Group (OMG) den Meta-Object Facility (MOF) Standard[1] etabliert. Eine Kurzform zeigt Abbildung 1.1.

Die MOF Architektur unterteilt sich in vier Ebenen. Ebene `M0` enthält die Objekte der realen Welt. Dies sind die Objekte, die untersucht oder simuliert werden sollen, von denen also das Modell gebildet wird. Elemente der Ebene `M1` stellen das Modell dar. Diese beschreiben die Objekte in Ebene `M0`. Dabei können nicht relevante Eigenschaften ausgelassen werden, um die hohe Komplexität der Realität bewältigen zu können.

Ebene `M2` beschreibt wiederum die Elemente der Modell-Ebene. Sie wird auch als *Metamodell*-Ebene bezeichnet. Sie legt fest, welcher Syntax das Modell gehorchen muss und damit, welche Modelle gültig sind. In dieser Ebene sortieren sich bekannte Modellierungssprachen wie die Unified Modeling Language (UML) oder Petri-Netze ein. Die letzte Ebene `M3` stellt die Grundlage der Metamodelle dar, bezeichnet als *Meta-Meta*-Modellierungsebene. Um die Kette der definierenden Modelle abzuschließen, werden Meta-Metamodelle durch Modelle der Ebene `M3` selbst beschrieben. In diese Kategorie fallen das MOF selbst, aber auch das Meta-Metamodell des EMF (*Ecore*), das in Abschnitt 3.2 näher beleuchtet wird.

In Übereinstimmung mit der UML[2] sei im Folgenden das Modell der semantische Inhalt, der zum Beispiel als Textform in Extensible Markup Language (XML) oder einer Datenbank festgehalten werden kann. Das Diagramm zu einem Modell ist dann die graphische Repräsentation, bestehend aus Rechtecken und Verbindungslinien, die durch das Layout mit Positionsdaten versehen werden und dann dargestellt werden können.

[1] `http://www.omg.org/spec/MOF`

[2] `http://www.omg.org/spec/UML/2.5/Beta2/`

1.1.2. Diagramme und Layout

Zur Erstellung und Bearbeitung von Modellen steht mittlerweile eine große Palette an Programmen zur Verfügung.[3] Doch vielen ist das manuelle Layout der Diagramme gemein.

Bis mittels manuellem Layout ein ansehnliches Diagramm entsteht, braucht es Zeit und Nerven. Darüber hinaus kann leicht der Sinn für das Wesentliche verloren gehen. Schon bei relativ kleinen Diagrammen verwendet man mehr Zeit auf das Herumschieben von Knoten und Kanten als damit, das Diagramm mit Inhalt zu füllen (Fuhrmann et al. [6]). Wie das bei Modellen jenseits der 100 Knoten aussieht, bleibt der Vorstellungskraft des Lesers überlassen.

Sollen dann in einem fertig gelayouteten Diagramm Änderungen vorgenommen werden, wie zum Beispiel Knoten eingefügt werden, so muss erst mühsam Platz für die neuen Inhalte geschaffen werden. Die Elemente eines Diagramms zufriedenstellend anzuordnen nimmt also oft mehr Zeit in Anspruch, als das Diagramm selbst beziehungsweise dessen Inhalt zu erstellen. Abhilfe schaffen hier automatische Layout-Verfahren.

1.2. Automatisches Layout

Nach dem Prinzip der MDSE ist das Modell zwar zentrales Artefakt, soll aber trotzdem nur ein Mittel sein, Informationen darzustellen. Automatische Layout-Verfahren bieten die Möglichkeit, Knoten und Kanten ohne Hilfe des Nutzers anzuordnen. Eine Vielzahl an Algorithmen bieten für verschiedene Anforderungen entsprechende Lösungen ([2], [8], [9]). Auf diese Weise kann aus einem semantischen Modell ohne großen Aufwand ein Diagramm generiert werden, das nicht erst zeitintensiv vom Nutzer angeordnet werden muss. Der Nutzer erstellt folglich nur den Inhalt – das semantische Modell – selbst. Eine erhöhte Konzentration auf das Wesentliche des Modells ist die Folge.

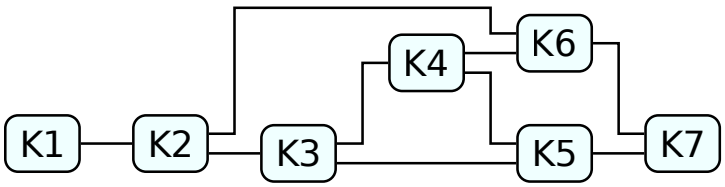
Das Graphical Modeling Framework (GMF),[4] das als Eclipse-Plugin zur Verfügung steht, stellt dazu eine Verbindung zwischen dem Graphical Editing Framework (GEF)[5] und EMF her. GMF ermöglicht es, graphische Editoren zu gegebenen Metamodellen zu generieren. Zu einem vorhandenen semantischen Modell kann so auch ein Diagramm initialisiert werden. Es werden jedoch immer zwei Modelle verwaltet, eines, das das semantische Modell darstellt und eines, das das Diagramm mit Positionen usw. enthält.

Das KLighD Framework synthetisiert zur Laufzeit automatisch gelayoutete Diagramme. Das Diagramm selbst wird dabei nicht persistiert. Änderungen können beispielsweise per Texteditor an beliebiger Stelle eingefügt werden und

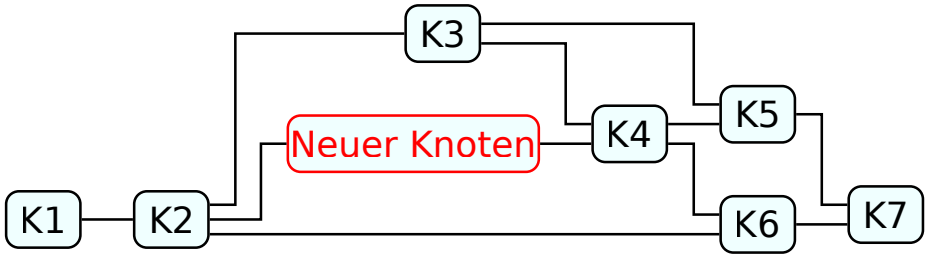
[3] http://www.umltools.net/

[4] http://www.eclipse.org/modeling/gmp/

[5] http://www.eclipse.org/gef/



(a) Vorhandenes Layout



(b) Layout nach dem Einfügen eines neuen Knotens

Abbildung 1.2. Beispiel für unterschiedliche Layoutergebnisse bei Modelländerungen (Erstellt mit KLighD, nach Schipper [16])

werden on-the-fly im Diagramm dargestellt. Eine detailliertere Beschreibung des KLighD-Frameworks folgt in Abschnitt 3.5.

1.3. Mental Map

Während der Arbeit mit Modellen hat ein Nutzer bereits eine Vorstellung des Modells im Kopf, eine so genannte *mental map*. Diese Vorstellung enthält unter anderem Erwartungen, wie sich Änderungen im Modell auf das Diagramm auswirken. Wie Archambault und Purchase [1] beschreiben, ist es wichtig diese mental map zu erhalten, um einen ungestörten Arbeitsablauf mit Modellen zu gewährleisten.

Das automatische Layout der Diagramme während der Bearbeitung wirft also ein neues Problem auf: Beim Hinzufügen oder Entfernen von Knoten oder Kanten ergibt das automatische Layout möglicherweise eine ganz andere Anordnung der Diagrammelemente als vor der Änderung, wie Abbildung 1.2 zeigt. Hier wird deutlich, wie schwer es fällt, die bekannte Position der Knoten im Geiste zu aktualisieren – wobei hier beide Diagramme parallel zu sehen sind. Bei einer Bearbeitung im Editor wäre das Diagramm in Abbildung 1.2(a) dagegen schon durch das aktualisierte Diagramm in Abbildung 1.2(b) ersetzt worden. Dies stört die Übersichtlichkeit erheblich und macht die Änderung schwerer nachvollziehbar. Dazu kommen Probleme bei der Arbeit mit hierarchischen Modellen. Enthalten Knoten innere Graphen, so soll der Expansionsstatus erhalten bleiben. Doch

während der Synthese ist noch gar nicht bekannt, ob der Nutzer einen Knoten aufgeklappt hat oder nicht.

1.4. Problemstellung

Es bedarf also einer Möglichkeit, das neu generierte Diagramm mit dem schon vorhandenen zusammenzuführen. In der Anzeige hat dies den Vorteil, dass nicht das komplette Diagramm neu angezeigt werden muss, sondern dass nur Änderungen visualisiert werden. So kann der Nutzer einfach nachvollziehen, was passiert und wie sich seine Anpassungen auswirken.

Eclipse bietet mit EMF Compare[6] eine einfache Möglichkeit, EMF-Modelle miteinander zu vergleichen, Unterschiede zu berechnen und in ein neues Modell zu vereinen. Für die bisher verfügbare EMF Compare Version 1.3 existiert eine Anbindung zur Verwendung mit KLighD. Diese *Updatestrategie* kümmert sich um die Vorbereitung der zu vergleichenden Modelle sowie Initialisierung und Aufruf des Vergleichs.

Mit dem Release der Eclipse Version *Kepler* wurde auch EMF Compare von Version 1.3 auf 2.1 aktualisiert. Dabei wurden grundlegende Änderungen am Appliaction Programing Interface (API) des EMF Compare vorgenommen.[7]

Die bisherige Implementierung der Anbindung von EMF Compare an KLighD lässt sich deshalb nicht weiter verwenden, eine Neuimplementierung beziehungsweise Anpassung ist erforderlich.

Diese Neuimplementierung muss sich wie zuvor auch um die Vorbereitung und Initialisierung des Vergleichs kümmern. Dazu gehört, festzulegen beziehungsweise anzupassen, wie Modellelemente einander zugeordnet werden und wie Unterschiede berechnet und zusammengeführt werden.

Abgesehen davon sind Inkompatibilitäten mit dem KLighD Framework aufgrund geänderter Verhaltensweisen von EMF Compare zu erwarten. In einem solchen Fall muss die Ursache gefunden werden und behoben werden. Es ist dabei durchaus möglich, das Probleme durch Fehler im EMF Compare verursacht werden, da das Framework sich noch im aktiven Entwicklungsstatus befindet, wie die gut gefüllte Bugliste[8] zeigt.

1.5. Aufbau der Arbeit

Das folgende Kapitel 2 stellt andere Arbeiten aus dem Bereich der Modellverarbeitung und -zusammenführung vor. Es soll einen Überblick darüber geben, was

[6] http://www.eclipse.org/emf/compare/

[7] https://projects.eclipse.org/projects/modeling.emfcompare/releases/2.1.0/plan

[8] https://bugs.eclipse.org/bugs/buglist.cgi?product=EMFCompare

beim Vergleichen und Verarbeiten von Modellen zu beachten ist, und was heutzutage schon möglich ist. Kapitel 3 beschreibt Programme und Frameworks, die für die Entwicklung und von der Implementierung genutzt werden. Der Entwurf, vorgestellt in Kapitel 4, erklärt das geplante Vorgehen und welche Anforderungen die Implementierung erfüllen soll. Diese wird in Kapitel 5 im Detail erläutert. Zum Abschluss fasst Kapitel 6 die Arbeit zusammen und gibt einen Ausblick über zukünftige Arbeiten.

Kapitel 2

Verwandte Arbeiten

Dieses Kapitel beschäftigt sich mit wissenschaftlichen Arbeiten zu ähnlichen oder verwandten Themen. Eine Vielzahl an Veröffentlichungen beschäftigen sich mit MDSE im Allgemeinen und mit der Verarbeitung von Modellen im Speziellen. Zum konkreten Thema der Anwendung von EMF Compare 2 mit eigenen Metamodellen lassen sich keine Arbeiten finden. Im folgenden Abschnitt sind daher hauptsächlich Arbeiten vorgestellt, die sich auch etwas entfernter mit dem Zusammenführen von Modellen beschäftigen oder die vorherige EMF Compare Version 1 betreffen.

Bendix und Emanuelsson [3] berichten aus dem Alltag der Modellierung im Team. Selbst unter besten Umständen, wie sie in einem gut koordinierten Team herrschen, das einheitliche Werkzeuge und Stilkonventionen verwendet, ist es kein triviales Problem, nebenläufige Änderungen an Modellen in den Griff zu bekommen. Richtlinien und terminliche Vereinbarungen zur Vermeidung paralleler Arbeit an Modellen sind da nur eine Notlösung, die gleichwohl funktioniert. Um so wichtiger ist es, Modelle in einer Art und Weise bearbeiten zu können, die eine einfache Möglichkeit der Teamarbeit unterstützt und dennoch die Vorteile der Diagrammdarstellung nutzt.

Brun und Pierantonio [4] beschäftigen sich mit dem Problem des Modellvergleichs an sich und welche Probleme dabei zu lösen sind. Außerdem geben sie einen Einblick, wie EMF Compare die auftretenden Schwierigkeiten zu lösen versucht. Brun und Pierantonio unterteilen das Hauptproblem in drei Abschnitte:

- **Berechnung der Unterschiede:** Es existieren bereits diverse Algorithmen zu diesem Thema, die sich in zwei Kategorien einteilen lassen. ID-basierte Algorithmen können sehr einfach feststellen, ob zwei Elemente korrespondieren, sind jedoch an die Entwicklungsumgebung gebunden, von der die IDs berechnet werden. Algorithmen, die eine Ähnlichkeitsmetrik verwenden, haben dieses Problem nicht, sind jedoch ungleich schwieriger zu implementieren und müssen eventuell an das verwendete Metamodell der zu vergleichenden Modelle angepasst werden.

- **Repräsentation der Unterschiede:** Die Wahl der Repräsentation legt fest, welche Möglichkeiten zur Weiterverarbeitung der Unterschiede zur Verfügung stehen. Die Repräsentation sollte also alle verfügbaren Informationen zu den Unterschieden enthalten, aber dennoch eine ausreichende Abstraktion wahren, um verschiedenartige Verarbeitungsmöglichkeiten offen zu halten. Brun und Pierantonio stellen dazu einige Bedingung vor, die eine Repräsentation erfüllen sollte.

- **Visualisierung der Unterschiede:** Die Visualisierung stellt genau die Aspekte der Repräsentation dar, die für die aktuelle Aufgabenstellung entscheidend sind. Dabei kann die Darstellung auf verschiedenste Art und Weise erfolgen, beispielsweise intuitiv als Diagramm oder zur maschinellen Weiterverarbeitung als Tabelle.

EMF Compare verwendet einen Metrik-basierten Ansatz, der Ecore-Modelle ohne zusätzliche Anpassungen verarbeiten kann, aber trotzdem einfach erweiterbar ist. Unterschiede werden in Modellform repräsentiert und sind damit unabhängig vom verwendeten Metamodell der Nutzdaten.

MetaDiff ist ein von Kofman und Perjons [10] vorgestelltes Framework, um Modelle in Relation zu bringen, Unterschiede zu berechnen und Modelle zusammenzuführen. MetaDiff basiert auf einer Menge von *Extension-Templates* für *Match-*, *Diff-* und *Merge-*Algorithmen. Durch diesen Aufbau ist das Framework einfach an eigene Bedürfnisse anzupassen. An spezielle Metamodelle angepasste Algorithmen können beispielsweise durch die Extension-Templates einfach vom Framework genutzt werden.

Für die Anwendung mit KLighD eignet sich EMF besser als MetaDiff, da EMF Compare schon generische Algorithmen für EMF-Modelle mitbringt. MetaDiff eignet sich eher für Anwendungsfälle, in denen ein eigenes Metamodell verwendet wird und dazu Algorithmen schon vorhanden sind oder entscheidende Vorteile gegenüber generischeren Algorithmen bieten.

Es gibt darüber hinaus diverse weitere Ansätze und Lösungsvorschläge, wie Modelle am einfachsten und effizientesten in Relation zu bringen sind. Kolovos et al. [11] stellen einige davon vor und vergleichen sie in Hinblick auf Genauigkeit, Effizienz und Unabhängigkeit von Modellierungssprache und -Tool. Dabei stehen bestimmte Eigenschaften in Kontrast zueinander. Ein Algorithmus, der exakt auf eine Modellierungssprache abgestimmt ist, erreicht mit großer Wahrscheinlichkeit eine bessere Genauigkeit und Performance, als ein generischer Algorithmus, erfordert im Gegenzug aber auch einen höheren Implementierungs- und Konfigurationsaufwand. Es gibt also keine allgemeingültige, beste Lösung. Je nach Anforderung und Möglichkeiten eignen sich einige Ansätze besser als andere.

Es ist nicht nur entscheidend, nach welchem Verfahren zusammengehörende Modellelemente gefunden werden. Auch die Art und Weise, in der gefundene Elementpaare zusammengeführt werden, wirkt sich auf das entstehende Modell aus. Dazu stellen Pottinger und Bernstein [15] einen Metamodell-unabhängigen *Merge*-Operator vor, der aus zwei gegebenen Modellen und einem *Mapping* ein zusammengeführtes Modell erzeugt. Schwierigkeiten entstehen, wenn beispielsweise das Mapping nicht eindeutig festlegt, welche Werte in das gemeinsame Modell Übernommen werden sollen. Weiter müssen bestimmte Bedingungen eingehalten werden, zum Beispiel dürfen durch die Zusammenführung keine zusätzlichen oder redundanten Informationen entstehen und das neue Modell muss weiterhin im gegebenen Meta-Metamodell gültig sein. Ein weiteres Augenmerk liegt auf der Auflösung von Konflikten. Pottinger und Bernstein unterteilen diese in drei verschiedene Klassen:

- **Repräsentationskonflikte:** Dies sind Konflikte auf Modellebene. Sie entstehen beispielsweise durch verschiedene Repräsentation realer Objekte oder ganz einfach durch nebenläufige Bearbeitung desselben Modells. Zum Beispiel löscht ein Entwickler ein Element, ein anderer benennt es um. Hier ist die Interaktion des Nutzers erforderlich, um den Konflikt aufzulösen, wie man es beim Mergen versionierter Textdateien schon kennt.
- **Metamodellkonflikte:** Aufgrund der Metamodellunabhängigkeit des vorgestellten *Merge*-Operators ist es möglich, zwei Modelle zu mergen, die unterschiedlichen Metamodellen gehorchen. Zum Beispiel könnte man sich vorstellen, einen KIELER Graph (KGraph) mit einem Sequentially Constructive Chart (SCChart) zu kombinieren. Beide Modelle bestehen aus Knoten und Kanten, doch keines der Modelle ist im Metamodell des jeweils anderen gültig. Lösungen müssten an die verwendeten Metamodelle angepasst werden.
- **Fundamentale Konflikte:** Konflikte dieser Art entstehen durch Restriktionen im Meta-Metamodell. Sie treten auf, wenn zum Beispiel zwei Elemente verschiedenen Typs zusammengeführt werden sollen. Konflikte diesen Typs können im in dieser Bachelorarbeit betrachteten Fall jedoch nicht auftreten und werden daher hier nicht näher erläutert.

In seiner Diplomarbeit geht Schipper [16] auf die Erkennung und Darstellung von Unterschieden speziell in Statecharts (Harel [7]) ein. Für die sinnvolle Visualisierung ist es wichtig, dass semantische Unterschiede von Unterschieden in den Diagrammdaten (Position oder Ähnliches) getrennt und in geeigneter Form Dargestellt werden. Gibt es zu viele Unterschiede, so ist durch farbliche Kennzeichnung nichts zu gewinnen, da dann zu viel markiert würde und die Übersicht verloren ginge. Schipper stellt daher verschiedene Visualisierungsmöglichkeiten vor, wie zum Beispiel die Ansicht beider Modelle nebeneinander oder

die Überführung eines Modells in das andere in einer Animation. Weiter geht er auf die Erhaltung des Layouts ein, um einen ungestörten Arbeitsablauf des Entwicklers zu ermöglichen (siehe auch mental map, Abschnitt 1.3).

Könemann stellt dar ([12], [13]), dass aktuelle Algorithmen und Programme nicht mit in Betracht ziehen, was ein Unterschied zwischen zwei Modellen eigentlich meint. Außerdem stellt er das Prinzip der modellunabhängigen Unterschiede vor und wie man daraus Patches für andere Modelle gewinnen kann. Angenommen, die Suche nach Unterschieden ergibt, dass eine Menge von Unterelementen gelöscht wurde und unter einem anderen Knoten wieder eingefügt wurde. Gemeint ist aber vom Entwickler, dass alle Unterelemente (auch solche, die in seinem Modell gar nicht vorhanden sind, aber vielleicht in dem anderer Entwickler) an einen neuen Knoten verschoben werden sollen. Um diese Beschreibung als Patch nutzen zu können, müssen alle Informationen enthalten sein, um den Patch auf ein beliebiges Modell anwenden zu können. Das beinhaltet nicht nur, wo sich etwas geändert hat, sondern auch den Wert vorher und nachher, wie die Änderung zu finden ist und von welchem Typ sie ist (Element/Attribut/Referenz, geändert/hinzugefügt/entfernt). Unterschiede dürfen geänderte Elemente nicht direkt referenzieren. Stattdessen werden sie über symbolische Referenzen angesprochen. Diese werden zur Laufzeit anhand der enthaltenen Beschreibung aufgelöst. Im Gegenzug geht durch die Generalisierung der Unterschiede ein gewisser Teil Genauigkeit verloren, was unter Umständen Benutzereingriffe währende der Anwendung des Patches erfordert.

Die bisher vorgestellten Arbeiten haben nur den algorithmischen Teil betrachtet, wie Modelle maschinell zusammengeführt werden können. Lutz et al. [14] stellen eine Studie zu dem Thema vor, wie Menschen Modelle mergen. Den Probanden wurden verschiedene UML Diagramme vorgelegt, die ohne bestimmte Vorgaben mit eigenen Strategien zusammengeführt werden sollten. Das Experiment ergab einige Designrichtlinien für Anwendungen, die das interaktive Mergen von Modellen unterstützen sollen. Neben relativ offensichtlichen Hilfestellungen, wie Konflikthervorhebungen, Änderungshistorie oder Unterstützung durch automatisches Matching sollte beispielsweise auch ein individueller Arbeitsfluss ermöglicht werden, da verschiedene Menschen auch verschiedene Herangehensweisen verwenden. Außerdem sollten Anmerkungen und Gruppierungen unterstützt werden, um komplexe Zusammenhänge zu vereinfachen oder erklären zu können.

Kapitel 3

Verwendete Technologien

Das folgende Kapitel stellt verwendete Programme und Frameworks vor. Einige wurden schon in Kapitel 1 erwähnt, werden hier jedoch der Vollständigkeit halber noch einmal aufgeführt und ausführlicher erläutert.

3.1. Das Eclipse Projekt

Das Eclipse Projekt wurde ursprünglich 2001 von IBM und führenden Firmen der Industrie als Plattform für Entwickler und Firmen im Bereich Open Source Software ins Leben gerufen. Ende 2003 war dieses Konsortium auf über 80 Mitglieder angewachsen. 2004 wurde daraus die Eclipse Foundation, eine unabhängige non-Profit Organisation, die eine anbieterneutrale, offene und transparente Gemeinschaft sicherstellen soll.[1]

3.1.1. Die Eclipse Anwendung

Kernbestandteil des Projekts ist die Open Source Anwendung Eclipse. Ursprünglich als erweiterbare Integrated Development Environment (IDE) für Java und andere objektorientierte Sprachen gedacht, implementiert Eclipse seit der Version 3.0 die OSGi Spezifikation.[2] Diese definiert ein plattformunabhängiges Komponentenmodell, das eine einfachere und eindeutigere Modularisierung von Software fördert. Eclipse selbst besteht damit nur noch aus einem kleinen Kern, genannt *Equinox*, welcher die *Bundles* (nach OSGi) bzw *Plugins* (Eclipse-Terminologie) nachlädt. Eclipse bietet nunmehr eine Basis für die Plugins, die sogenannte Eclipse Rich Client Platform (RCP). Die Menge der geladenen bzw. mitgelieferten Plugins kann an den jeweiligen Verwendungszweck angepasst werden. Dies bietet nicht nur Performancevorteile, sondern steigert auch die Übersichtlichkeit für den Nutzer, da im Idealfall nur die Elemente und Funktionen geladen, also sichtbar sind, die für den konkreten Problemfall benötigt werden. Eine solche angepasste Eclipse Version wird als Rich Client Appilcation (RCA) bezeichnet. Diverse Anwendungen

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

[2] http://www.osgi.org/

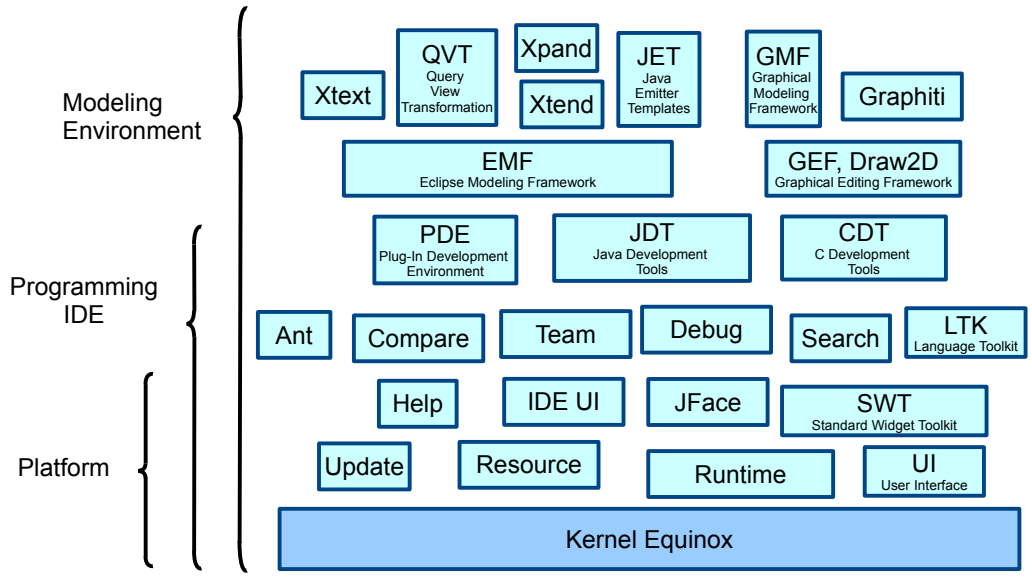


Abbildung 3.1. Übersicht der Eclipse Plattform (Fuhrmann [5])

für die unterschiedlichsten Einsatzgebiete wurden auf dieser Basis entwickelt, von Planungstools für Flugsimulatoren bis hin zu Enterprise Anwendungen zur Kontrolle und Überwachung nuklearwissenschaftlicher Experimente.[3] Ein weiteres Beispiel für eine solche RCA ist KIELER.

In der Eclipse-Gemeinschaft haben sich einige geläufige Plugin-Zusammenstellungen entwickelt, von denen Abbildung 3.1 ein Beispiel-Setup zeigt. Zusammen mit einigen grundlegenden Plugins stellt der Kernel die Eclipse Platform dar. Dazu kommen Plugins für konkrete Programmiersprachen wie Editoren, Compiler oder Debugger, die dann gemeinsam mit der Plattform die IDE bilden. Auch eine Reihe speziell für MDSE entwickelte Plugins stehen zur Verfügung. In Eclipse bilden sie das Modeling Environment.

Zur Interaktion und Kommunikation zwischen Plugins verwendet Eclipse sogenannte *Extension Points*, skizziert in Abbildung 3.2. Auf diese Weise ist es einfach möglich, eigene Plugins zu entwickeln, welche die Funktionalität anderer Plugins nutzen und eigene Funktionen zur Verfügung stellen, ohne dass der Code der kompletten IDE bekannt sein oder geändert werden muss. Jedes Eclipse-Plugin beschreibt dazu in textueller Form (`MANIFEST.MF` und `plugin.xml`), welche Extension Points es nutzt und welche es anderen Plugins zur Verfügung stellt. Damit wird auch festgelegt, welche Abhängigkeiten zwischen Plugins bestehen. So kann der Kernel genau die Plugins laden, die auch benötigt werden, ohne dass der Java-Code zur Laufzeit instanziiert und untersucht werden muss.

[3] `http://www.eclipse.org/community/example_rcp_applications_v2.pdf`

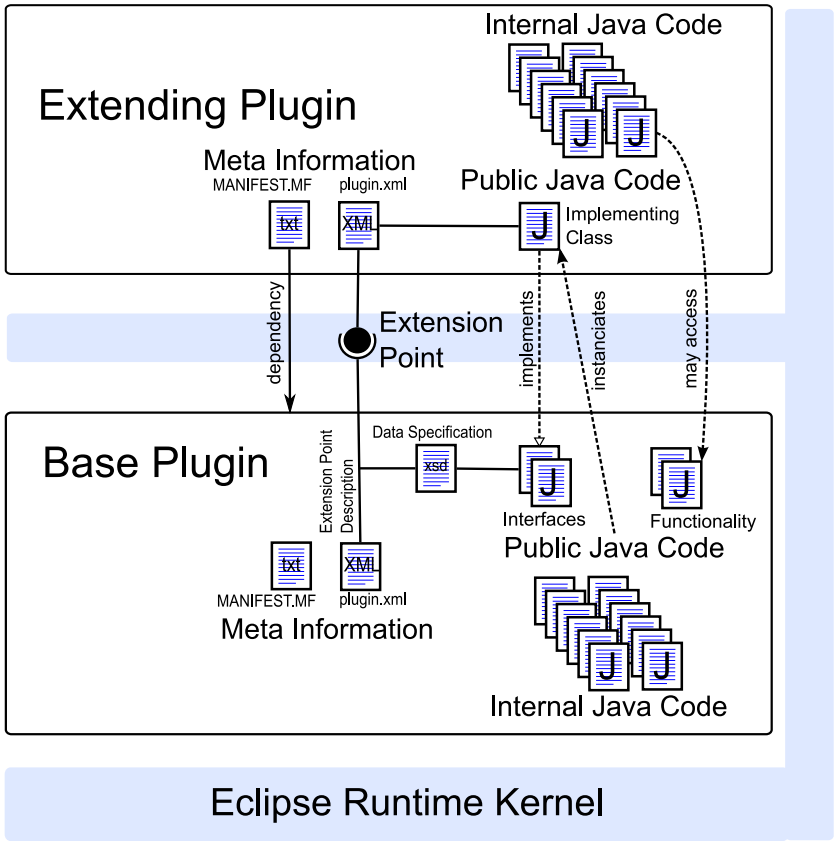


Abbildung 3.2. Beziehungen zwischen Eclipse Plugins (Fuhrmann [5])

3.2. Eclipse Modeling Framework (EMF)

Hauptaugenmerk des EMF ist die Konstruktion von abstrakter Syntax in Form von Modellen und Verarbeitung von Instanzen derselben. Mit EMF lassen sich Modelle mittels Baumstruktur-basierter Editoren oder via Programmcode erstellen. EMF stellt also die Brücke zwischen Modellierung und Programmierung her. Ziel von EMF ist beispielsweise die Spezifikation von Prozessen, um daraus passenden Programmcode zu generieren, die Austauschbarkeit von Informationen zu gewährleisten oder die Konfiguration von Software. Ein prominentes Beispiel dazu ist die Organisation der Eclipse *Workbench* (e4). Um das Modell der Workbench zu vereinheitlichen und GUI-Erweiterungen zu vereinfachen, wurden die Spezifikationen als EMF Modell redesignt.[4]

Zur Handhabung größerer Modelle unterstützt EMF die Modularisierung in *Fragmente*. So können Teilmodelle in separaten Dateien abgespeichert werden.

[4] http://www.slideshare.net/LarsVogel/eclipse-e4-tutorial-eclipsecon-2010-3587897

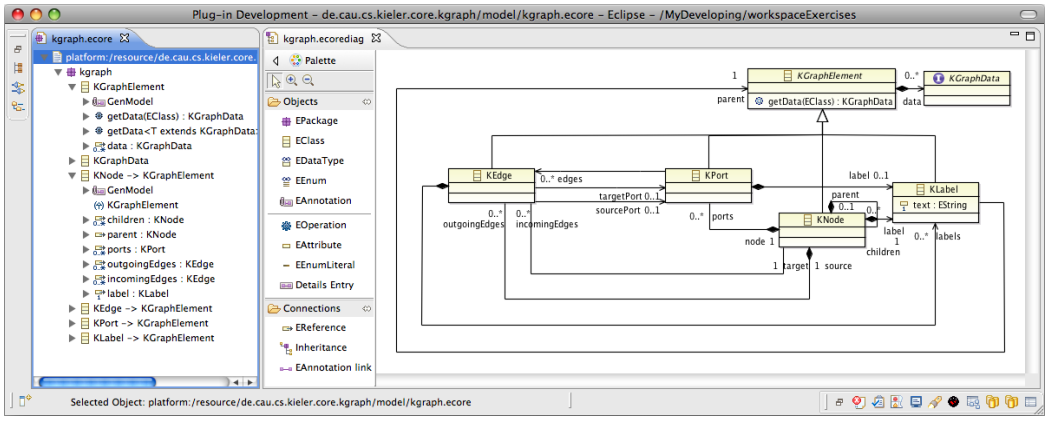


Abbildung 3.3. Baumbasierte und graphische Ecore-Editoren für EMF (Schneider [17])

Als Meta-Metamodell wird *Ecore* verwendet, das gemäß der MOF ein M3 Modell darstellt (Abbildung 1.1). Zur Bearbeitung der Ecore Modelle steht in EMF sowohl ein Baumstruktur-basierter als auch ein graphischer Editor bereit. Ein Beispiel dazu zeigt Abbildung 3.3.
EMF besteht aus 3 Kernkomponenten:

- **EMF Core:**

  Der EMF-Kern enthält Definitionen zum Metamodell (*Ecore*), Laufzeitunterstützung inklusive Persistierung in Form von XML Metadata Interchange (XMI) und Schnittstellen, um EMF-Objekte zu verarbeiten.

- **Editoren:**

  Das Editoren-Framework stellt Klassen zur Erstellung von Editoren zur Verfügung. Diese Editoren können dazu verwendet werden, graphische Darstellungen der Modelle anzuzeigen und zu bearbeiten. Ebenso ist es möglich, Modelle auf programmatischer Ebene zu bearbeiten.

- **Codegenerierung:**

  Mit EMF lässt sich zu einem Metamodell alles generieren, was für einen Modelleditor benötigt wird. Es stehen drei Generierungsstufen zur Auswahl:

  - **Modell:** Generiert nur Java Klassen und Interfaces für die Klassen des Modells.
  - **Adapter:** Generiert zusätzlich Implementierungen für Bearbeitung und Darstellung der Modellklassen.
  - **Editor:** Generiert den kompletten Editorcode, der dann an eigenen Bedürfnisse angepasst werden kann.

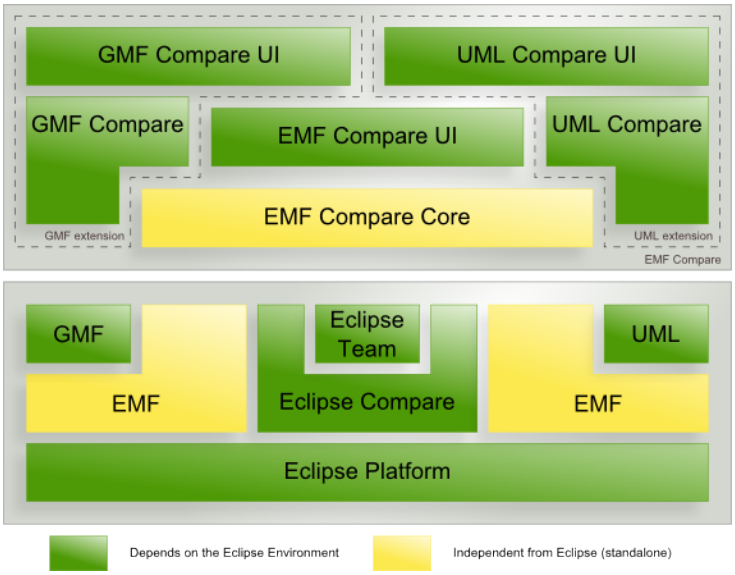


Abbildung 3.4. EMF Compare Architektur [4]

3.3. EMF Compare

EMF Compare ist ein Unterprojekt des EMF. Es ergänzt EMF um Modellvergleiche und das Zusammenführen von Modellen. Zur Verfügung stehen ein GUI-basiertes Vergleichstool, wie man es von Textvergleichen bereits kennt, und die Möglichkeit, aus dem eigenen Programmcode heraus die Funktionalität von EMF Compare zu nutzen. Dazu bietet es metamodellunabhängige Schnittstellen, die im Bedarfsfall spezialisiert werden können.

EMF Compare setzt auf der Eclipse Plattform auf. Abbildung 3.4 veranschaulicht die Projekt-Architektur. Es basiert auf EMF, Eclipse Compare (Zuständig für textbasierte Vergleiche) und Eclipse Team, das Schnittstellen zu bekannten Repositories (Git, CVS, SVN, ...) bereitstellt. Trotz der engen Verknüpfung mit den genannten Komponenten ist EMF Compare in der Lage, unabhängig in eigenständigen Anwendungen verwendet zu werden, ebenso wie EMF selbst (gelbe Markierung in der Abbildung).

EMF Compare wurde 2006 von Toulmé [19] vorgestellt. Ziel war es, ein Framework für den Modellvergleich zu entwickeln, dass die wirklichen, semantischen Unterschiede findet, und zwar nicht nur in der Historie desselben Modells, sondern auch zwischen komplett unabhängig entwickelten Modellen. EMF Compare beruht daher nicht auf dem Vergleich von IDs, da diese Editorabhängig sind und im Grunde nichts über Gleich- oder Ungleichheit zweier Objekte aussagen. Stattdessen wird eine Entfernung berechnet, die angibt, wie ähnlich sich zwei

[4] http://www.eclipse.org/emf/compare/doc/21/developer/developer-guide.html

Elemente sind. Die Attribute und Referenzen aller Modellelemente werden dazu mit einer Gewichtung versehen, die von zwei Faktoren bestimmt wird:

- Die Häufigkeit, mit der das Attribut auftritt. Je häufiger ein Attribut bei der rekursiven Suche im Modell angetroffen wird, desto unwichtiger ist es für die Unterscheidung zwischen Elementen.
- Die Menge der möglichen Werte, die ein Attribut annehmen kann. Hat ein Attribut beispielsweise nur zwei mögliche Werte, wie ein Boolean, so ist es nicht besonders gut zur Unterscheidung geeignet.

Ein empirisch festgelegter Wert gibt die maximale Entfernung an, ab der zwei Elemente als vollkommen verschieden interpretiert werden. EMF Compare unterstützt ebenfalls den Vergleich zwischen drei Modellen, wie er beispielsweise in der Versionierung häufig anzutreffen ist. Dieser wird hier jedoch nicht benötigt. Der Vergleichsprozess kann grob in drei Phasen unterteilt werden.

- **Match:** In der ersten Phase werden die zu vergleichenden Modelle parallel durchsucht und zusammenpassende Elemente gefunden. Die gefundenen Paare (Tripel beim Drei-Wege-Vergleich) bilden das *Matchmodel*, ein Mapping zwischen den Modellen. Die Elemente des Matchmodels, die *Matches*, enthalten *Submatches* entsprechend der Hierarchie der verglichenen Modelle. Das Matchmodell spiegelt damit die Struktur der Modelle wieder.

- **Diff:** Im zweiten Schritt werden Unterschiede berechnet. Die Unterschiede ergeben ebenfalls in der hierarchischen Struktur der verglichenen Modelle das sogenannte *Diffmodel*.

- **Merge:** In der letzten Phase werden die Modelle anhand des Diffmodels zu einem neuen Modell zusammengeführt. Dieser Schritt ist optional, da man unter Umständen nur an der Berechnung der Unterschiede zwischen Modellen interessiert ist.

Verschiedene Arten von Unterschieden können erkannt und verarbeitet werden:

- Unterschiede zwischen Elementen in Modellen:
  - *Creation*: Ein Element wurde hinzugefügt.
  - *Deletion*: Ein Element wurde gelöscht.
  - *Order*: Die Reihenfolge zwischen Elementen wurde geändert.

- Unterschiede zwischen Attributen eines Elements:
  - *Set*: Der Wert des Attributs hat sich geändert.
  - *Add*: Das Attribut wurde hinzugefügt.
  - *Remove*: Das Attribut wurde entfernt.

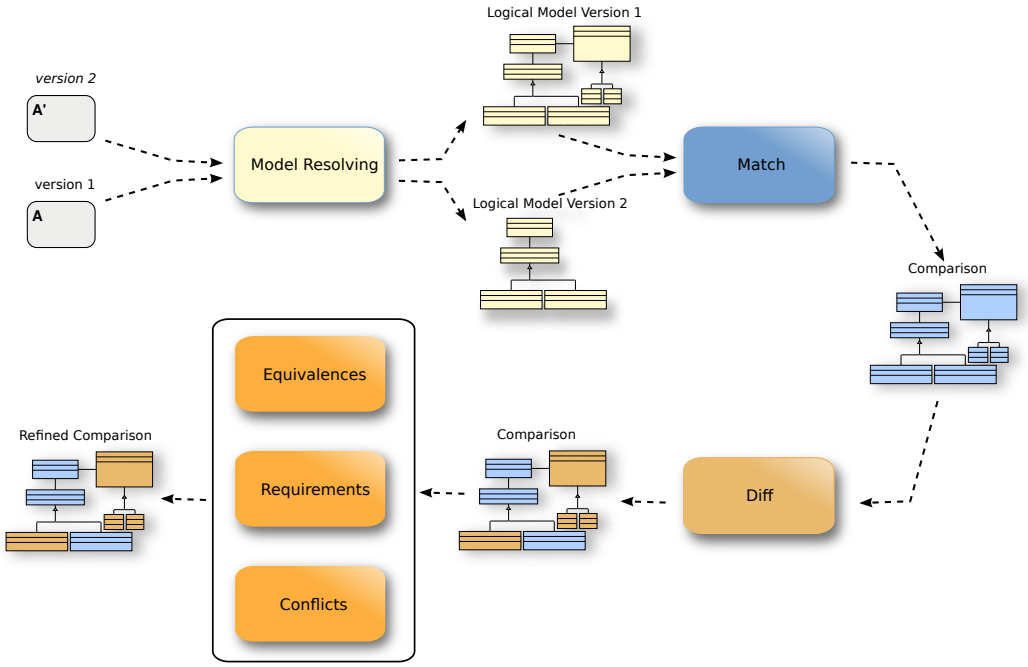


Abbildung 3.5. EMF Compare Prozessablauf [4]

Bei der Entwicklung der ersten Version stand die Perfomance noch nicht im Vordergrund. Elemente wurden öfters als notwendig miteinander verglichen, was zu Laufzeiten von mehreren Minuten bei relativ kleinen Modellen mit ca. 100 Knoten führen kann.

3.3.1. Version 2

EMF Compare 2 ist eine komplette Überarbeitung des Frameworks. Durch die Überarbeitung sowohl des Designs als auch der Architektur ist das API nicht kompatibel zur vorherigen Version 1. Es gibt nicht mehr, wie voher, ein explizites Match- und Diffmodel. Alles ist jetzt gekapselt in einer `Comparison`-Klasse, die den kompletten Vergleich darstellt. Sie stellt die Wurzel des Vergleichmodells dar, das aus den einzelnen Matches und deren Submatches aufgebaut wird. Jedes Match-Objekt enthält dann die Unterschiede der gemappten Modellelemente. Die Dreiteilung in Match/Diff/Merge ist immer noch vorhanden, jedoch spiegelt sie sich nicht mehr im API wieder. Abbildung 3.5 zeigt den Ablauf schematisch.

Zunächst werden die Modelle geladen (`Model Resolving`). Bei fragmentierten Modellen (Abschnitt 3.2) werden hier alle Dateien zusammengetragen, die Teile des Modells enthalten. Auch dieser Schritt musste in der vorherigen Version durchgeführt werden, ein entscheidender Unterschied ist jedoch, dass mit Version 2 nur noch die Fragmente des Modells geladen werden, die sich wirklich geändert

haben. Sind zwei Dateien, die Fragmente der Modelle enthalten, identisch, so kann sich auch nichts in den enthaltenen Modellteilen geändert haben.[5]

Im aktuellen Anwendungsfall entfällt dieser Schritt, da für das inkrementelle Update der Diagramme die Modelle schon in Form ihrer Java-Implementierung vorliegen.

Es folgen die Match- und Diff-Phasen. Diese sind im API nicht mehr getrennt sichtbar, sondern werden als *Compare* zusammengefasst. Intern wird zunächst das Matchmodel aufgestellt. Das Matchmodel wird dann um die Unterschiede erweitert.

Im Anschluss werden äquivalente Unterschiede (wie zum Beispiel bei bidirektionalen Referenzen), Implikationen (Container müssen vor ihrem Inhalt hinzugefügt werden) und Konflikte aufgelöst.

Wie zuvor schon erwähnt, bietet EMF Compare vielfältige Erweiterungs- und Anpassungsschnittstellen. Welche Elemente für den Vergleich berücksichtigt werden, kann über einen sogenannten *Scope* festgelegt werden, ein Filter, der nur die zu vergleichenden Elemente zurückliefert. Im späteren Verlauf des Zusammenführens zweier Modelle können weitere Filter spezifiziert werden, die bestimmen, welche Attribute und Referenzen auf Unterschiede untersucht werden sollen, oder welche Unterschiede letztendlich übernommen werden sollen. Für eigene Datentypen kann ein *Equality Helper* spezifiziert werden, der Auskunft über Gleichheit zweier Elementen gibt. Weiter können eigene Verfahren für das Matching, Differencing und Merging verwendet werden. Außerdem können nachgelagerte Erweiterungen registriert werden, die nach den jeweiligen Phasen (s.o.) die Ergebnisse anpassen können.

3.4. Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)

Das KIELER-Projekt[6] ist ein Forschungsprojekt der Arbeitsgruppe für Echtzeitsysteme und Eingebettete Systeme an der Christian-Albrechts-Universität zu Kiel. Ziel des Projekts ist es, automatisches Layout für verschiedene Diagramme innerhalb der Modellierungsumgebung zu etablieren. KIELER wird als Open Source Anwendung unter der Eclipse Public License veröffentlicht. Wie in Abschnitt 3.1.1 beschrieben, ist KIELER eine RCA, basiert also auf der Eclipse Plattform. Abbildung 3.6 zeigt die Gliederung des Projekts.

Der Bereich Pragmatics fasst alle Aspekte der praktischen Anwendung der MDSE zusammen. Darunter fallen das Erstellen und Bearbeiten von Modellen

[5] http://www.eclipse.org/emf/compare/doc/21/developer/logical-model.html

[6] http://www.informatik.uni-kiel.de/rtsys/kieler

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

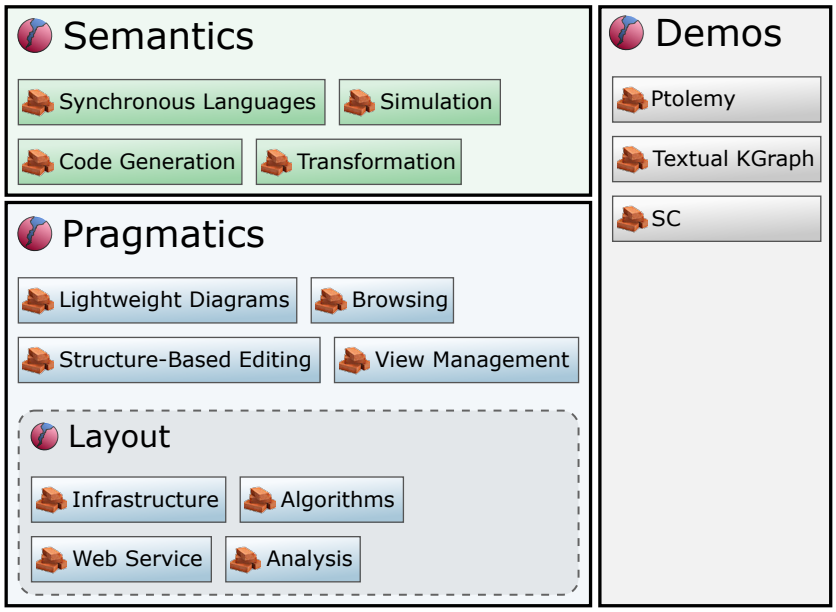


Abbildung 3.6. Struktur des KIELER-Projekts [7]

sowie die Synthese von angepassten Sichten auf diese Modelle. Unterprojekte sind zum Beispiel das KIELER Structure-based Editing (KSbasE) oder KLighD.

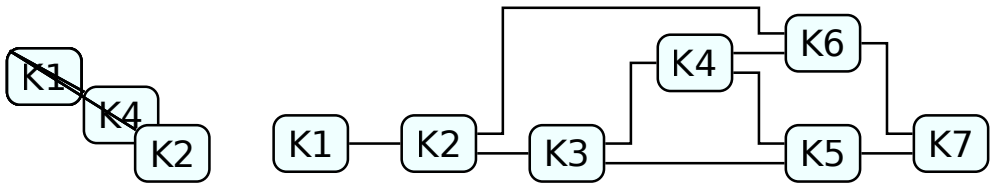
Zu dem Bereich Pragmatics zählt auch das Layout. Dieses Themengebiet bildet die eigentliche Grundlage für alles andere in KIELER. Als KIELER Layouters (KLay) werden die Layoutverfahren zusammengefasst, die im Rahmen von KIELER entwickelt wurden. Der wohl wichtigste Layouter ist KLay *Layered*, ein fortgeschrittener, Ebenen-basierter Layoutalgorithmus.

Der Bereich Semantics beschäftigt sich mit der Kompilierung und Simulation von Modellen. Er stellt eine Infrastruktur zur Verfügung, auf der die Ausführungssemantik für Metamodelle definiert werden kann.

Zusätzlich sind einige Demos vorhanden, die zu Testzwecken und zur Demonstration der anderen Bereiche verwendet werden. Hierzu zählen auch die SCCharts, die eine praxisnahe Anwendung ermöglichen.

3.5. KIELER Lightweight Diagrams (KLighD)

Das Ziel des KLighD Projekts ist es, leichtgewichtige Diagramme schnell und ohne großen Vorbereitungs- oder Konfigurationsaufwand aus einem semantischen Modell zu synthetisieren. KLighD ist Teil des KIELER-Projekts, genauer des Pragmatics-Teils. Es wird dort als universelles Tool zur Diagrammgenerierung verwendet. Schneider et al. [18] stellen das Konzept hinter KLighD näher vor. KLighD verwendet *KRendering*-Modelle zur Diagrammbeschreibung. Das Diagramm-Modell



(a) Diagramm ohne Layout

(b) Diagramm nach automatischem Layout

Abbildung 3.7. Automatisches Layout mit teilweise vorgegebenen Positionen

(*KRendering*-Modell) erweitert das semantische Modell um Informationen zur graphischen Darstellung. Es beschreibt nicht nur die Struktur des Diagramms, sondern auch das Layout (Position und Größe) sowie das Aussehen der Diagrammelemente (zum Beispiel abgerundete Rechtecke für SCChart-States). Das Diagramm letztendlich bezeichnet die graphische Abbildung, das gerenderte *KRendering*-Modell. Zur Anzeige von Modellen beliebiger Metamodelle reicht es aus, ein Mapping bereit zu stellen, das die Übersetzung vom semantischen Modell nach *KRendering* beschreibt. Auf Basis dieser formalen Beschreibung wird das Diagramm generiert, das dann gelayoutet und angezeigt wird.

Abbildung 3.7 zeigt beispielhaft, wie das von KLighD generierte Diagramm gelayoutet wird. In Abbildung 3.7(a) ist das Diagramm ohne Layout dargestellt. Zunächst ist nichts über die Lage oder Größe der Knoten bekannt, alle liegen im Nullpunkt der Diagrammfläche und haben keine Ausdehnung. Im Beispiel sind den Knoten Größen zugewiesen, um sie darstellen zu können. Den Knoten `K2` und `K4` wurden außerdem explizit Positionen zugewiesen. Das Layout-Framework führt dann eine Größenberechnung durch, die den Knoten eine Ausdehnung abhängig von ihrem Inhalt zuweist. Dies ist notwendig, damit das anschließende Layout keine Überlappungen produziert. Abbildung 3.7(b) stellt das Diagramm nach dem automatischen Layout dar. Die vorher festgelegten Positionen wurden dabei durch berechnete ersetzt.

Gemäß dem Prinzip „Überblick zuerst, zoomen und filtern, dann Details auf Anfrage“ bietet KLighD Möglichkeiten, Diagrammelemente hervorzuheben, Skalierungsfaktoren anzugeben und hierarchische Elemente zu expandieren. So können beispielsweise Simulationszustände graphisch dargestellt werden oder der Nutzer kann seinen Anforderungen entsprechend den Detailgrad der Synthese anpassen, um genau die Informationen zu erhalten, die er benötigt.

Kapitel 4

Entwurf

In diesem Kapitel wird das geplante Vorgehen näher beschrieben. Es gibt einen Überblick, welche Einschränkungen oder Voraussetzungen zu beachten sind und wie mögliche Lösungsansätze dazu aussehen.

Ziel ist es, wie in Abschnitt 1.3 beschrieben, Änderungen am zugrunde liegenden Modell so klar und minimalistisch wie möglich in das Diagramm zu übernehmen, ohne den Arbeitsfluss des Nutzers großartig zu behindern.

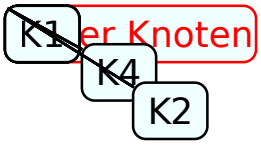
4.1. Anforderungen

Eine Lösung muss das neu synthetisierte Diagramm mit dem bereits existierenden zusammenführen. Bei dieser Zusammenführung sind bestimmte Bedingungen zu berücksichtigen:

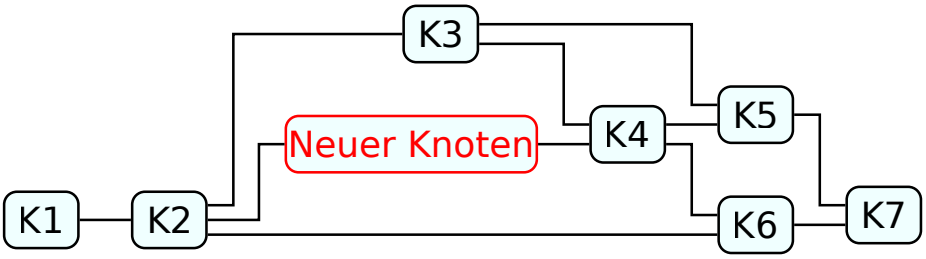
- Die Abstraktion vom Framework soll die Modularität von KLighD erhalten und Abhängigkeiten vermeiden.
- Das vorhandene Layout soll als Grundlage für das neue Diagramm verwendet werden. Dies erhöht die Nachvollziehbarkeit und vermeidet graphische Artefakte.
- Diagrammelemente sollen sich an die Größe des Inhalts anpassen.
- Augmentierungen des Diagramms durch den Viewer (Highlightings, Expansions, ...) sollen erhalten bleiben. Dadurch wird eine konsistente Ansicht auf das Diagramm sichergestellt.

4.2. Abstraktion vom Framework

KLighD kapselt die Aktualisierung des angezeigten Diagramms in sogenannte *Updatestrategien*. Eine Updatestrategie stellt eine Verbindung zwischen KLighD und dem Framework her, das für das Vergleichen und Zusammenfügen der Diagramme verwendet wird. Es können mehrere Updatestrategien registriert werden, um



(a) Von KLighD neu generiertes Diagramm



(b) Diagramm nach automatischem Layout

Abbildung 4.1. Aktualisierung des Layouts nach Einfügen eines neuen Knotens

verschiedene Frameworks nutzen zu können. Die Updatestrategie passt die zu vergleichenden Diagramm-Modelle an die Anforderungen des verwendeten Frameworks an und führt die Aktualisierung aus. Die Eigenheiten des verwendeten Frameworks bleiben so unsichtbar für KLighD und es entstehen keine direkten Abhängigkeiten zwischen KLighD und dem Framework.

4.3. Aktualisierung des Layouts

In Abschnitt 3.5 wurde bereits die Synthese von Diagrammen mit KLighD vorgestellt. Wenn bei Änderungen das Diagramm neu generiert wird, enthält es zunächst keine Informationen über Positionen und Größen, wie Abbildung 4.1(a) exemplarisch zeigt (nur explizit gesetzte Koordinaten wie die von Knoten `K2` und `K4` sind enthalten). Erst nach erneutem Layout erhält man das gewünschte Diagramm mit korrekten Positionen und Größen wie in Abbildung 4.1(b). In der Darstellung ergibt das unschöne Effekte, da zunächst alle Diagrammelemente in die obere linke Ecke gesetzt werden und sich dann erst an ihre richtige Position bewegen und ihre passende Größe annehmen.

Positions- und Größeninformationen, die durch das Layout an die Diagrammelemente angehängt wurden, sollen also nicht mit den leeren Standardwerten des neuen Diagramms überschrieben werden. KLighD speichert diese Daten als `KShapeLayout`. Enthält das `KShapeLayout` Standardwerte, so sollen diese nicht zur Berechnung der Unterschiede verwendet werden.

Wird die Position und Größe jedoch für den Vergleich der Diagramme ausmaskiert, so bleiben keine identifizierenden Werte zurück, anhand derer man die

`KShapeLayouts` während der Match-Phase voneinander unterscheiden könnte. Nur über das Matching der übergeordneten Containerelemente können die Layoutdaten einander zugeordnet werden. In der Matching-Phase ist also entsprechend eine Zuordnung nur dann zu treffen, wenn die Elternknoten ebenfalls zusammen passen.

4.4. Anpassung der Elementgröße

Die im Abschnitt 3.5 erwähnte Größenberechnung kann Diagrammelemente nur vergrößern. Enthält ein Element bereits eine Angabe zur Größe, so wird diese von KLighD als minimale Größe interpretiert. Betrachten wir einmal, was passiert, wenn ein Text in einem Knoten verkürzt wird, dessen Größe sich an den Inhalt anpassen soll. Im neu generierten Diagramm erhält der Knoten zunächst keine Größe. Das neue Diagramm wird dann mit dem bereits vorhandenen zusammengeführt. Dabei behält der geänderte Knoten die berechnete Größe von vorher, da diese nicht mit den Standardwerten des neuen Diagramms überschrieben werden soll. Damit kann die Größenberechnung den Knoten nicht verkleinern, da die angegebene Größe als vorher festgelegte minimale Größe verstanden wird.

Um dieses Problem zu umgehen, wird bei der Synthese des Diagramms eine neue Eigenschaft „minimale Größe“ gesetzt, die zunächst der gesetzten Größe entspricht. Diese minimale Größe wird vom Layout nicht angepasst und gibt daher immer an, bis zu welcher Größe ein Element verkleinert werden darf.

4.5. Erhaltung der Hilfsdaten

Nicht nur durch das Layout werden Informationen zum Diagramm hinzugefügt; auch Informationen über Expansionszustände, Highlightings oder Skalierungsfaktoren werden als *Key-Value-Paare* im Diagramm gespeichert. Welche Zusatzinformationen dieser Art genau erhalten bleiben und welche überschrieben werden sollen, ist im Voraus bekannt und kann daher einfach über einen Filter implementiert werden. Ähnlich wie bei den Layoutdaten tritt hier jedoch das Problem der Identifizierung auf. Hat sich in einem potenziellen Match zweier Zusatzinformationen der Key geändert, so kann diese Zuordnung nicht richtig sein. Der Key gibt nur an, welche Eigenschaft des Diagramm(-elements) betroffen ist. Stimmt der Key nicht überein, wurde das Hilfsdatum entweder gelöscht, gerade hinzugefügt oder es wurde noch nicht der passende Match gefunden. Es darf hier also nur eine Zuordnung getroffen werden, wenn die Keys übereinstimmen.

Kapitel 5

Implementierung

Nachdem das theoretische Vorgehen und zu erwartende Probleme bekannt sind, soll dieses Kapitel einen Einblick in die Implementierung geben.

Zunächst wird die allgemeine Struktur der Implementierung beschrieben. Der folgende Abschnitt stellt dann die Implementierung im Detail vor. Dabei werden die Anforderungen aus Kapitel 4 wieder aufgegriffen und deren Umsetzung erläutert. Zum Abschluss wird auf Probleme eingegangen, die während der Implementierung aufgetreten sind.

5.1. Allgemeine Struktur

Ergebnis der Implementierung ist das Eclipse-Plugin `de.cau.cs.kieler.klighd.incremental`, welches im Kontext des KIELER-Projekts veröffentlicht wird. Die Klasse `UpdateStrategy` stellt das Kernelement des Plugins dar. Sie implementiert den Extension-Point `de.cau.cs.kieler.klighd.extensions.updateStrategy` und wird darüber dem KLighD-Framework als verfügbare Strategie bekannt gemacht. Angelehnt an die drei Phasen des Modellvergleichs — Zuordnung, Vergleich und Zusammenfügen — sind die Klassen des Plugins in drei Pakete aufgeteilt:

- **match**: Der `KScopeFilter` legt fest, welche Modellelemente für den Vergleich beachtet werden sollen. Die `KDistanceFunction` wird zur Berechnung der Entfernung im Sinne der Ähnlichkeit verwendet.
- **diff:** Die `KDiffEngine` ist eine Anpassung der Standard-Implementierung, die den `KFeatureFilter` verwendet, um zu bestimmen, für welche Attribute oder Referenzen (*Features*) eines Elements Unterschiede berechnet werden sollen.
- **merge:** Der `KDiffFilter` filtert die gefundenen Unterschiede, sodass Daten, die vom nachfolgenden Layout noch angepasst werden, nicht aus dem neuen Modell übernommen werden.

In Abschnitt 3.5 wurde bereits die Dreiteilung in Modell, Diagramm-Modell und Diagramm vorgestellt. Zu dem Zeitpunkt, zu dem das inkrementelle Update

```java
package de.cau.cs.kieler.klighd.incremental;

public class UpdateStrategy implements IUpdateStrategy {
    :
    /**
     * Performs an update of the base view model (the view model that is currently being displayed)
     * by equalizing it the (updated) newModel.
     */
    public void update(KNode baseModel, KNode newModel, ViewContext viewContext) {

        try {
            IEObjectMatcher matcher = new ProximityEObjectMatcher(new KGraphDistanceFunction());
            :
            /* EMF Compare Konfiguration */
            :
            EMFCompare comparator =
                    EMFCompare.builder().setMatchEngineFactoryRegistry(matchEngineRegistry)
                            .setDiffEngine(new KDiffEngine(new DiffBuilder())).build();

            FilterComparisonScope scope = new FilterComparisonScope(newModel, baseModel, null);
            scope.setEObjectContentFilter(new KScopeFilter());
            Comparison comparison = comparator.compare(scope);

            IMerger.Registry mergerRegistry = EMFCompareRCPPlugin.getDefault().getMergerRegistry();
            IBatchMerger merger = new BatchMerger(mergerRegistry, new KDiffFilter());
            merger.copyAllLeftToRight(comparison.getDifferences(), new BasicMonitor());
        } catch (RuntimeException e) {
            // if incremental updating failed, apply the SimpleUpdateStrategy
            if (this.fallbackDelegate == null) {
                this.fallbackDelegate = new SimpleUpdateStrategy();
            }
            this.fallbackDelegate.update(baseModel, newModel, viewContext);
        }
    }
}
```

Listing 5.1. UpdateStrategy.java (Auszug)

aktiv wird, ist die Diagrammsynthese durch KLighD bereits vollständig abgeschlossen. Im Kontext der Implementierung sind also die Modelle, die mit EMF Compare verarbeitet werden, die generierten Diagramm-Modelle. Daher sei der Einfachheit halber im Folgenden das Diagramm-Modell als *Modell* bezeichnet; das *semantische Modell* meint wie zuvor das Modell, zu dem das Diagramm dargestellt werden soll.

5.2. Details der Implementierung

Listing 5.1 zeigt die wichtigsten Ausschnitte der Java-Klasse `UpdateStrategy`. Die Aktualisierung des Diagramms wird über die `update()` Methode initiiert. Diese nimmt zwei Modelle als Parameter entgegen: das vorhandene Modell und das

neu synthetisierte. Außerdem ist noch ein Parameter *viewContext* vorhanden, der den ViewPart in der Eclipse Workbench angibt, in dem das Diagramm dargestellt werden soll. Dieser wird hier jedoch nicht benötigt.

Kernelement des EMF Compare-Frameworks ist die Klasse `EMFCompare`, über die das Framework konfiguriert werden kann. Über diverse *Factories* und *Registries*, welche die Instanziierung der benötigten Objekte (zum Beispiel Matcher oder DiffEngines) verwalten, können eigene Anpassungen vorgenommen werden.

Zu Beginn wird in Zeile 13 ein *Matcher* definiert, der eine Implementierung des ähnlichkeitsbasierten Matchingverfahrens darstellt, aber eine eigene Anpassung der Ähnlichkeitsmetrik verwendet. Zusammen mit der `KDiffEngine`, die nur ausgewählte Attribute und Referenzen vergleicht, wird eine Instanz der Klasse `EMFCompare` erstellt, die die vorgenommenen Anpassungen kapselt und Methoden zur Anwendung von EMF Compare bereitstellt.

Es wird dann ein *Scope* erstellt, der die zu vergleichenden Modelle enthält, in diesem Fall das neu synthetisierte und das alte Modell. Dem Scope wird dann ein Filter zugewiesen, der die zu vergleichenden Elemente definiert. Der Aufruf `compare(scope)` in Zeile 25 startet die Zuordnung der Elemente und die Berechnung der Unterschiede.

Zum Schluss (Zeile 29) werden dann die Modelle anhand der gefundenen Unterschiede zusammengefügt. Dabei werden nur die Unterschiede verarbeitet, die vom `diffFilter` zugelassen werden.

Sollte das inkrementelle Update der Modelle aus einem unerwarteten Grund fehlschlagen, wird als Ausweichlösung eine `SimpleUpdateStrategie` angewendet, die das vorhandene Modell durch das neu synthetisierte ersetzt. Dies stellt sicher, dass auch bei aufgetretenen Fehlern immer ein gültiges Diagramm angezeigt wird, das dem semantischen Modell entspricht.

Mit der Implementierung der Updatestrategie konnte die Anforderung nach Abstraktion aus Abschnitt 4.2 vollständig umgesetzt werden.

5.2.1. Matching

Nachdem der Vergleich der Modelle gestartet wurde, wird zunächst das Matchmodel erstellt. Mithilfe des `KScopeFilters` wird eine Vorauswahl getroffen, welche Elemente verglichen werden. Paare von ähnlichen Elementen werden zwischen strukturellen Elementen wie Knoten, Kanten und Ports gesucht, sowie zwischen Elementen der Darstellung wie Layouts und Insets, die die Größe angeben und Rendering-Daten, die das Aussehen der Diagrammelemente beschreiben. Ausgeschlossen werden dagegen Start-, Ziel- und Knickpunkte von Kanten. Ein erster Schritt zur Erhaltung des vorhandenen Layouts (Abschnitt 4.3) wird erreicht, indem konkrete Positionen, die einzig vom automatischen Layout abhängig sind, ausgelassen und nur strukturelle Informationen und Rendering-Daten übernommen werden.

```java
package de.cau.cs.kieler.klighd.incremental.match;

public class KDistanceFunction extends EditionDistance {
    ⋮
    @Override
    public double distance(Comparison inProgress, EObject a, EObject b) {
        if (a instanceof IPropertyToObjectMapImpl && b instanceof IPropertyToObjectMapImpl) {
            // Match IPropertyToObjectMaps only for identical keys.
            IPropertyToObjectMapImpl aMap = (IPropertyToObjectMapImpl) a;
            IPropertyToObjectMapImpl bMap = (IPropertyToObjectMapImpl) b;
            if (!aMap.getKey().equals(bMap.getKey()))
                return Double.MAX_VALUE;
        } else if (a instanceof KShapeLayout && b instanceof KShapeLayout) {
            // Match KShapeLayouts only for matching containers.
            if (!haveMatchingContainers(inProgress, a, b))
                return Double.MAX_VALUE;
        } else if (a instanceof KChildArea && b instanceof KChildArea) {
            // Match KChildAreas only for matching containers.
            if (!haveMatchingContainers(inProgress, a, b))
                return Double.MAX_VALUE;
        }
        return super.distance(inProgress, a, b);
    }
    ⋮
}
```

Listing 5.2. KDistanceFunction.java (Auszug)
Berechnet die Änhlichkeit zwischen zwei Modellelementen

Während des eigentlichen Matchingprozesses wird die `KDistanceFunction` verwendet, um die Ähnlichkeit zwischen zwei Elementen zu berechnen. Sie erweitert die `EditionDistance`, eine Standardimplementierung von EMF Compare, deren drei Hauptmethoden `distance()`, `areIdentic()` und `getThresholdAmount()` überschrieben wurden.

Die Berechnung der Entfernung erfolgt in der Methode `distance()` über die Anzahl der Unterschiede sowie eine Gewichtung der Features, in denen sie auftreten (siehe dazu auch Abschnitt 3.3). Soll nur überprüft werden, ob zwei Elemente identisch im Sinne der Zuordnung sind, also sicher ein Paar bilden, so kann die Funktion `areIdentic()` verwendet werden, die unter Umständen performanter arbeitet, da nicht immer die Entfernung berechnet werden muss.

Allein über die Entfernung zweier Elemente zueinander lässt sich aber noch keine Aussage darüber treffen, ob sie wirklich dasselbe Element in verschiedenen Versionen darstellen. Ein Grenzwert legt fest, bis zu welcher Entfernung zwei Elemente einander noch zugeordnet werden sollen und ab welcher keine Ähnlichkeit mehr besteht. Die Funktion `getThresholdAmount()` berechnet diesen, indem für jedes Feature, das gesetzt wurde, die Gewichtung aufsummiert wird. Dies ergibt einen sehr allgemeinen Wert, der für die meisten Elemente funktioniert, aber in speziellen Fällen eine Anpassung erfordert. Layout-Informationen können

sich beispielsweise stark ändern (Subgraph eines Knotens wird ausgeblendet oder Ähnliches) und trotzdem weiterhin zusammengehören. Daher ist für diese Werte der Grenzwert der maximal mögliche Wert. So werden Layout-Daten einander nur über die Elternelemente zugeordnet. Knoten dagegen haben diverse Features, die optional gesetzt sein können, wie zum Beispiel aus- oder eingehende Kanten, Ports, Kindknoten und Elternknoten. Die Grenzwerte weichen stark voneinander ab und sind daher nicht brauchbar. Für Knoten wird deshalb ein empirisch festgelegter Wert verwendet, der zwar noch keine optimalen, aber zufriedenstellende Ergebnisse liefert.

Listing 5.2 zeigt die angepasste `distance()`-Funktion. Für *Property-Maps*, welche die in Abschnitt 4.5 erwähnten *Key-Value-Paare* darstellen, ist eine Zuordnung nur dann sinnvoll, falls sie identische Keys haben (Zeile 12). Anderenfalls sind es mit großer Wahrscheinlichkeit zwei verschiedene Maps, da sich bei Änderung einer Eigenschaft (Minimale Größe, Expansionsstatus, ...) maximal der Wert der Eigenschaft ändert, nicht aber der Key. `KChildAreas`, die den Bereich für Subgraphen festlegen, und `KShapeLayouts`, die in Abschnitt 4.3 schon angesprochen wurden, haben keine eigenen identifizierenden Attribute. Sie können nur dann ein Paar bilden, wenn die übergeordneten Containerobjekte ebenfalls zusammenpassen (Zeilen 16 und 20).

5.2.2. Differencing

Bei der Berechnung der Unterschiede (*Differencing*) werden die gefundenen Elementpaare aus der Matchingphase miteinander verglichen und die Unterschiede dem Matchmodel hinzugefügt. Die verwendete `KDiffEngine` ist größtenteils identisch mit der Standard-Implementierung, nur verwendet sie den angepassten `KFeatureFilter`, der festlegt, welche Features eines Elements auf Unterschiede überprüft werden sollen. Listing 5.3 zeigt den spezialisierten Filter.

Standardmäßig ignoriert EMF Compare abgeleitete Referenzen und Aggregationen. Für den Vergleich von KLighD-basierten Modellen sind diese jedoch wichtig und müssen daher ebenfalls mit in Betracht gezogen werden. Die angepasste Version ignoriert deshalb keine Referenzen (Zeile 8).

Diverse Attribute, die vom Layout angepasst werden und daher erhalten bleiben sollen, werden bis jetzt noch nicht berücksichtigt. Positionen, Größen und Abstände sollen nur dann in das neue Modell übernommen werden, wenn sie explizit gesetzt sind und sich von den Standardwerten unterscheiden. Ein Attribut wird folglich dann ignoriert, wenn die folgenden drei Bedingungen gelten:

- Das Element, dessen Attribut gerade untersucht wird, hat eine Entsprechung im neu generierten Modell. Damit ist sichergestellt, dass das Element nicht gelöscht wurde.

```java
package de.cau.cs.kieler.klighd.incremental.diff;

public class KFeatureFilter extends FeatureFilter {

    @Override
    protected boolean isIgnoredReference(Match match, EReference reference) {
        // Consider all references for differences.
        return false;
    }

    @Override
    public Iterator<EAttribute> getAttributesToCheck(final Match match) {
        if (match.getLeft() == null) {
            return match.getRight().eClass().getEAllAttributes().iterator();
        }
        return Iterators.filter(match.getLeft().eClass().getEAllAttributes().iterator(),
                new Predicate<EAttribute>() {
                    public boolean apply(EAttribute attribute) {
                        // Don't detect differences in layout attributes with default value.
                        // These will get adjusted by following layout.
                        return !(IGNORED_ATTRIBUTES.contains(attribute)
                                && !match.getLeft().eIsSet(attribute));
                    }
                });
    }

    /** List of attributes that will be ignored. */
    private static final List<EAttribute> IGNORED_ATTRIBUTES = ImmutableList.of(
            KLayoutDataPackage.eINSTANCE.getKShapeLayout_Xpos(),
            KLayoutDataPackage.eINSTANCE.getKShapeLayout_Ypos(),
            KLayoutDataPackage.eINSTANCE.getKShapeLayout_Width(),
            KLayoutDataPackage.eINSTANCE.getKShapeLayout_Height(),
            KLayoutDataPackage.eINSTANCE.getKInsets_Left(),
            KLayoutDataPackage.eINSTANCE.getKInsets_Right(),
            KLayoutDataPackage.eINSTANCE.getKInsets_Top(),
            KLayoutDataPackage.eINSTANCE.getKInsets_Bottom(),
            KLayoutDataPackage.eINSTANCE.getKPoint_X(),
            KLayoutDataPackage.eINSTANCE.getKPoint_Y());
}
```

Listing 5.3. KFeatureFilter.java
Filtert, in welchen Attributen und Referenzen nach Unterschieden gesucht wird.

- ▷ Das Attribut ist eines der potenziell zu ignorierenden Attribute, die in der Liste `IGNORED_ATTRIBUTES` definiert werden.
- ▷ Das Attribut ist nicht gesetzt oder ist gleich dem Standardwert.

Diese Beschränkung des Suchbereichs komplettiert die Anforderung nach der Erhaltung des vorhandenen Layouts (Abschnitt 4.3). Der nachfolgende Layoutvorgang kann jetzt die aus dem vorherigen Modell übernommenen Werte an das neue Modell anpassen.

5.2.3. Merging

Alle Voraussetzungen für das zusammenfügen der Modelle sind nun erfüllt: Zusammengehörende Elemente sind gefunden und Unterschiede berechnet. Doch nicht alle Unterschiede sollen übernommen werden. Immer noch ungelöst ist bis jetzt das Problem der Größenanpassung an kleinere Inhalte und des Erhaltens von Hilfsdaten. Daher wird für das Mergen ein `KDiffFilter` verwendet, der die Auswahl der gefundenen Unterschiede nochmals verfeinert. Listing 5.4 zeigt den Filter.

Zusätzliche Informationen, die als Property-Map an Style-Elemente (*KRenderings*) angehängt werden, sollen erhalten bleiben, da sie nicht von KLighD generiert werden, sondern nachträglich hinzugefügt werden. Ausnahme sind die in `MERGE_PROPERTIES` aufgelisteten Eigenschaften, die vorberechnete Daten zur Performanceoptimierung oder die Referenz zum dargestellten semantischen Element betreffen. Diese drei Spezialfälle werden weiterhin übernommen, alle übrigen Unterschiede werden heraus gefiltert und nicht angewendet. Wie in Abschnitt 4.5 gefordert, bleiben die Hilfsdaten erhalten.

Weiter werden Unterschiede gefiltert, die das `INITIAL_NODE_SIZE`-Flag betreffen. Das Flag gibt an, ob die Größe eines Elements bereits vom Layouter angepasst wurde oder noch mit der bei der Synthese gesetzten übereinstimmt. Im neu generierten Modell ist das Flag nicht vorhanden und wird daher als *true* angenommen, das heißt, die angegebene Größe ist noch die initiale Größe. Im vorhandenen Modell wurde das Flag vom Layout-Framework auf *false* gesetzt. Das heißt, Unterschiede, die hier untersucht werden, sind vom Typ `DELETE`. Ziel ist es, die Unterschiede so zu filtern, dass Knoten beim nachfolgenden Layout verkleinert werden können. Dazu ist es entscheidend, ob im neuen Modell eine Größe angegeben ist.

Ist dies der Fall, so soll diese Größe vom nachfolgenden Layout als minimale Größe erkannt werden. Das `INITIAL_NODE_SIZE`-Flag muss also gelöscht werden, damit es vom Layouter als *true* erkannt wird. Der gerade untersuchte Unterschied wird also belassen, um das Flag zu löschen.

Ist im neuen Diagramm keine Größe gesetzt, so wird die Größe, die durch das vorherige Layout im vorhandenen Diagramm eingetragen ist beibehalten (siehe

```java
package de.cau.cs.kieler.klighd.incremental.merge;

public class KDiffFilter implements Predicate<Diff> {

    private final List<IProperty<?>> MERGE_PROPERTIES = ImmutableList.<IProperty<?>> of(
            GridPlacementUtil.CHILD_AREA_POSITION,
            GridPlacementUtil.ESTIMATED_GRID_DATA,
            KlighdInternalProperties.MODEL_ELEMEMT);

    public boolean apply(Diff diff) {
        if (diff instanceof ReferenceChange) {
            ReferenceChange refChange = (ReferenceChange) diff;
            if (refChange.getValue() instanceof IPropertyToObjectMapImpl
                    && refChange.getValue().eContainer().eClass().getEPackage()
                            .equals(KRenderingPackage.eINSTANCE)
                    && !MERGE_PROPERTIES.contains(((IPropertyToObjectMapImpl) refChange.getValue())
                            .getKey()))
                // Omit all IPropertyToObjectMapImpls that are attached to KRendering-related
                // data structures except for those mentioned in MERGE_PROPERTIES.
                // Those are set by the KLighD translation infrastructure rather than the
                // Piccolo-based rendering binding; they must be updated/replaced during the merge!
                // The remaining ones have been attached to the displayed view model to cache
                // information and shall be preserved!
                return false;
            else if (refChange.getValue() instanceof IPropertyToObjectMapImpl
                    && refChange.getKind() == DifferenceKind.DELETE) {
                IPropertyToObjectMapImpl map = (IPropertyToObjectMapImpl) refChange.getValue();
                if (map.getKey().equals(KlighdProperties.INITIAL_NODE_SIZE)) {
                    KShapeLayout layout = (KShapeLayout) refChange.getMatch().getLeft();
                    if (layout != null && layout.getHeight() == 0 && layout.getWidth() == 0)
                        // Keep the INITIAL_NODE_SIZE flag (which is false after the first
                        // Layoutiteration) (i.e. delete the difference), if no size is set in
                        // the newly generated Knode.
                        return false;
                    // If there is a size set, reset the flag (i.e. keep the difference, so the
                    // property gets deleted), such that the set size gets layouted.
                }
            }
        }
        return true;
    }
}
```

Listing 5.4. KDiffFilter.java
Trifft eine genauere Auswahl, welche Unterschiede gemerged werden sollen.

vorheriger Abschnitt). Diese Größenangabe soll aber vom nachfolgenden Layouter nicht als minimale Größe interpretiert werden. Daher wird der Unterschied, der das `INITIAL_NODE_SIZE`-Flag löschen würde, ausgefiltert. Das Flag bleibt als *false* im gemergten Diagramm erhalten.

Diese Filterung ermöglicht die Anpassung von Knoten an kleinere Inhalte, wie in Abschnitt 4.4 beschrieben.

5.3. Probleme während der Implementierung

Nach der kompletten Reorganisation des EMF Compare-Frameworks verfährt EMF Compare 2.1 beim Verarbeiten von Unterschieden nach einer *Top-Down*-Strategie, während EMF Compare 1.3 nach einem *Bottom-Up*-ähnlichen Ansatz vorging.

Angenommen, eine Gruppe von Elementen ist in einem Modell geändert worden. Beim Zusammenführen der Modelle erstellt EMF Compare 1.3 zunächst ein Teilmodell, das die aktualisierten Elemente enthält. Sind alle Unterschiede übernommen, so wird im zu aktualisierenden Modell die geänderte Gruppe von Elementen komplett durch das aktualisierte Teilmodell ersetzt. Das hat zur Folge, dass die Anzeige nur einmal aktualisiert wird. Mit der neuen Version 2.1 wird jeder abgearbeitete Unterschied direkt in das finale Modell eingefügt. Für jede dieser Änderungen wird das Diagramm neu gerendert und die Anzeige aktualisiert. Um dieses Problem zu umgehen, wurde das Update der Diagrammdarstellung in einen asynchronen Prozess ausgelagert und mit einem Timeout versehen. Der Timeout wird bei jeder Änderung des Modells zurückgesetzt. So wird die Wahrscheinlichkeit, dass Änderungen am Modell während des Renderings vorgenommen werden, drastisch reduziert.

Ein weiteres Problem entsteht durch die direkte Übertragung der Unterschiede. Werden Referenzen geändert, so kann es passieren, dass zwischenzeitlich ein ungültiger Zustand herrscht. Beispielsweise besteht die Änderung des Ziels einer Kante aus zwei Schritten: Löschen und neu Hinzufügen der Kante. Bei der Weiterverarbeitung des Modells durch KLighD oder in Kombination mit der oben erwähnten Aktualisierung der Anzeige treten an verschiedenen Stellen ungesicherte Zugriffe auf nicht gesetzte Werte auf. Durch die Bottom-Up-Verarbeitung der vorherigen EMF Compare Version stellte dies kein Problem dar, da dort immer ein komplett gültiges Modell vorlag. Mit der neuen Version mussten diese ungesicherten Zugriffe zunächst gefunden werden. Es musste dann untersucht werden, ob die fehlerhafte Referenz gefahrlos ignoriert werden kann oder ob weitere Vorkehrungen, wie beispielsweise ein ordnungsgemäßer Abbruch, vorgenommen werden müssen.

Kapitel 6

Fazit und Ausblick

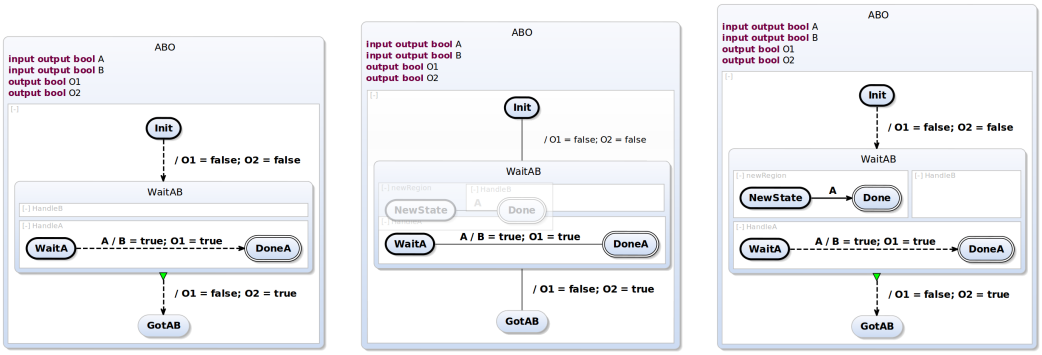
Dieses abschließende Kapitel fasst die erreichte Lösung und die Anwendbarkeit in der Praxis zusammen und gibt einen Ausblick über zukünftige Arbeiten.

6.1. Zusammenfassung

Entwickelt wurde ein Eclipse-Plugin, das die Verwendbarkeit von EMF Compare mit KLighD wiederherstellt. Durch grundlegende Änderungen im EMF Compare-API war die vorhandene Implementierung nicht mehr kompatibel. Die Eigenheiten und geänderten Verhaltensweisen des neuen EMF Compare-Frameworks wurden untersucht und Anforderungen aufgestellt, welche die neue Implementierung erfüllen soll.

Das inkrementelle Update wurde als Updatestrategie gekapselt. Dies vermeidet direkte Abhängigkeiten zwischen KLighD und dem EMF Compare-Framework. Es wurden eigene Filter implementiert, die konkrete Postions- und Größeninformationen vom Vergleich der Diagramm ausnehmen. So kann das vorhandene Layout als Grundlage für das neue Diagramm verwendet werden, was graphische Artefakte durch umherwandernde Diagrammelemente vermeidet. Die Verarbeitung von Größenangaben wurde speziell an KLighD und die automatische Größenberechnung angepasst, sodass sich Elemente an die Größe des Inhalts anpassen. Hilfsdaten, die nach der Synthese der Diagramme angelegt werden, bleiben erhalten und ermöglichen eine konsistente Sicht auf das Diagramm.

Abbildung 6.1 zeigt den Verlauf des inkrementellen Updates eines SCCharts beim Einfügen einer neuen Region. Das Diagramm 6.1(a) zeigt den Zustand vor der Änderung. Zu erkennen ist die eingeklappte Region oben im Macrostate WaitAB, deren Zustand während des Aktualisierens erhalten bleiben soll. Im mittleren Diagramm 6.1(b) lässt sich gut erkennen, wie Änderungen am Modell in das Diagramm übernommen werden, ohne das komplette Diagramm neu anzuzeigen. Diagramm 6.1(c) schließlich stellt den Zustand nach abgeschlossener Aktualisierung dar. Die neue Region wurde in das Diagramm übernommen und der Expansionsstatus von schon bestehenden Elementen wurde erhalten.



(a) Vor der Aktualisierung **(b)** Während der Aktualisierung **(c)** Nach der Aktualisierung

Abbildung 6.1. Verlauf des inkrementellen Updates beim Einfügen einer zusätzlichen Region in ein SCChart.

6.2. Ausblick

Das Konzept des inkrementellen Updates für Diagramme ist mit der vorliegenden Implementierung erreicht. Jedoch gibt es noch diverse Verbesserungsmöglichkeiten. Viele Probleme, die durch die Andersartigkeit des neuen EMF Compare in Verbindung mit dem KLighD-Framework auftraten, konnten im zeitlichen Rahmen dieser Bachelorarbeit nur rudimentär gelöst werden. Durch die Probleme, die während der Implementierung auftraten, konnte neues Wissen über das Verhalten von EMF Compare und dessen Zusammenspiel mit KLighD gewonnen werden. Die Implementierung stellt eher einen *Proof-of-Concept* als eine endgültige Lösung dar, die für den Produktiveinsatz geeignet ist.

Insbesondere ist der Timeout vor dem Start des Renderings eines Diagramms (siehe dazu Abschnitt 5.3) keineswegs eine elegante Lösung. Eine Möglichkeit wäre es, das Einfügen der Unterschiede in das bestehende Diagramm-Modell so zu gestalten, dass Unterschiede zu Transaktionen zusammengefasst werden. Dadurch könnte erreicht werden, dass das Rendern des Diagramms nicht unnötig oft angestoßen wird und darüber hinaus nur Modelle gerendert werden, die nach dem *KRendering*-Metamodell gültig sind.

Literatur

[1] Daniel Archambault und Helen C. Purchase. „Mental Map Preservation Helps User Orientation in Dynamic Graphs“. In: *Proceedings of the 20th International Conference on Graph Drawing*. GD'12. Redmond, WA: Springer-Verlag, 2013, S. 475–486. ISBN: 978-3-642-36762-5. DOI: 10.1007/978-3-642-36763-2_42. URL: http://dx.doi.org/10.1007/978-3-642-36763-2_42.

[2] Giuseppe Di Battista, Peter Eades, Roberto Tamassia und Ioannis G. Tollis. Graph Drawing: Algorithms for the Visualization of Graphs. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998. ISBN: 0133016153.

[3] Lars Bendix und Pär Emanuelsson. „Diff and Merge Support for Model Based Development“. In: *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*. CVSM '08. Leipzig, Germany: ACM, 2008, S. 31–34. ISBN: 978-1-60558-045-6. DOI: 10.1145/1370152.1370161. URL: http://doi.acm.org/10.1145/1370152.1370161.

[4] Cédric Brun und Alfonso Pierantonio. „Model differences in the eclipse modeling framework“. In: *UPGRADE, The European Journal for the Informatics Professional* 9.2 (2008), S. 29–34.

[5] Hauke Fuhrmann. „‚On the Pragmatics of Graphical Modeling'“. Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.

[6] Hauke Fuhrmann, Miro Spönemann, Michael Matzen und Reinhard von Hanxleden. „Automatic layout and structure-based editing of UML diagrams“. In: *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED'10)*. Dresden, März 2010.

[7] David Harel. „Statecharts: A visual formalism for complex systems“. In: *Science of Computer Programming* 8.3 (Juni 1987), S. 231–274.

[8] Michael Jünger und Petra Mutzel, Hrsg. Graph Drawing Software. Springer, 2004. ISBN: 978-3-642-62214-4.

[9] Michael Kaufmann und Dorothea Wagner, Hrsg. Drawing Graphs: Methods and Models. LNCS 2025. Berlin, Germany: Springer-Verlag, 2001. ISBN: 3-540-42062-2.

[10] Mark Kofman und Erik Perjons. MetaDiff–a Model Comparison Framework. 2003. URL: http://metadiff.sourceforge.net.

[11] D.S. Kolovos, D. Di Ruscio, A. Pierantonio und R.F. Paige. „Different models for model matching: An analysis of approaches to support model differencing“. In: *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on*. Mai 2009, S. 1–6. DOI: 10.1109/CVSM.2009.5071714.

[12] Patrick Könemann. „Capturing the Intention of Model Changes“. In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II*. MODELS'10. Oslo, Norway: Springer-Verlag, 2010, S. 108–122. ISBN: 3-642-16128-6, 978-3-642-16128-5. URL: http://dl.acm.org/citation.cfm?id=1929101.1929114.

[13] Patrick Könemann. „Model-independent Differences“. In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. CVSM '09. Washington, DC, USA: IEEE Computer Society, 2009, S. 37–42. ISBN: 978-1-4244-3714-6. DOI: 10.1109/CVSM.2009.5071720. URL: http://dx.doi.org/10.1109/CVSM.2009.5071720.

[14] R. Lutz, D. Wurfel und S. Diehl. „How Humans Merge UML-Models“. In: *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. Sep. 2011, S. 177–186. DOI: 10.1109/ESEM.2011.26.

[15] Rachel A. Pottinger und Philip A. Bernstein. „Merging Models Based on Given Correspondences“. In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, S. 862–873. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315525.

[16] Arne Schipper. „‚Layout and Visual Comparison of Statecharts‘“. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf. Diploma Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dez. 2008.

[17] 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.

[18] 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.

[19] Antoine Toulmé. „Presentation of EMF Compare Utility“. In: *Eclipse Modeling Symposium at Eclipse Summit Europe 2006*. Esslingen, Germany, Okt. 2006.