FAKULTÄT FÜR !NFORMATIK

Konflikterkennung in der Modellversionierung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

ausgeführt von

Philip Langer
Matrikelnummer 0325934

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuerin: O.Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel
Mitwirkung: Univ.-Ass. Dipl.-Ing. Dr. Martina Seidl
Univ.-Ass. Mag. Dr. Manuel Wimmer

Wien, 12.02.2009

(Unterschrift Verfasser) (Unterschrift Betreuerin)

Eidesstattliche Erklärung

Philip Langer, Liebhartsgasse 4/11, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Februar 2009

Philip Langer

Danksagung

Zu aller erst gilt mein Dank meinen Eltern, Regine und Hans, die nicht nur durch ihre finanzielle Hilfe, sondern auch durch deren moralische Unterstützung mein Studium ermöglichten und mir stets den Rückhalt boten, der schlussendlich auch zum Abschluss meines Studiums führte.

Außerdem möchte ich mich herzlich bei meiner Betreuung und damit bei Gerti Kappel, Martina Seidl und Manuel Wimmer bedanken, die mich auf freundschaftliche Weise vor und während der Entstehung dieser Arbeit mit großem Einsatz unterstützten. Ich freue mich, auch weiterhin mit diesen Personen arbeiten zu dürfen.

Weiterhin bedanke ich mich besonders bei meiner Freundin, Andrea Rucker, die während der Entstehung dieser Arbeit großes Verständnis für viele arbeitsreiche Abende zeigte und mir nicht nur durch eifriges Korrekturlesen eine große Hilfe war.

Nicht zuletzt danke ich auch meinen Studienkolleginnen und -kollegen, die in den unzähligen produktiven und unproduktiven Stunden, die ich mit ihnen in den vergangenen fünf Jahren verbrachte, meine Studienzeit abwechslungsreich, spannend und vor allem äußerst unterhaltsam gestalteten.

Kurzfassung

In den letzten Jahren gewannen Softwaremodelle als zentrale Artefakte der Softwareentwicklung zunächst mit dem CASE-Ansatz (Computer Aided Software Engineering) und später mit dem MDE-Ansatz (Model Driven Engineering) immer mehr an Bedeutung. Sie dienen mittlerweile nicht nur zur Dokumentation und zur Bildung des gemeinsamen Verständnisses sondern auch als Grundlage für die Generierung eines lauffähigen Systems. Der modellgetriebene Ansatz ist in der Softwareentwicklung zu einer etablierten und weit verbreiteten Methode geworden.

An dem Softwareentwicklungsprozess ist in größeren Softwareprojekten üblicherweise eine Vielzahl an EntwicklerInnen beteiligt. Diese verfeinern in einem zumeist iterativen Prozess das zu entwickelnde System und passen es stetig an sich verändernde Anforderungen, Verständnisse und laufend zu treffende Designentscheidungen an. Für eine erfolgreiche Durchführung eines Softwareprojekts ist daher ein effizientes Änderungs- und Konfigurationsmanagement ausschlaggebend und ermöglicht erst die effektive Zusammenarbeit mehrerer EntwicklerInnen. Traditionelle Versionierungssysteme implementieren größtenteils nur zeilenbasierte Konflikterkennung und bieten daher keine ausreichende Unterstützung für Erkennung und Resolution von Konflikten bei der Versionierung von Softwaremodellen, die eine graphenbasierte Form aufweisen.

Das Ziel dieser Arbeit ist die Erarbeitung und Implementierung einer intelligenten, adaptierbaren Konflikterkennung für die Versionierung von Modellen. Es wird ein Rahmenwerk geschaffen, das *out-of-the-box* einen verwendbaren Vergleichs- und Konflikterkennungsalgorithmus für Modelle bietet und bei Bedarf durch die Erstellung spezifischer Beschreibungen für eine konkrete Modellierungssprache oder -domäne adaptiert werden kann. Diese sprachspezifischen Erweiterungen erhöhen die Qualität und Genauigkeit der Konflikterkennung und verhindern dadurch unnötige Konfliktmeldungen und manuelle Eingriffe.

Abstract

Within the last years, the popularity and appearance of model-driven software development (MDSD) has increased significantly. The model-driven paradigm induces a shift from code-centric to model-centric development. Software models are considered as first class entites building the basis for generating an executable software. Nowadays, MDSD is a widely accepted technology helping software developers to accelerate and simplify their work by raising the abstraction level of the software artefacts they create.

Large-scale software is usually produced by a high number of software developers, who are often working in parallel and spread all over the world. Therefore, the version management of software models is crucial for an effective, collaborative software development. One of the most important components of a versioning system for model artefacts is a precise difference and conflict detection, which is capable of identifying all operations executed by the developers and the eventually resulting conflicts that arise at merging independently changed models. Without an appropriate difference and conflict detection system, software developers are forced to find occuring problems manually, which is a time consuming process. Especially complex operations, refactorings and language specific conflicts are hard to identify and can lead to serious malfunctions in the produced software if they remain undiscovered.

The main goal of this work is the development and implementation of an intelligent and adaptable conflict detection mechanism for models in the context of versioning. A framework will be presented which is able to precisly calculate the differences and detect conflicts in a generic way. Users may extend this system by providing language specific definitions which will be used to refine the functionality for specific modeling languages. In that way, the quality of the difference and conflict detection can be improved to minimize the need for manual interaction.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In den letzten Jahren gewannen Softwaremodelle als zentrale Artefakte der Softwareentwicklung (*first-class entities*) vorerst mit dem CASE[1]-Ansatz und später mit dem MDE[2]-Ansatz immer mehr an Bedeutung. Sie dienen mittlerweile nicht nur zur Dokumentation und Bildung des gemeinsamen Verständnisses sondern auch als Grundlage für die Generierung eines lauffähigen Systems. Der modellgetriebene Ansatz konnte sich in der Softwareentwicklung zu einer weit verbreiteten Methode etablieren.

An dem Softwareentwicklungsprozess sind in größeren Softwareprojekten überlicherweise eine Vielzahl an EntwicklerInnen beteiligt. Diese verfeinern in einem zumeist iterativen Prozess das zu entwickelnde System und passen es stetig an sich verändernde Anforderungen, Verständnisse und laufend zu treffende Designentscheidungen an. Für eine erfolgreiche Durchführung eines Softwareprojekts ist daher ein effizientes Änderungs- und Konfigurationsmanagement (SCM[3]) ausschlaggebend und ermöglicht erst die effektive Zusammenarbeit mehrerer EntwicklerInnen.

Traditionelle SCM-Systeme (z.B. CVS [1] und SVN [3]) zielen auf die Versionierung von Textdateien wie beispielsweise dem Quellcode ab. Softwaremodelle können zwar in Textdateien serialisiert werden (z.B. mittels XMI[4]), unterliegen aber einer graphenbasierten Struktur. Zeilenorientierte Konflikterkennung von Modellen liefert daher inadäquate und unzureichende Ergebnisse. Beispielsweise kann eine einzige strukturelle Änderung eines Modells zu einer Manipulation mehrerer, oftmals nicht zusammenhängender Zeilen in einem serialisierten Modell führen. Dieses Problem wird in der Literatur oft als *impedance mismatch* [18, 26] oder *abstraction mismatch* bezeichnet [40]. Textbasierte Technologien bieten einem/r SoftwareeintwicklerIn bei der modellgetriebenen Softwareentwicklung demnach kaum Unterstützung und können häufige, manuelle Eingriffe zur Konflikterkennung und -behebung kaum verhindern [40].

Der Bedarf an Versionierungssystemen, die SoftwareentwicklerInnen bei der Versionierung von Softwaremodellen unterstützen, ist daher unbestritten [40]. Die großen Herausforderungen bei der Entwicklung solcher Systeme ist der einheitliche Vergleich von unabhängig veränderten Modellversionen, die Erkennung der aus dem Vergleich resultierenden Konflikte, die, soweit möglich, automatische Auflösung der erkannten Konflikte, sowie die Erzeugung eines neuen, zusammengeführten Modells.

Die Realisierung eines solchen Systems wird durch einige Faktoren deutlich erschwert. Die vermutlich größte Herausforderung stellt die Vielzahl an existierenden Modellierungssprachen, deren Semantik nur in impliziter Form definiert ist. Für jede dieser

[1] CASE - Computer Aided Software Engineering
[2] MDE - Model Driven Engineering
[3] SCM - Software Configuration Management
[4] XMI - XML Metadata Interchange

Sprachen eine spezielle Implementierung zu entwickeln ist neben dem immensen zeitlichen Aufwand auch wegen der zukünftigen Verwendbarkeit für ständig neu entstehende Sprachen keine geeignete Lösung. Eine allgemeine, vollkommen generisches Versionierungssystem bietet wiederum zu wenig Unterstützung bei der Versionierung von Modellen.

1.2 Zielsetzung

Das Ziel dieser Diplomarbeit ist die Erarbeitung und Implementierung einer intelligenten, adaptierbaren Konflikterkennung für die Versionierung von Modellen. Es soll ein Rahmenwerk geschaffen werden, das *out-of-the-box* einen verwendbaren Vergleichs- und Konflikterkennungsalgorithmus für Modelle bietet und bei Bedarf durch die Erstellung spezifischer Beschreibungen für eine konkrete Modellierungssprache oder -domäne adaptiert werden kann. Diese sprachspezifische Erweiterung erhöht die Qualität und Genauigkeit der Konflikterkennung und verhindert dadurch unnötige Konfliktmeldungen und manuelle Eingriffe.

Von besonderer Bedeutung ist es, die sprachspezifische Schicht so leichtgewichtig, intuitiv und deskriptiv wie möglich zu halten. Es wäre nicht sinnvoll, die Benutzer des Systems bei der Anpassung an eine neue Modellierungssprache zu zeitaufwendigen Implementierungen zu zwingen. Wünschenswert wäre diese Anpassung auf eine modellbasierte Beschreibung der spezifischen Merkmale zu beschränken.

Außerdem muss eine Abhängigkeit von speziellen Editoren verhindert werden, um den BenutzerInnen des Systems die freie Wahl über ihre verwendeten Werkzeuge offen zu lassen. Das bedeutet, dass zur Ermittlung der Änderungen eine editorspezifische Erweiterung keine anzustrebende Lösung ist, sondern eine plattformweite oder retrospektive Errechnung der Modellmodifikationen erforderlich wird. Darüber hinaus soll diese Konflikterkennung mit Refactorings wie beispielsweise der Umbenennung einer Klasse und den sich daraus ergebenden Konsequenzen umgehen können. Diese stellen bei der Versionierung eine große Herausforderung dar. Refactorings erzeugen eine Vielzahl an Änderungen an vielen unterschiedlichen Stellen. Durch diese hohe Verteilung vieler Änderungen ist die Wahrscheinlichkeit mit konkurrierenden Änderungen anderer Personen zusammen zu stoßen sehr groß. Trotz der vielen Änderungen steckt hinter einem Refactoring aber nur eine semantische Operation mit einer klaren Absicht. Können Refactorings daher erkannt und bei der Konflikterkennung berücksichtigt werden, verringert dies üblicherweise die Anzahl der zu meldenden Konflikte erheblich.

In Abbildung 1.1 ist eine Übersicht über das präsentierte Gesamtsystem dargestellt. Zwei Personen editieren parallel das Ausgangsmodell und erzeugen damit jeweils eine neue Version des Ausgangsmodells. Das System ermittelt die atomaren Modifikationen durch einen sprach- und editorunabhängigen Modellvergleich und stellt diese in modellbasierter Form dar. Diese Operationen werden anschließend von dem Operationsinterpreter analysiert und mit weiteren Informationen (zusammengesetzte, sprachspezifische Operationen oder Refactorings) angereichert. Um beispielsweise zusammengesetzte, sprachspezifische Operationen abbilden und in das System einbinden zu können, wird ein beispielgetriebener und anwenderfreundlicher Prozess vorgestellt. Schließlich erzeugt der Operationsinterpreter aus beiden Operationsfolgen einen Konfliktreport mit allen gefunden generischen sowie sprachspezifischen Konflikten und deren Beschreibungen.

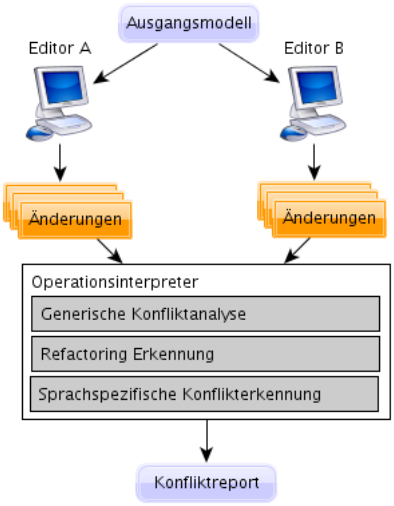

Abbildung 1.1: Überblick über das Gesamtsystem.

1.3 Aufbau dieser Arbeit

Die vorliegende Arbeit ist in acht weitere Kapitel und eine abschließende Zusammenfassung in Kapitel 10 unterteilt. Zu Beginn wird in Kapitel 2 in das Umfeld dieser Arbeit eingeführt, um anschließend die theoretischen Grundlagen des Modellvergleichs, der Repräsentation der Unterschiede, sowie der Konflikterkennung in Kapitel 3 zu erläutern. In Kapitel 4 werden bestehende Ansätze und Implementierungen dieses Bereichs vorgestellt und mit einander verglichen. Dem folgt eine Ableitung der Anforderungen an die Konflikterkennung innerhalb eines Versionierungssystems für Modelle in Kapitel 5. Bevor nun die im Zuge dieser Arbeit durchgeführte Implementierung in Kapitel 7 besprochen wird, gibt das Kapitel 6 einen Überblick über die verwendete Infrastruktur. Kapitel 8 stellt ein Konzept vor, wie sprachspezifische, zusammengesetzte Operationen und Refactorings auf benutzerfreundliche und beispielgetriebene Weise definiert und anschließend erkannt werden können. Dem folgt Kapitel 9, das Techniken und Methoden bespricht, wie Konflikte auf Basis der ermittelten Deltadokumente erkannt werden können.

2 Umfeld

Ziel dieses Kapitels ist es, dem Leser/der Leserin ein Grundverständnis über das Umfeld und die Hintergründe dieser Arbeit zu vermitteln. Die Konflikterkennung ist in diesem Zusammenhang ein Teil des Managements gleichzeitiger Änderungen an Softwareartefakten durch mehrere BenutzerInnen. Dieses Änderungsmanagement wird als Teil der Versionierung im Rahmen der Softwareentwicklung verstanden. Die Versionierung selbst ist wiederum Teil des Konfigurationsmanagements. Die Aufgaben eines Konfigurationsmanagementsystems, sowie der Verionierung werden in den folgenden Unterkapiteln genauer erläutert.

2.1 Konfigurationsmanagement

Softwarekonfigurationsmanagement (SCM) bezeichnet die Handhabung von Änderungen während der gesamten Entwicklungszeit einer Software. SCM ermöglicht einen systematischen und rückführbaren Entwicklungsprozess, sodass zu jedem Zeitpunkt ein wohldefinierter Status der Software existiert [20].

Konfigurationsmanagement, das die Grundlage für das SCM darstellt, wurde erstmals in den 50er Jahren in der Luft- und Raumfahrt betrieben, nachdem eine unzureichend dokumentierte, technische Änderung die erfolgreiche Produktion eines Raumfahrzeugs verhinderte [20]. Einige Zeit später traten ähnliche Probleme auch bei der Softwareentwicklung auf. Dies führte zu der Entstehung des Software-Konfigurationsmanagements als eigene Informatikdisziplin in den späten 70er Jahren und brachte Produkte wie beispielsweise SCCS[1] und RCS[2] hervor. Die stetig angestiegene Komplexität der Systeme, die ständige Verbesserung der Hardware und nicht zuletzt der Druck des schnelllebigen geschäftlichen Umfeldes trieb die Weiter- bzw. Neuentwicklung vieler Konfigurationsmanagementsysteme weiter an [20].

Moderne SCM-Systeme bieten eine große Anzahl an Funktionen, die in [16] wie folgt kategorisiert wurden.

- **Komponenten.** Das SCM-System muss dem/der BenutzerIn ermöglichen, alle Artefakte eines Softwaresystems speichern, finden und darauf zugreifen zu können. Hierbei ist zu berücksichtigen, dass diese Artefakte in mehreren Versionen und Varianten vorliegen.
- **Struktur.** Das SCM-System muss die Struktur des zu entwickelnden Softwaresystems unterstützen und angemessen darstellen, sodass Zusammenhänge zwischen den verschiedenen Artefakten erkennbar bleiben.

[1] SCCS - Source Code Control System (M. J. Rochkind 1975)
[2] RCS - Revision Control System (W. F. Tichy 1982)

- **Konstruktion.** Das SCM-System soll eine Erzeugung der lauffähigen Software in der aktuellen Version oder auch in einer älteren Version unterstützen.
- **Prüfung.** Das SCM-System muss Änderungsinformationen sammeln und speichern, die den Zeitpunkt, die ändernde Person und den Grund einer Änderung beinhalten. Dies fördert die Nachvollziehbarkeit der Evolution und Erkennung von Problemen in einer Software.
- **Buchführung.** Das SCM-System soll Statistiken über den Softwareentwicklungsprozess erzeugen können.
- **Kontrolle.** Das SCM-System soll die BenutzerInnen darin unterstützen, die Bedeutung einer Änderung zu verstehen und diese beispielsweise einem Bugfix oder einem Featurerequest zuordenbar zu machen.
- **Prozess.** Das SCM-System soll den Entwicklungsprozess unterstützen, sodass eine Aufgabenverwaltung und -verteilung im Kontext der Softwareentwicklung integrierbar ist.
- **Team.** Das SCM-System muss die Zusammenarbeit aller EntwicklerInnen erleichtern und in diesem Zusammenhang auch eine Erkennung und Bereinigung von Konflikten, die zwischen mehreren Versionen bestehen, ermöglichen.

Man wird nur selten all diese oben genannten Funktionalitäten in einem einzigen Produkt finden. Oftmals bieten die Entwicklungsumgebungen (z.B. Eclipse [2], NetBeans[3], ...) als zentrale Infrastruktur die Möglichkeit, andere Produkte über Plug-Ins zu integrieren, sodass alle benötigten, oben genannten Funktionalitäten zur Verfügung stehen.

Estublier et al. [20] unterteilen diese Funktionalitäten in die drei technischen Bereiche *Produktsupport*, *Toolsupport* und *Prozesssupport*. Diese spiegeln auch die Evolution der SCM-Systeme in den letzten Jahrzehnten wider, die, begonnen mit versioniertem Dateimanagement über Integration dieser Versionierung in Tools, zur vollen Prozesskontrolle führte.

- **Produktsupport** bildet die Kernfunktionalitäten des SCM. Dies umfasst in erster Linie die *Versionierung*, die das historische Aufzeichnen und das Änderungsmanagement beinhaltet. Da ein Softwareprojekt aber nicht Datei für Datei behandelt werden kann, werden zusätzlich Beziehungen zwischen Softwareartefakten verwaltet, um so Inkonsistenzen weitgehend zu vermeiden.
- **Toolsupport** besteht aus den Schnittstellen, die ein/e BenutzerIn verwenden kann, um mit der Versionsverwaltung zu kommunizieren (*Workspace Control*). Außerdem fällt die Konstruktion (*Building*) der Software in diesen Bereich.
- **Prozesssupport** ist der bisher wohl jüngste Bereich, der gerade erst in große SCM-Produkte Einzug gefunden hat. Zu Beginn beschränkte sich die Unterstützung auf einfache Prozesse, die vom Versionskontrollsystem vorgegeben wurden. Diese folgten üblicherweise dem *Check-Out*/*Change*/*Check-In* Modell. Moderne, high-end SCM-Systeme erlauben mittlerweile das Design und die Durchführung eigens definierter Prozesse.

[3] NetBeans IDE – Java Entwicklungsumgebung (http://www.netbeans.org).

2.2 Versionierung

Wie schon im vorigen Abschnitt angesprochen, ist die *Versionierung* das Kernmodul und der älteste Bestandteil eines Software-Konfigurationsmanagementsystems [20]. Ziel der Versionierung ist die historische Archivierung älterer oder alternativer Versionen aller Artefakte einer Software während der gesamten Entwicklungszeit.

Im Laufe der Entwicklungsgeschichte von Versionierungssystemen entstanden die unterschiedlichsten Ansätze und Realisierungen, um die Anforderungen an ein solches System zu erfüllen. Der Aufbau und die zugrunde liegenden Konzepte eines Versionierungssystems können in verschiedene *Versionsmodelle* unterteilt werden. Ein Versionsmodell beschreibt, wie versionierte Objekte und deren Versionen abgebildet werden, wie Versionsidentifikation und -organisation implementiert sind und definiert die Operationen für das Abfragen von bestehenden Versionen oder das Erstellen neuer Versionen [14].

Bei der Betrachtung eines Versionsmodells können zwei Dimensionen, der *Produktraum* und der *Versionsraum*, unterschieden werden. Ein Versionsmodell definiert in diesem Sinne den Aufbau und die Organisation dieser beiden Dimensionen sowie deren Zusammenspiel. Die Kombination von Produkt- und Versionsraum wird auch *versioned object base* genannt und beinhaltet daher den zweidimensionalen Raum von Objekten und Zeit (Veränderung und Version) [14].

2.2.1 Produktraum

Der Produktraum beschreibt die Struktur des zu versionierenden Softwareprodukts ohne die Veränderung und Versionierung des selben zu berücksichtigen [13, 14]. In dieser Dimension existieren Softwareobjekte und Beziehungen zwischen diesen Objekten. Softwareobjekte sind die zu versionierenden Artefakte des Softwareprodukts. Die Granularität dieser Artefakte variiert in den unterschiedlichen Versionierungsystemen. Die meisten definieren jedoch eine Datei im Sinne des Dateisystems als atomares Objekt. Domänenspezifische Systeme können beispielsweise durch Syntaxbäume oder -graphen ein Objekt feingranularer betrachten. Diese Betrachtung ist jedoch naturgemäß sprachenabhängig implementiert. Zusäztlich können Objekte im Produktraum miteinander in Beziehung stehen. Diese Beziehungen sind meist Kompositions- oder Abhängigkeitsbeziehungen.

2.2.2 Versionsraum

Der Versionsraum definiert, wie der Name schon vermuten lässt, die Versionen eines Softwareobjekts, wobei das Softwareobjekt nur als abstrakter Bestandteil aus einem anderen Raum (Produktraum) betrachtet wird. Diese Dimension bildet die Veränderung über die Zeit ab und stellt Befehle bereit, bestimmte Versionen abzurufen und neue zu erzeugen. Eine Version setzt sich aus einem Tupel, bestehend aus einem Objekt des Produktraums und einem des Versionsraums zusammen [13, 14]. Alle versionierten Objekte teilen sich gemeinsame Eigenschaften wie beispielsweise eine Versions-ID und andere Werte. Diese werden Invarianten genannt, da sie im Lauf der Zeit unverändert bleiben. Welche Eigenschaften das sind, ist von System zu System sehr unterschiedlich [14]. Neue Versionen können mit zwei unterschiedlichen Absichten erstellt werden.

EntwicklerInnen wollen beispielsweise bei der Fehlerbehebung in einem Softwareobjekt die ursprüngliche Version ersetzen. Die alte, fehlerhafte Version wird nicht mehr benötigt und weiterentwickelt. In diesem Fall spricht man von einer *Revision*. Eine neue Revision ist somit der Nachfolger der jeweils älteren Version. Revisionen befinden sich daher auf der Zeitachse sequenziell. Im Gegensatz dazu existieren *Varianten*. Varianten können koexistieren und erzeugen daher eine parallele Zeitachse, auf der sie sich als Revisionen wiederum weiter entwickeln können. Dies kann sinnvoll sein, wenn ein Softwareobjekt für unterschiedliche Plattformen existiert oder eine neue Technologie unterstützen soll, ohne die Verwendung der alten gleich vollständig zu ersetzen. Eine parallele Zeitachse kann auch gleichzeitig für mehrere oder alle Artefakte existieren. In einem solchen Fall wird von einem *Branch* oder Zweig gesprochen.

Im Versionsraum wird auch festgelegt, wie die Differenzen (Deltas) der Artefakte abgebildet werden. Man unterscheidet hierbei zwischen symmetrischen und gerichteten Differenzen (*symmetric* und *directed deltas*) [13, 14, 31]. Bei der Darstellung der Unterschiede als symmetrische Deltas werden die spezifischen, unterschiedlichen Eigenschaften beider Versionen abgebildet. Es handelt sich um die Vereinigung der beiden Differenzen (siehe Gleichung 2.1).

$$\triangle (v_1, v_2) = (v_1 \setminus v_2) \cup (v_2 \setminus v_1) \tag{2.1}$$

Bei gerichteten Deltas werden die Unterschiede als Sequenz von Änderungsoperationen abgebildet, die, wenn angewendet auf eine Version, das Artefakt zu der nächsten Version überführen (siehe Gleichung 2.2).

$$\triangle (v_1, v_2) = op_1, \ldots, op_n \tag{2.2}$$

Jedes versionierte Objekt ist ein Container für ein Set V an Versionen. Man kann Versionssysteme nun danach unterscheiden, wie dieses Set an Versionen definiert ist. Beim *extensional versioning* ist V als Vektor von Versionen definiert (siehe Gleichung 2.3) [13,14]. Alle Versionen sind explizit und können direkt mit einer Versions-ID identifiziert und abgefragt werden.

$$V = \{v_1, \ldots, v_n\} \tag{2.3}$$

Den Gegensatz zur *extensionalen* Versionierung stellt die *intensionale* Versionierung dar. Eine Version wird nunmehr nicht als Vektor von Versionen abgebildet, sondern durch ein Prädikat c beschrieben (siehe Gleichung 2.4). In diesem Fall sind Versionen implizit definiert, sodass je nach Ausprägung von c eine Version, abhängig von verschiedenen Beschränkungen, konstruiert werden kann [13,14]. Intensionale Versionierung ist daher eher dem Variantenmanagements zuzuordnen und ist insbesondere dann sinnvoll, wenn die Anwendung direkt aus der Versionierung gebaut wird und dieser Build beispielsweise für eine Zielplattform (Linux, Windows, ...) parametrisiert werden muss. In diesem Fall ist c eine Funktion für unterschiedliche Zielplattformen (z.B. `osWindows()`, `osLinux()`, ...). Ein plattformabhängiges Objekt erfüllt dann jeweils nur eine dieser Funktionen. Beim Build kann dann eine Beschränkung auf `osLinux() == true` gesetzt werden, sodass nur jene (plattformabhängigen) Artefakte mit einbezogen werden, die

diese Beschränkung erfüllen. Der Build wird jedoch üblicherweise nicht vom Versionierungssystem selbst vorgenommen. Moderne Softwareplattformen, beispielsweise Java oder Ruby, können so modular aufgebaut werden, dass ein plattformabhängiger oder datenbankspezifischer Build nicht mehr notwendig ist. Außerdem ist die Handhabung und Wartung intensionaler Systeme komplexer und aufwendiger als ihre extensionalen Gegenstücke. Der Bedarf an intensionaler Versionierung hat daher im Laufe der Jahre und mit dem Gewinn an Flexibilität der Softwaresprachen nachgelassen, sodass die meisten heutigen VCS keine intensionale Versionierung mehr implementieren.

$$V = \{v|c(v)\} \tag{2.4}$$

Orthogonal zur Unterscheidung von intensionaler und extionsionaler Versionierung steht der Unterschied zwischen *statusbasierter* und *änderungsbasierter* Versionierung [14]. Bei der statusbasierten Versionierung ist eine Version durch den Status eines versionierten Objektes definiert. Eine Version entspricht daher einer zeitlich sequentiellen Revision oder zeitlich koexistierenden Variante. Die änderungsbasierte Versionierung ändert diese Definition einer Version. Eine Version wird hier als Änderung der Basisversion gesehen. Jede Änderung ist eindeutig identifizierbar und betrifft genau ein Artefakt. Diese Kategorisierung bezieht sich also auf die Beschreibung oder Zusammensetzung der Version selbst. Die Begriffe *state-based* und *change-based* haben aber auch im Zusammenhang mit dem *Merge* (siehe Abschnitt 3), insbesondere dem Vergleich zweier Revisionen im Zuge des Merges, große Bedeutung. Es wird daher weiter unten erneut auf *state-based*, *changed-based* und auch *operation-based merging* [30, 31] eingegangen (siehe Kapitel 3.2, 3.3 und 3.4).

Versionierungssysteme können auch in Bezug auf die Granularität der Versionierung voneinander abweichen. In [14] werden drei verschiedene Stufen der Granularität genannt. Diese sind *(i)* Produktversionierung, *(ii)* totale Versionierung und *(iii)* Komponentenversionierung. Die Produktversionierung ist mittlerweile die wohl am meisten angewendete Granularität. Hierbei sind alle Änderungen, Revisionen und Varianten global und es gibt nur einen gemeinsamen Versionsraum für alle Artefakte. Die Popularität begründet sich vermutlich in der einfachen und intuitiven Form der Versionierung. Nachteilig ist dennoch die fehlende Modularität des Versionsraums. Dies ist jedoch bei modernen Softwaresystemen kein allzu großes Problem, da es meist nur eine gültige Konfiguration gibt und beispielsweise Plattformabhängigkeiten oder ähnliches innerhalb des Systems selbst gekapselt werden können. Sollten dennoch mehrere Varianten benötigt werden, können diese in einem parallelen Versionsraum überführt werden (*Branch*). Bei der Komponentenversionierung werden atomare Objekte (meistens Dateien), also ausschließlich die Blätter des Produktbaums, versioniert. Die totale Versionierung generalisiert die Komponentenversionierung, indem hier nicht nur die Blätter versioniert werden, sondern auch zusammengesetzte Objekte wie beispielsweise Ordner. Der große Nachteil dieser beiden letzten Versionierungsarten ist die weitaus komplexere Auswahltechnik. Es ist nicht mehr möglich mit einer Version einen Status des gesamten Produktraums zu identifizieren. Vielmehr müssen durch Regeln oder Annotationen konsistente Zusammensetzungen des gesamten Produkts beschrieben werden. Dies ist jedoch ein zeit- und fehlerintensiver Vorgang.

In diesem Kapitel wurde das Umfeld und die Grundlagen der Versionierung vorgestellt. Insbesondere wurden drei Merkmale der Versionierung beschrieben. Diese betreffen einerseits die Ableitungsform der Versionen, die entweder sequentiell bei der

extensionalen Versionierung oder durch ein Prädikat bei der intensionalen Versionierung vorgenommen wird. Andererseits können Versionen entweder statusbasiert, also durch den Status eines Artefakts, oder änderungsbasiert, also über die Änderungen, die in einem Artefakt vorgenommen wurden, dargestellt werden. Das letzte Merkmal dreht sich um die Granularität der Versionen. Hierbei können Komponenten entweder individuell oder als gesamtes Produkt versioniert werden.

3 Vergleich, Deltarepräsentation und Konflikterkennung

In diesem Kapitel werden die theoretischen Grundlagen, verschiedene Herangehensweisen und deren Eigenschaften bei dem Vergleich von Modellen, der Deltarepräsentation und schlussendlich der Konflikterkennung besprochen. Zuvor erfolgt jedoch eine Beschreibung des übergeordneten Prozesses, dem Merge, der diese und andere Aufgaben umfasst.

3.1 Merge

Bei der Versionskontrolle kann auf zwei entgegengesetzte Mechanismen zurückgegriffen werden – einerseits *pessimistische* und andererseits *optimistische* Versionierung [31]. Der Unterschied dieser beiden Vorgehensweisen liegt im Umgang mit konkurrierenden Änderungen.

Bei der *pessimistischen Versionierung* werden alle Artefakte, die ein/e SoftwareentwicklerIn zu ändern plant, für andere Manipulationen gesperrt. Die meisten Versionskontrollsysteme unterstützen diesen Mechanismus und bieten die Möglichkeit, Objekte mit einem *lock* zu annotieren. Auf diese Weise können gleichzeitige Änderungen verhindert und dadurch möglicherweise entstehenden Konflikten im Voraus vorgebeugt werden. Der Nachteil ist jedoch offensichtlich. Bei größeren, parallel arbeitenden Entwicklergruppen ist folgedessen laufend ein erheblicher Teil der Software gesperrt, sodass mit großer Wahrscheinlichkeit mehrere EntwicklerInnen ihre Aufgaben nicht vollständig durchführen können, ohne auf den *lock* einer anderen Person zu stoßen.

Wird im Gegensatz dazu *optimistische Versionierung* verwendet, arbeitet jede Person uneingeschränkt auf einer persönlichen Kopie (*working copy*). Der Preis für diese Flexibilität muss erst beim anschließenden Einchecken der Änderungen bezahlt werden. Denn mit der selben Wahrscheinlichkeit von gleichzeitig durchgeführten Änderungen wie oben, kommen diese auch hier vor. In diesem Fall müssen die unabhängigen Änderungen zu einer neuen, gemeinsamen Version zusammengeführt werden (*merge*).

In Abbildung 3.1 ist der Prozess der optimistischen Versionierung dargestellt. Es existiert eine Version *v0* eines Softwareartefakts. Zwei Personen, in der Abbildung die linke und rechte Person, checken diese Version aus. Die zweite Person kann dieses Objekt ebenfalls auschecken, da beim optimistischen Verfahren keine Sperre gesetzt wird. Beim Auschecken wird am lokalen Rechner der beiden Personen eine Arbeitskopie der gemeinsamen Version *v0* angelegt. Alle Veränderungen, die diese Personen von nun an vornehmen, betreffen vorerst nur ihre Arbeitskopie. Ist eine Person mit ihren Änderungen fertig, checkt diese Person ihre veränderte Kopie (*v0'*) wieder ein (*commit*).

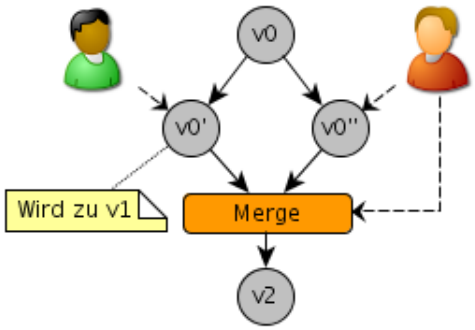

Abbildung 3.1: Optimistische Versionierung und Zusammenführung.

Da keine zwischenzeitliche Änderung von anderen Personen in die zentrale Objektbasis (*object base* oder Repository) geladen wurde, ist das Einchecken problemlos möglich. Es wird also eine neue Version als Nachfolger von *v0* mit der Bezeichnung *v1* erzeugt. Die Version *v1* hat nun den Inhalt von *v0'*. Sobald die zweite Person ihre Änderungen einspielen will, wird eine gleichzeitige Änderung festgestellt. Die Vorgängerversion *v0* der veränderten Arbeitskopie (*v0"*) ist nicht mehr die aktuellste Version, die auch *head revision* genannt wird. Diese zweite Person muss nun einen Merge von *v1* mit *v0"* vornehmen und das Resultat dieser Zusammenführung als neue Version *v2* einchecken. Damit das Repository nicht wieder einen Änderungskonflikt meldet, muss beim Checkin der Konflikt (*v0* ist Vorgänger von der *working copy*, aber nicht *head revision*) als aufgelöst *resolved* markiert werden. Die Version *v2* beinhaltet nun die Zusammenführung von *v0"* und *v1*. Sobald die erste Person wiederum ihre *working copy* auf den neuesten Stand bringt (*update*), arbeitet sie an *v1* weiter.

Trotz der Notwendigkeit eines Merges ist bei steigender Anzahl an entwickelnden Personen die *optimistische Versionierung* vorzuziehen. Traditionelle VCS sperren beim pessimistischen Ansatz die gesamte Datei, da noch nicht bekannt ist, welche Teile der Datei geändert werden. Bei der optimistischen Versionierung und dem resultierenden Merge sind die Unterschiede der beiden Versionen bekannt. Es kommt daher in vielen Fällen zwar zu einem gleichzeitigen Ändern einer Datei aber nicht zwingenderweise zu einem manuell aufzulösenden Konflikt. Dies hängt natürlich stark von der Änderung, der Art des geänderten Softwareartefakts und der Qualität des Merges ab.

Abbildung 3.2 zeigt eine mögliche Aufspaltung des Mergeprozess in seine Subaufgaben. Diese sind durch abgerundete Rechtecke dargestellt. Die übrigen Rechtecke repräsentieren die durch die Subaufgaben erzeugten Artefakte. Bei dem Vergleich werden die vorhandenen Informationen über die zusammenzuführenden Objekte analysiert, um deren Unterschiede zu erkennen. Diese gefundenen Deltas werden durch den Änderungsbericht repräsentiert. Ausgehend von diesem Bericht werden im nächsten Schritt mögliche Konflikte erkannt und in einen Konfliktbericht geschrieben. Die Konfliktresolutionskomponente kann nun für jeden Konflikt des Berichts unter Umständen auch mit Hilfe von Benutzerinteraktion Konfliktlösungen erzeugen. Da eine Konfliktlösung wiederum neue Konflikte erzeugen kann, ist der Merge als iterativer Prozess aufzufassen. Sind schlussendlich alle Konflikte beseitigt, können die beiden Objektversionen zusammengeführt werden.

Die vorliegende Arbeit konzentriert sich auf den Vergleich der Versionen und die Konflikterkennung. In den folgenden Kapiteln werden daher nur jene Aspekte bzw. Ansätze

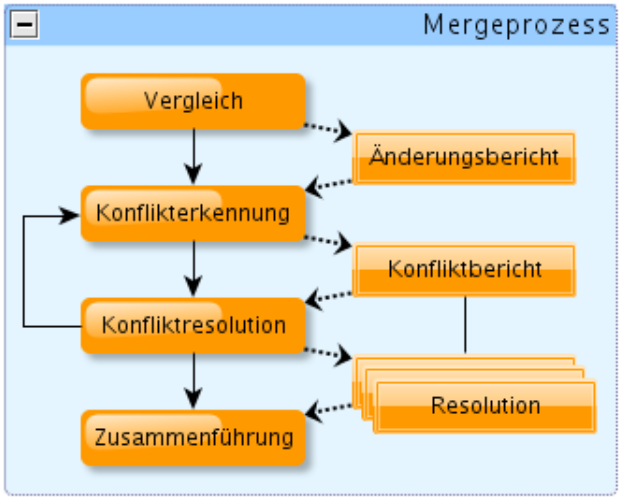

Abbildung 3.2: Aufspaltung des Mergeprozesses.

aufgezeigt und analysiert, die für diese beiden Phasen des Mergeprozesses von Bedeutung sind.

3.2 Vergleich von Versionen

In Abbildung 3.2 sind die Phasen eines Mergeprozesses dargestellt. Dieses Kapitel durchleuchtet nun verschiedene Aspekte und Möglichkeiten des Vergleichs (erste Phase). Neben dem Anwendungsfall *Merge* sind für einen Vergleichsalgorithmus noch andere Anwendungsfälle für den Vergleich von Versionen denkbar. Ein effektiver Vergleichsalgorithmus für Modelle kann beispielsweise auch wiederverwendet werden, um Modelltransformationen zu testen [40].

Match

Um zwei Versionen eines Objekts zu vergleichen und Unterschiede zu finden, müssen zuerst die Äquivalenzen erkannt werden. Dieser *Match* kann auf zwei unterschiedliche Weisen implementiert werden. Die erste und einfachere Methode ist die Verwendung von eindeutigen Attributen. Haben zwei Objekte die selbe UUID[1] können sie als äquivalent erkannt werden. Diese Technik hat entscheidende Vorteile. Neben ihrer Einfachheit, Effizienz und Skalierbarkeit können so auch Objekte als Objekte mit dem selben Ursprung erkannt werden, auch wenn beide Seiten jeweils stark verändert wurden. Der große Nachteil dieser Technik ist die Abhängigkeit von den UUIDs und damit von den Werkzeugen, in denen die Objekte erzeugt und editiert werden. Diese müssen immer eine neue UUID vergeben, sobald ein neues Objekt erstellt wird, und dürfen diese UUID im weiteren Verlauf nicht verändern. Außerdem müssen diese UUIDs auch genau so platziert oder ausgezeichnet werden, wie es vom Vergleichsalgorithmus erwartet

[1] UUID - Universally Unique Identifier

wird. Das ist aber erwartungsgemäß von Tool zu Tool nicht immer einheitlich. Außerdem ist so der Vergleich auf hinter einander folgende Versionen beschränkt und kann keine Äquivalenzen erkennen, wenn die beiden Versionen nicht vom selben Ursprung ausgehen. Die Alternative ist die Anwendung von Heuristiken auf Elementnamen und andere Eigenschaften der Objekte wie beispielsweise `isAbstract` oder `isInterface`. Oft finden hier Distanzberechnungen wie die Levenshtein- [28] oder Hamming-Distanz Anwendung. Zwei Objekte können so aber nicht als äquivalent identifiziert, sondern nur mit einer gewissen Wahrscheinlichkeit als gleich eingestuft werden. Es entsteht daher eine gewisse Unschärfe. Speziell wenn das selbe Objekt auf beiden Seiten stark verändert wurde, können große Ungenauigkeiten auftreten.

Basis des Vergleichs

Vergleichsmethoden können bezüglich der Informationen klassifiziert werden, welche sie zum durchführen des Vergleichs benötigen. Im einfachsten Fall sind es nur die beiden zu vergleichenden Status eines Objekts. Man spricht in diesem Fall von einem statusbasierten *2-Wegsvergleich* (*2-way state-based*). Unter der Annahme, dass der *Match* alle Äquivalenzen erkannt hat, können so neue Objekte und Unterschiede als solche erkannt werden. Es kann jedoch nicht festgestellt werden, ob das selbe Element auf beiden verändert wurde, oder nur auf einer Seite. Außerdem kann der 2-Wegsvergleich nicht differenzieren, ob ein Element nur auf einer Seite erzeugt, oder auf der anderen Seite entfernt wurde. Diese beiden Erkenntnisse sind jedoch für die Zielerreichung – die größtmögliche Automatisierung der Zusammenführung umzusetzen – sehr wichtig. Der statusbasierte 2-Wegsvergleich erzwingt daher bei nahezu jedem erkannten Unterschied eine Benutzerinteraktion, da nicht mit Sicherheit ermittelt werden kann, wie der gemeinsame Nachfolger aussehen soll. Große Verbesserungen bringt eine Miteinbeziehung der gemeinsamen Vorgängerversion. Der Vergleich findet daher zwischen den beiden aktuellen Status unter Berücksichtigung des gemeinsamen Ursprungs statt. Es handelt sich daher um einen *3-Wegsvergleich*. Steht die Vorgängerversion zur Verfügung, kann erkannt werden, ob ein Element auf der einen Seite gelöscht oder auf der anderen Seite hinzugefügt wurde. Genauso ist es möglich eine Änderung auf einer Seite als solche zu erkennen und von einer gleichzeitigen Änderung auf beiden Seiten zu unterscheiden. Aufgrund der genannten Vorteile und der Tatsache, dass die Vorgängerversion in Versionierungssystemen üblicherweise verfügbar ist, unterstützen die meisten Systeme den 3-Wegsvergleich [31]. Änderungen können naturgemäß noch exakter ermittelt werden, wenn diese direkt, z.B. aus der Entwicklungsumgebung, verfügbar sind. In diesem Fall spricht man von einem *änderungsbasierten Vergleich* (*changed-based*). Es werden also nicht die beiden Status des Objekts miteinander verglichen, sondern die Änderungen selbst einander gegenübergestellt. Änderungen sind atomare Transformationen, die einen Status in einen anderen überführen. Die Änderungen an sich sind bei änderungsbasierten Systemen generisch und umfassen daher meist nur die allgemeinen Transformationen *Add(e)*, *Delete(e)* und *Change(e, e')*, wobei *e* bzw. *e'* das der Transformation unterzogene Ursprungselement ist. Eine besondere, erweiterte Form des änderungsbasierten Vergleichs ist der *operationsbasierte Vergleich* (*operation-based*) [31], der die Vorzüge der änderungsbasierten Vorgehensweise erst vollständig zur Geltung bringt. Dieser wurde erstmals in [30] im Zuge des operationsbasierten Merges vorgestellt. Dabei werden die Änderungen als explizite Operationen oder Transformationen von der Vorgängerversion zur aktuellen Version ausgedrückt. Die Vorgängerversion wird daher, ähnlich dem statusbasierten 3-Wegsvergleich, berücksichtigt. Bei einer

operationsbasierten Vorgehensweise wird der Merge als Zusammenführung der Operationen, angewandt auf den Produktraum, gesehen. Die Operationen ähneln meistens stark den Operationsmöglichkeiten der Entwicklungsumgebung [31] und ermöglichen eine noch effizientere und genauere Analyse des Änderungsvorganges, sodass in weiterer Folge auch semantische Konflikte oder Refactorings [21] berücksichtigt werden können. Nach [30] ist der operationsbasierte Ansatz ein allgemeinerer Ansatz als der statusbasierte. Besonders der statusbasierte 3-Wegsvergleich, sofern er die Unterschiede in gerichteter Form darstellt (siehe Sektion 2.2.2 und 3.3) und damit Transformationen aus den drei zur Verfügung stehenden Versionen ableitet, kann als spezielle, eingeschränkte Form des operationsbasierten Ansatzes betrachtet werden. Um den Nutzen der operationsbasierten Technik zu maximieren, ist die Abbildung der Operationen oftmals auch sprachen- oder zumindest domänenabhängig (z.B. Umbenennung der Methode m in Klasse c). Eine detaillierte Beschreibung der Vorzüge dieser Vorgehensweise im Bezug auf die Konflikterkennung ist im Kapitel 3.4 dargestellt. Diese Vorteile des änderungs- bzw. operationsbasierten Vergleichs werden jedoch durch die starke Abhängigkeit von der Entwicklungsumgebung, die ja die Änderungen aufzeichnen und abbilden muss, getrübt.

Metamodellabhängigkeit

Die zuvor genannte Sprachenabhängigkeit ist auch gleichzeitig eine weitere Eigenschaft des Vergleichs. Ein Vergleichsalgorithmus kann abhängig von einem Metamodell, also einer Sprache wie UML oder Java sein. In diesem Fall können oft effizientere Vergleiche mit genauerem Ergebnis ermöglicht werden (siehe Kapitel 3.4). Die Wiederverwendbarkeit ist jedoch naturgemäß auf die Sprache bzw. Domäne beschränkt. Eine der beiden Alternativen, diese Abhängigkeit zu verhindern, ist ein Lifting der Abhängigkeit auf Metametamodellebene, also jene Ebene, die die Sprache selbst beschreibt. Im Modellierungsbereich wird dafür oft MOF[2] verwendet. Die andere Alternative ist eine vorherige Transformation der Objektstatus auf ein gemeinsames Format durchzuführen.

Dem Vergleich vorangehende Transformation der Objektstatus

Einige Verfahren machen sich eine vor dem Vergleich angewendete Transformation der zu vergleichenden Objekte zu Nutze. Diese Verwendung einer Transformation kann einen von zwei entgegengesetzten Gründen haben. Der Grund ist entweder die Spezialisierung oder die Verallgemeinerung bezüglich der Modellierungssprache. Bei der Spezialisierung wird der Objektstatus in ein spezielles, domänenabhängiges Format gebracht, um eine exaktere und speziellere Analyse eines oder mehrerer bestimmter Aspekte (z.B. Struktur, Semantik) der Objektversionen vornehmen zu können. Die Transformation selbst ist immer metamodellabhängig und erfüllt einen bestimmten Zweck. Beispielsweise kann über die Transformation einer Sprache in einen Syntaxbaum ein Syntaxbewusstsein geschaffen werden [10] und Syntaxfehler im zusammengeführten Artefakt verhindert werden. Ein anderes Beispiel ist die Umwandlung in eine semantische Sicht, die eine Entdeckung semantischer Aspekte möglich macht [5]. Der zweite Grund ist die Generalisiserung. Hier werden Objektstatus in eine abstraktere Sprache transformiert, um die Algorithmen, die nur auf dieser abstrakten Metametamodellebene operieren, auf

[2]MOF - Meta Object Facility von der Object Management Group (OMG).

viele Sprachen wiederverwendbar zu machen. Beispielsweise wurden in [37] für einige Sprachen (BOC ADONIS, NoMagic MagicDraw UML, Fujaba, ...) Transformationen implementiert, die die konkreten Modelle in ein abstraktes, gemeinsames Modell umwandeln.

3.3 Repräsentation der Deltas

Nachdem der Vergleich der beiden Objektversionen durchgeführt und alle Unterschiede ermittelt wurden, müssen diese für die Konflikterkennung in einem Änderungsbericht aufbereitet werden. Dieser Abschnitt beschäftigt sich mit den unterschiedlichen Möglichkeiten der Deltarepräsentation (siehe in Abbildung 3.2: *Änderungsbericht*). Die Art, wie Deltas abgebildet werden und welche Information diese beinhalten, setzen natürlich einen geeigneten Vergleichsalgorithmus voraus, der jene abzubildende Daten ermittelt. Die hier genannten Eigenschaften von Deltaabbildungen sind daher eng mit den zuvor besprochenen Eigenschaften des Vergleichs und den zugrundeliegenden Algorithmen verbunden.

Die Unterscheidung zwischen symmetrischen und gerichteten Deltas wurde in Kapitel 2.2.2 bereits diskutiert. Bei symmetrischen Deltas berechnet der Vergleich die Unterschiede der beiden Versionen als mengentheoretisches Komplement [14, 31]. Man spricht von symmetrischer Differenz, da es keine Richtung (v_1 zu v_2 oder umgekehrt) gibt und beide Versionen eine gleichberechtigte Rolle spielen. Diese Darstellungsform hat ihren Ursprung in der Speicherplatzoptimierung und findet meistens nach einem statusbasierten 2-Wegsvergleich ihre Anwendung. Das Unix Programm *diff* [23] erzeugt beispielsweise eine symmetrische Differenzdarstellung für Textdateien.

Gerichtete Deltas sind im Gegensatz dazu eine Sequenz von Operationen oder Transformationen, die ein Objekt in ein anderes überführen [31]. Diese Deltarepräsentation wird meist nach einem statusbasierten 3-Wegsvergleich oder natürlich bei änderungsbasierten Vorgehensweisen angewendet. Diese Sequenz von Operationen ist aber nicht, wie symmetrische Deltas, immer minimal. Werden Änderungen von einer Entwicklungsumgebung automatisch mitgeschrieben, sind diese üblicherweise in hohem Maße redundant. Elemente, die einer Änderung unterzogen und anschließend gelöscht oder ein weiteres Mal geändert wurden, erzeugen unnötig lange, redundante Operationssequenzen [30]. Es empfielt sich also die Verwendung einer Minimierungsfunktion, die redundante Schritte entfernt, wie dies beispielsweise in [4] präsentiert wurde.

Für die Visualisierung von Objektdifferenzen ist der symmetrische Ansatz besser geeignet [12]. Die Unterschiede sind in dieser Form intuitiv erfassbar, da sie in der Sprache des verglichenen Modells neben den unveränderten, und damit dem User bekannten Objekten dargestellt werden können. Bei der Darstellung von symmetrischen Differenzen wird meist auf Coloring-Techniken zurückgegriffen, die jeweils unveränderte, gelöschte und hinzugefügte Elemente in einer anderen Farbe oder Schattierung auf der Benutzeroberfläche auftragen. Gerichtete Differenzen sind für BenutzerInnen nicht besonders intuitiv. Bei dieser Darstellungsform wird nur eine Liste an Operationen dargestellt, die, speziell bei größeren Änderungen, nicht in vergleichbarer Geschwindigkeit vom User erfasst werden können. Die Visualisierung ist jedoch unabhängig von der internen Repräsentation. Es ist durchaus möglich, symmetrische Visualisierungsformen von gerichteten Differenzen abzuleiten und dem oder der BenutzerIn zu präsentieren.

Außerdem haben gerichtete Deltas entscheidende Vorteile für die interne, technische Repräsentation. Sie sind meist transformativ und können daher direkt auf eine Version angewendet werden, um die nächsten Version herzuleiten [12]. Wenn diese Transformationsschritte unabhängig von einer bestimmten Version und ihre Vorbedingungen erfüllt sind, können diese daher auch auf andere (frühere, spätere oder parallele) Versionen des Objekts angewendet werden. So kann bei einer Zusammenführung zweier unabhängigen Versionen eine Operation nach der anderen auf einen Objektstatus durchgeführt werden, um die vereinigte Version zu erzeugen. Speziell bei der intensionalen Versionierung (siehe Kapitel 2.2.2), bei der es keine sequenzielle Versionsfolge gibt, ist die Wiederholbarkeit und Umkehrbarkeit von Operationen besonders nützlich [31].

Die Herleitung von Transformationen ist, im Gegensatz zum operationsbasierten Ansatz, beim statusbasierten 3-Wegsvergleich komplizierter. Es ist nicht leicht möglich, aus den konkreten Modifikationen eine abstrakte, sinnhafte Operation abzuleiten, die die Absicht der EntwicklerInnen widerspiegelt und unabhängig vom geänderten Objekt anwendbar ist. Dies ist besonders dann der Fall, wenn diese Operationen aus mehreren, atomaren Modifikationen bestehen wie beispielsweise bei Refactorings [21]. Beim operationsbasierten Ansatz ist es leichter, komplexere Transformationen zu erkennen oder sogar deren domänenspezifische Semantik aufzuzeichnen. Diese sind meist eng mit den Änderungsfunktionen der Entwicklungsumgebung verbunden und können daher direkt aus der IDE als bedeutungshafte Operationen geloggt werden.

Zwei weitere wünschenswerte Eigenschaften der Deltarepräsentation ist *self-containment* [8] und *compositionality* [8, 12]. Deltas sind *self-contained*, wenn diese nicht von fremden Quellen, wie beispielsweise dem Basismodell, abhängig sind. Die Kompositionsfähigkeit (*compositionality*) ist gegeben, sofern Differenzmodelle kombiniert, zusammengesetzt und gemeinsam ausgeführt werden können. Diese beiden Eigenschaften steigern die Weiter- und Wiederverwendbarkeit von Differenzen erheblich und eröffnen in Verbindung mit der zuvor beschriebenen Transformativität neue Anwendungsmöglichkeiten, wie beispielsweise das *Patching* von Modellen bei der modellgetriebenen Softwareentwicklung.

Gerade bei der Versionierung von Modellen ist es außerdem von Bedeutung, dass die Deltas in einer modellbasierten Form vorliegen und, nach dem Prinzip *everything is a model*, einem gemeinsamen Metamodell folgen [12]. Auch diese Eigenschaft steigert die weitere Verwendbarkeit. Beispielsweise wäre es dann möglich, über Model-To-Text-Transformation Änderungsberichte zu erzeugen. Es ist aber auch in diesem Zusammenhang wichtig, dass dieses Differenzmetamodell, genauso wie der Vergleich, metamodellunabhängig ist, sodass die Differenzen aller Sprachen des selben Metametamodells abbildbar sind.

3.4 Konflikterkennung

Beim Vergleich von zwei unabhängig voneinander durchgeführten Änderungen des selben Softwareartefakts sind gelegentlich auftretende Konflikte nicht zu verhindern. Ein Konflikt besteht dann, wenn widersprüchliche Änderungen vorgenommen wurden bzw. die Absichten der Personen, die die Änderungen vorgenommen haben, nicht eindeutig vereinbar sind [14]. Genauer gesagt tritt ein Konflikt auf, wenn die Änderungen beider Personen nicht lokal kommutativ sind [30]. Lokale Kommutativität liegt vor, wenn

die Reihenfolge, in der die Änderungen durchgeführt werden, das Endergebnis nicht beeinflussen.

3.4.1 Konfliktarten

In [14] und [26] wird zwischen zwei Hauptklassen an Konflikten unterschieden. Es gibt demnach Konflikte, die entweder durch gleichzeitige Manipulationen (*Concurrent Changes*) herbeigeführt werden oder jene, die entstehen, wenn die Änderungen Inkonsistenzen oder Integritätsverletzungen hervorrufen. Inkonsistenzen treten auf, wenn Änderungen nicht mehr auf den aktuellen Status des zu ändernden Objekts durchgeführt werden können. Das wäre beispielsweise der Fall, wenn das von einer Änderung betroffene Objekt zum Zeitpunkt der Zusammenführung nicht mehr existiert. Eine integritätsverletzende Änderung wäre hingegen – z.B. im Kontext eines Klassendiagramms – das Einfügen einer Vererbungsbeziehung, die im zusammengeführten Modell einen Vererbungskreis erzeugen würde. Diese Integritätskonflikte sind meist domänenspezifisch, da sie nur erkannt werden können, wenn ein Sprachbewusstsein vorherrscht.

Mens klassifiziert Konflikte anhand deren Level an Semantik [31] und unterteilt diese in *textuelle*, *strukturelle* und *semantische* Konflikte. Textuelle Konflikte haben keinerlei Sprachenbezug und basieren nur auf einer textuellen Repräsentation der Sprache. Meist wird eine *Unit Of Consistency* definiert, die angibt, ab welcher Granularität zwei Änderungen im Konflikt miteinander stehen. Dies ist üblicherweise Datei, Absatz oder Zeile. Ein textueller Konflikt tritt dementsprechend auf, wenn beispielsweise zwei EntwicklerInnen die selbe Zeile ändern. Strukturelle Konflikte wurden in der Domäne des Quellcodes als Konflikte definiert, die die statische Struktur (beispielsweise Klassen und deren Vererbungsbeziehungen, Prozeduren oder Subroutinen) des Programms betreffen. Für die Erkennung dieser Konfliktart muss also schon ein gewisses Sprachenverständnis vorliegen, um die Struktur der Artefakte zu verstehen. Die strukturelle Konflikterkennung ist daher zumindest domänenspezifisch (objektorientierte Programmierung, Auszeichnungssprachen, ...). Semantische Konflikte treten auf, wenn sich die Änderungen zweier EntwicklerInnen semantisch widersprechen oder deren eigentliche Absicht nicht eindeutig vereinbar ist. In [31] wurde als Beispiel für einen semantischen Konflikt die Verwendung einer Variable genannt, deren Deklaration von einem/r anderen EntwicklerIn entfernt wurde. Das Programm kann daher nicht mehr ausgeführt werden, da die verwendete Variable im zusammengeführten Zustand nicht mehr existiert. Die Erkennung derartiger Konflikte erfordert bereits detailliertes Sprachverständnis.

Altmanninger et al. unterteilen in [5] semantische Konflikte in drei Kategorien. Diese sind *Equivalent Concepts*, *Static Semantic Conflicts* und *Behavioral Semantic Conflicts*. Die erste Konfliktart, *Equivalent Concepts*, betrifft Fälle, in denen mit unterschiedlichen syntaktischen Konzepten die gleiche Semantik ausgedrückt wird. Der Unterschied ist also nur auf syntaktischer Ebene feststellbar. Bei der Ausführung des Modells und daher rein semantischer Betrachtung ist keine Differenz erkennbar. Statisch semantische Konflikte (*Static Semantic Conflicts*) treten auf, wenn beide Versionen nicht automatisch vereint werden können, da die Validierungsregeln der jeweiligen Modellierungssprache nicht gebrochen werden dürfen. Beispielsweise ist dies der Fall, wenn zwei EntwicklerInnen in einer geordneten Sequenz ein Element an der selben Stelle hinzufügen. Es kann nicht automatisch entschieden werden, wie die Reihenfolge der beiden Elemente in der zusammengeführten Version auszusehen hat. Beide Elemente können aber auch nicht an der selben Stelle eingefügt werden, da dies die Sprache bei geordneten

Sequenzen verbietet. Der dritte und letzte semantische Konflikt, *Behavioral Semantic Conflict*, entsteht bei Seiteneffekten von Änderungen, die sich rein syntaktisch gesehen nicht direkt tangieren. Dies ist beispielsweise der Fall, wenn eine Variable abhängig von zwei anderen Variablen ist (z.B. Summe) und beide Variablen unabhängig voneinander verändert werden.

In [20] ist außerdem die Rede von syntaktischen Konflikten. In der Quellcode-Domäne entstehen syntaktische Konflikte, wenn bei der textuellen Zusammenführung Syntaxfehler entstünden. Diese können nur durch syntaxbewusste Mergeverfahren (z.B. [10]) identifiziert und aufgelöst werden. Diese Art von Konflikten fallen, umgelegt auf die Softwaremodellierung, in die Hauptklasse der Integritätskonflikte. Schließlich sind die Bedingungen der Sprache, in der das Modell abgebildet ist, in diesem Fall, ähnlich wie im Beispiel der Quellcodedomäne, verletzt worden.

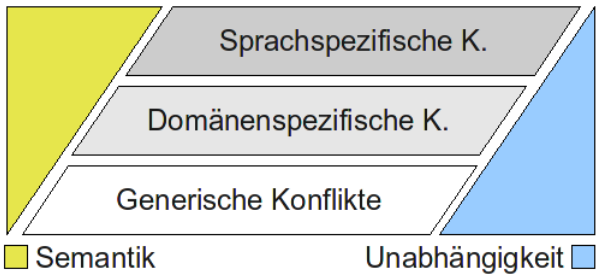

Abbildung 3.3: Konfliktarten basierend auf Mens [31].

In dieser Arbeit werden die Konflikte bei der Modellversionierung ähnlich der Kategorisierung von Mens unterschieden. Mens stellte diese Kategorisierung für Konflikte bei der Quellcodeversionierung auf, wodurch diese nicht direkt auf jene für die Modellversionierung anwendbar sind. Diese sind in Abbildung 3.3 zusammenfassend dargestellt. Wie bereits beschrieben, kategorisiert Mens in [31] die Konflikte anhand deren Level an Semantik. Auf der untersten Ebene befinden sich Konflikte, die auf textueller Basis auftreten. Bei der Quellcodeversionierung entspricht dies einem Konflikt auf Zeilenebene. Diese sind rein generischer Natur und haben keinen Bezug zur verwendeten Sprache. Bei der Modellversionierung können derartige Konflikte mit jenen verglichen werden, die bei der Betrachtung der Modelle in Form von sprachenunabhängigen Graphen erkannt werden. Die generische Konflikterkennung analysiert nur die Knoten und Kanten der Modelle und findet ausschließlich generische Konflikte, die bei gleichzeitigen Manipulationen entstehen (*Concurrent Changes*). Integritätskonflikte werden auf dieser Ebene nicht gefunden. Die darüberliegende Ebene betrifft strukturelle Konflikte. Im Kontext der Quellcodeversionierung können diese erkannt werden, wenn die Struktur einer Sprache vom Mergealgorithmus interpretiert und eventuell nicht vereinbare Änderungen in dieser Struktur gefunden werden. Bei der Modellversionierung entspricht dieses Level den domänenspezifischen Konflikterkennungsstrategien. Mit der domänenspezifischen Konflikterkennung können besondere Aspekte einer Domäne, beispielsweise Struktur- oder Verhaltensdiagramme, in die Konflikterkennung mit einbezogen werden. Fehlerhafte Struktureigenschaften, die bei der Zusammenführung entstehen würden, werden auf dieser Ebene entdeckt. Die höchste bzw. speziellste Ebene ist jene der semantischen Konflikte. Mens nennt als Beispiel im textuellen Kontext für Konflikte dieser Art einen Zugriff auf eine Variable, die weiter oben im Quellcode von einer anderen Person entfernt wurde. Diese Konflikte betreffen daher die Ausführung bzw. die Bedeutung des Codes. Das Mergeverfahren muss daher detailliertes, semantisches Verständnis auf-

weisen, um derartige Konflikte zu erkennen. Ein semantischer Konflikt in der Modellierungsdomäne tritt beispielsweise auf, wenn eine Person eine Klasse in ein Interface umwandelt und eine andere Person diese nun umgewandelte Klasse als Superklasse einer anderen definiert. Das Modell ist in diesem Fall nicht mehr konsistent, da eine Klasse keine Vererbungsbeziehung zu einem Interface aufweisen kann.

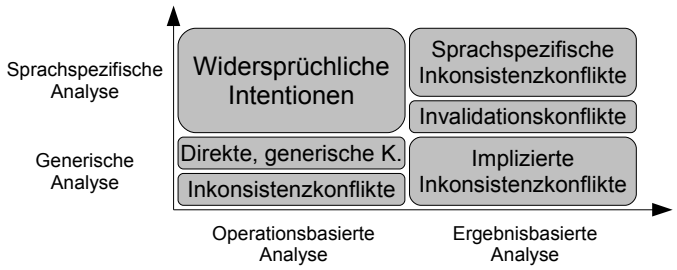

Abbildung 3.4: Konfliktarten nach Semantik und Erkennungsdimension.

Außerdem werden Konlfikte im weiteren Verlauf dieser Arbeit innerhalb einer Dimension unterschieden, die orthogonal zu der Dimension der Semantik bzw. der Sprachabhängigkeit steht. Innerhalb dieser Dimension wird unterschieden, auf welcher Basis diese Konflikte erkannt werden können. Diese sind in Abbildung 3.4 dargestellt. Konflikte können beispielsweise erkannt werden, wenn man ausschließlich die Operationsfolge oder die Differenzen zweier Änderungssessions ohne weitere Analyse des Ergebnisses und ohne Berücksichtigung der Semantik analysiert. Auf dieser Ebene können Inkonsistenzkonflikte nach [14] und [26] identifiziert werden. Inkonsistenzkonflikte treten bei Operationen auf, die aufgrund anderer Operationen nicht mehr ausführbar sind. Beispielsweise kann der Name einer Klasse nicht mehr verändert werden, wenn eine andere Operation diese Klasse entfernt hat (*DeleteUpdate*-Konflikt). Außerdem können auf dieser Ebene direkt auftretende, generische Konflikte erkannt werden. Dies wären beispielsweise Konflikte, bei denen auf beiden Seiten der selbe Knoten geändert wurde. Ein Sprachbewusstsein ist zur Erkennung dieser beiden Konflikttypen nicht nötig. Wird die Mergesituation weiterhin zwar ohne sprachspezifisches Wissen, jedoch auch unter Berücksichtigung des Ergebnisses analysiert, können auch implizierte Inkonsistenzkonlfikte erkannt werden. Konflikte dieser Art treten beispielsweise auf, wenn auf der einen Seite ein Element und damit implizit auch dessen *Containments* entfernt wurden und auf der anderen Seite eines dieser implizit entfernten *Containments* geändert wurde. Da keine explizite Entfernung des *Containments* im Deltadokument angeführt ist, kann kein Konflikt bei der ausschließlichen Betrachtung der Operationen erkannt werden. Dies hängt jedoch auch mit der Granularität der Differenzdarstellung zusammen, also ob implizierte Operationen ebenfalls explizit dargestellt werden. Dies wäre jedoch im Sinne der Minimalität nicht wünschenswert. Aus diesem Grund wird an dieser Stelle nicht davon ausgegangen. Bei der ausschließlichen Betrachtung der Operationsfolgen zweier Personen in Verbindung mit sprachspezifischem Wissen können Konflikte erkannt werden, die entstehen, wenn Operationen in Hinsicht auf deren Intention nicht vereinbar sind. Das ist im Kontext des UML Klassendiagramms beispielsweise der Fall, wenn eine Person ein *Interface* aus einer Klasse und eine andere Person eine abstrakte Klasse extrahiert (siehe Abbildung 9.3). Spezifisches Wissen ist zur Erkennung solcher Konflikte zwar nötig, die Analyse des Mergeresultats ist allerdings nicht erforderlich. Es

widersprechen sich lediglich die durchgeführten Operationen. Invalidationskonflikte, also Operationsfolgen, deren Kombination ein invalides Resultat zur Folge haben, können nicht durch die alleinige Analyse der Operationsfolgen erkannt werden. Hierfür ist die Interpretation der Status bzw. die des Resultats sowie sprachspezifisches Wissen notwendig. Das selbe trifft auch auf die sprachspezifischen Inkonsistenzkonflikte zu. Diese treten auf, wenn eine Änderungen bezüglich der sprachspezifischen Semantik zueinander inkonsistent sind. Folgendes Beispiel im Kontext des UML Klassendiagramms soll derartige Konflikte näher erläutern: Wenn eine Person eine neue Klasse hinzufügt und diese als *Containment* einer anderen Klasse definiert, indem er oder sie eine *Containment*-Referenz erstellt, dann ist dieses *Containment* nur durch die Semantik der Beziehung definiert und nicht durch ein physisches *Containment* auf generischer Elementebene. Wenn nun die andere Person die bestehende Klasse entfernt, die die neu erstellte Klasse durch die *Containment*-Referenz beinhaltet, steht die zuvor beinhaltete Klasse alleine da (siehe Abbildung 9.5). Das war aber vermutlich nicht die Intention der ModelliererInnen. Da das *Containment* einzig durch die Semantik der *Containment*-Beziehung definiert ist, kann dieser Konflikt nur durch sprachspezifisches Wissen und eine Analyse des Ergebnisses erkannt werden. Die in Abbildung 3.4 horizontal dargestellte Dimension ist eher für die Realisierung der Konflikterkennung bedeutend, als für die allgemeine Kategorisierung von Konflikten. Eine anschließende Konfliktresolution wird schließlich wenig mit der Information anfangen können, ob der Konflikt operations- oder ergebnisbasiert erkannt wurde. Dennoch hat diese Unterscheidung einen großen Einfluss auf die Architektur und Implementierung der sprachunabhängigen oder -spezifischen Konlifktdefinition und -erkennung.

3.4.2 Generische Konflikterkennung

Generische Konflikte können von sprachenunabhängigen Verfahren erkannt werden. Der Objektstatus wird als abstrakter Graph interpretiert, wobei die Elemente als Knoten und die Beziehungen als Kanten abgebildet werden. Ein abstraktes Element, sei es eine Zeile, ein Attribut einer Klasse oder eine Aktivität in einem Aktivitätsdiagramm, kann nach dem Vergleich mit der Ursprungsversion vier verschiedene Status einnehmen. Diese sind `Added`, `Deleted`, `Changed` oder `NotChanged`. Werden Elemente ihren in einer anderen Version befindlichen Gegenstücken gegenübergestellt, ergeben sich daher 4^2, also 16 mögliche Statuskombinationen [16].

Diese 16 Fälle sind in Tabelle 3.1 dargestellt. Davon sind 5 der möglichen Kombinationen in der Realität nicht existent (Fall 6, 7, 8, 10 und 14), da beispielsweise im Fall 14 das Element von Person 1 ($V_{0'}$) nicht geändert werden kann, da dieses erst von Person 2 ($V_{0''}$) hinzugefügt wurde. In Fall 1 wurde keine Manipulation durchgeführt. Daher ist auch dieser Fall nicht betrachtenswert. In 7 weiteren Fällen (2, 3, 4, 5, 9, 11 und 13) entsteht kein Konflikt, da beide Status einfach vereinbar sind. Der jeweilige, dominante Status ist in der Spalte ganz links angeführt. In Fall 12 und 15 handelt es sich jeweils um ein `Deleted` und ein `Changed`. Die dominante Strategie wäre in beiden Fällen, die Löschung durchzuführen und die Änderungen der anderen Seite zu verwerfen. Es können jedoch nicht die Absichten beider EntwicklerInnen vereint werden, da der oder die EntwicklerIn, der oder die die Änderung durchgeführt hat, ein weiteres Bestehen des Elements beabsichtigt hat. Andernfalls hätte er oder sie die Änderung nicht durchgeführt. Man kann in diesem Fall also von einem potentiellen Konflikt sprechen. Im Fall 16 handelt es sich um einen harten Konflikt, da hier die Änderungen beider EntwicklerInnen aus generischer Sicht in keinem Fall vereint werden können.

Nr.	V0'	V0"	Folge/Bemerkung
1	`NotChanged`	`NotChanged`	Keine Aktion
2	`NotChanged`	`Added`	→ `Added` aus V_0''
3	`NotChanged`	`Deleted`	→ `Deleted` aus V_0''
4	`NotChanged`	`Changed`	→ `Changed` aus V_0''
5	`Added`	`NotChanged`	→ `Added` aus V_0'
6	`Added`	`Added`	Nicht möglich
7	`Added`	`Deleted`	Nicht möglich
8	`Added`	`Changed`	Nicht möglich
9	`Deleted`	`NotChanged`	→ `Deleted` aus V_0'
10	`Deleted`	`Added`	Nicht möglich
11	`Deleted`	`Deleted`	→ `Deleted`
12	`Deleted`	`Changed`	Potentieller Konflikt
13	`Changed`	`NotChanged`	→ `Changed` aus V_0'
14	`Changed`	`Added`	Nicht möglich
15	`Changed`	`Deleted`	Potentieller Konflikt
16	`Changed`	`Changed`	Harter Konflikt

Tabelle 3.1: Kombinationen generischer Status eines Elements.

3.4.3 Sprachspezifische Konflikterkennung

Je sprachspezezifischer die Konflikterkennung aufgebaut ist, desto exakter ist diese. Der Preis dafür ist die Abhängigkeit der Konflikterkennung von dieser Sprache und die damit verbundene, geringe Wiederverwendbarkeit. Es ist daher bei der Entwicklung eines sprachspezezifischen Systems darauf zu achten, die sprachenabhängige Schicht so dünn wie möglich zu halten. Das Ausmaß des Sprachenbewusstseins (generisch bis spezifisch) eines Merges ist orthogonal zu seiner Basis (statusbasiert, operationsbasiert, ...) und kann nach Wunsch kombiniert werden.

Abbildung 3.5 zeigt eine Situation im Kontext des UML-Klassendiagramms, die mit generischer Konflikterkennung und anschließender Zusammenführung zu einem inkonsistenten Modell führen würde. Auf beiden Seiten (V_0' und V_0'') wurde das Ursprungsmodell V_0 um eine Vererbungsbeziehung erweitert. Auf der linken Seite (V_0') von A zu C und auf der rechten (V_0'') von C zu B. Eine generische Konflikterkennung kann keine Konflikte entdecken. Es wurden keine Elemente direkt von beiden Personen geändert oder entfernt. Es tritt daher nach der Tabelle 3.1 keine Kombination auf, die zu einem Konflikt führt. Eine Zusammenführung würde allerdings einen Vererbungskreis verursachen, der das Modell in einen inkonsistenten Zustand versetzt.

Zur Erkennung derartiger Konflikte ist Sprachbewusstsein notwendig. Im obigen Beispiel würde eine sprachspezifische Konflikterkennung berücksichtigen, dass Vererbungskreise in UML Klassendiagrammen nicht erlaubt sind und kann daher erkennen, dass in diesem Beispiel nach der Zusammenführung der Modelle eine Inkonsistenz vorliegen würde.

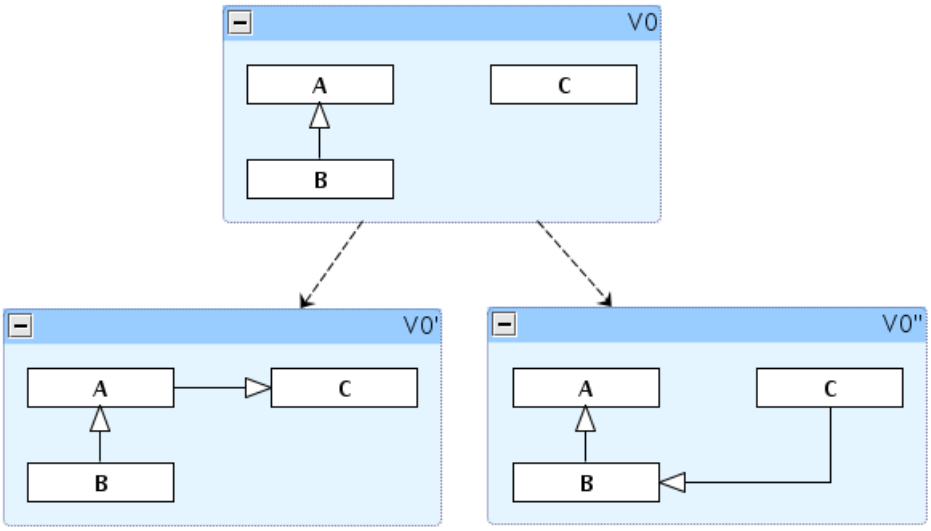

Abbildung 3.5: Sprachspezifische Konflikterkennung: Vererbungskreis.

3.4.4 Konflikterkennung und der 2-Wegsvergleich

Der 2-Wegsvergleich ist immer statusbasiert. Bei änderungsbasierten Vorgehensweisen kann die ursprüngliche Version hergeleitet werden, indem die Transformationen invertiert und auf die beiden Status ausgeführt werden. Sobald der Ursprung jedoch bekannt ist, handelt es sich bereits um einen 3-Wegsvergleich. Beim statusbasierten 2-Wegsvergleich werden hingegen nur zwei verschiedene Versionen verglichen. Die Deltarepräsentation ist zwingenderweise symmetrisch, da gerichtete Transformationen zwischen den beiden unabhängigen Objekten, deren Ursprung unbekannt ist, keinen Sinn ergäbe. Eine echte Konflikterkennung kann in diesem Fall nicht durchgeführt werden. Der Merge kann lediglich eine Vereinigung aller Elemente, nicht jedoch eine Vereinigung der Absichten der beiden EntwicklerInnen, durchführen, da die Absichten nicht ausreichend erkennbar sind, sodass letztlich bei jedem aufzulösenden Konflikt eine Benutzerentscheidung nötig wird.

In Abbildung 3.6 ist ein einfaches Beispiel für die folgende Gegenüberstellung der Konflikterkennungstechniken (statusbasiert, 2-Weg, 3-Weg, ...) in einem UML Klassendiagramm dargestellt. *V0* ist die Ausgangssituation mit einer Klasse *Person* und zwei Attributen vom Typ *String*. Zwei EntwicklerInnen verändern dieses Modell nun auf unterschiedliche Art. In *V0'* wurde das Attribut *name* entfernt und statt dessen zwei neue Attribute *firstName* und *lastName* angelegt. Außerdem wurde das Attribut *birthDate* in *dateOfBirth* umbenannt. *V0"* beinhaltet keine Änderung des Attributs *name*. Allerdings wurde der Typ des Attributs *birthDate* auf *Date* geändert.

Bei dem 2-Wegsvergleich, die nur auf beide veränderten Status zurückgreift, kann, wie bereits angesprochen, keine echte Konflikterkennung durchgeführt werden. Die Version *V0* wird nicht berücksichtigt. Es wird daher nicht festgestellt, dass in *V0'* das Attribut *name* entfernt wurde. Der *Deleted—NotChanged*-Fall bleibt daher unerkannt. Auch der *Changed—Changed*-Konflikt der Attribute *dateOfBirth* und *birthDate* kann nicht direkt entdeckt werden.

Erweitert man jedoch den 2-Wegsvergleich mit einem UUID-basierten Match, kann

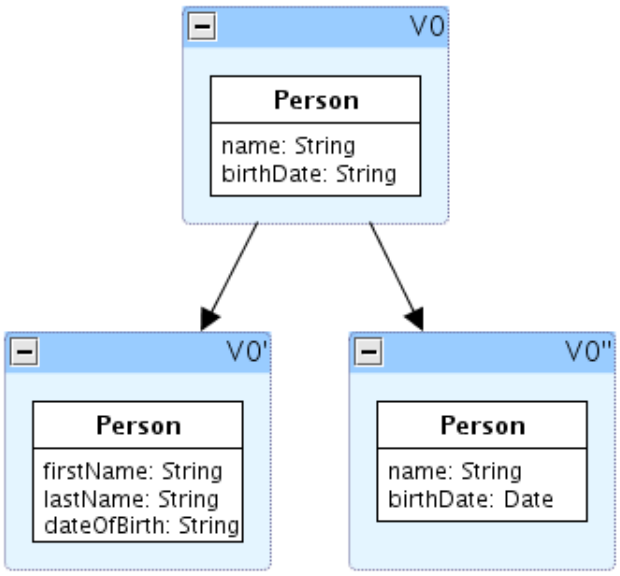

Abbildung 3.6: Beispiel der Konflikterkennung: Ausgangssituation.

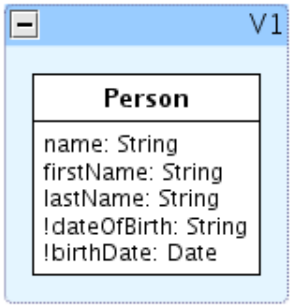

Abbildung 3.7: Ergebnis der Konflikterkennung: Statusbasiert, 2-Wegsvergleich.

festgestellt werden, dass die UUIDs der Attribute *dateOfBirth* und *birthDate* die selben sind und dass damit ein *Changed—Changed*-Konflikt vorliegt. Dies ist mit dem vorangestellten *!* in Abbildung 3.7 notiert. Ohne UUID-Vergleich kann nur aufgrund der Namensgleichheit vermutet werden, dass die Klasse *Person* in *V0'* und *V0"* den selben Ursprung hat. Die Attribute *birthDate* und *dateOfBirth* können einander nicht mit Sicherheit zugeordnet werden, da sie einen unterschiedlichen Typ und keinen besonders ähnlichen Namen aufweisen. Die Levenshtein-Differenz [28], die die minimale Anzahl an Buchstabenoperationen (Löschen, Ändern, ...) angibt, ist schließlich mit *10* relativ hoch. Abbildung 3.7 zeigt die Zusammenführung, die durch den statusbasierten 2-Wegsvergleich ermittelt werden würde. Wie bereits angemerkt, ist der *Changed—Changed*-Konflikt theoretisch erkennbar. Allerdings nur, wenn eindeutige Eigenschaften zur Elementwiedererkennung verfügbar sind.

3.4.5 Konflikterkennung und der 3-Wegsvergleich

Der statusbasierte 3-Wegsvergleich kann die Anzahl der Benutzerentscheidungen drastisch reduzieren, da die Ursprungsversion für viele Fälle die nötigen Informationen enthält, um die durchgeführte Änderung zu identifizieren (z.B. Löschen eines Elements, etc.).

Wie bereits weiter oben angemerkt, sind änderungsbasierte Ansätze stets 3-Wegsvergleiche, da durch die gerichteten Änderungen implizit die Ursprungsversion angegeben ist und durch deren Umkehr die Ursprungsversion auch explizit hergeleitet werden kann. Beim 3-Wegsvergleich werden die Unterschiede beider Versionen typischerweise in gerichtete Deltas überführt. Das selbe trifft auch auf änderungsbasierte Systeme zu [31]. Bei diesen ist jedoch keine Umwandlung nötig, da die Änderungen bereits vorliegen. Ein weiterer Vorteil der änderungsbasierten Vorgehensweisen gegenüber den statusbasierten ist, dass der Zusammenhang zwischen den Elementen der Ursprungsversion und der veränderten Versionen immer explizit vorliegt und nicht erst durch Heuristiken errechnet werden muss. Eine explizite Transformation beim änderungsbasierten Ansatz muss ein Element der Ursprungsversion über ein identizierendes Merkmal referenzieren. Auch wenn dies keine UUID in dem Sinn ist, kann dies z.B. auch über eine eindeutige Adresse oder die hierarchische Position geschehen. Z.B. kann in Abbildung 3.6 das veränderte Attribut *name* mit `Diagramm1/Person/@name` addressiert sein. Über die Ursprungsversion kann demnach auch ein Äquivalenzmapping zwischen den Elementen in den beiden veränderten Versionen hergestellt werden, sodass beispielsweise ein *Changed—Changed*-Konflikt immer erkannt werden kann. Beim statusbasierten 3-Wegsvergleich ist dieses Mapping vom Ursprungselement zum veränderten Element nicht immer vorhanden und muss entweder über Heuristiken geschätzt oder über UUIDs realisiert werden.

Beim 3-Wegsvergleich im Allgemeinen kommt der Merge in vielen Fällen ohne Benutzerinteraktion aus. Wenn eine Änderung (Hinzufügen, Löschung, ...) nur auf einer Seite vorgenommen wurde, kann diese problemlos in die Endversion übernommen werden. Ein Konflikt entsteht nur dann, wenn eine Änderung auf beiden Seiten das selbe Element einbezieht. Formal ausgedrückt, tritt ein Konflikt auf, wenn eine Transformation nicht lokal kommutativ ist. Eine Transformation ist dann lokal kommutativ, wenn die Reihenfolge der Transformationen das Endergebnis nicht beeinflusst. In der Gleichung 3.1 ist diese Kommutativität formal notiert, wobei *T1* und *T2* voneinander unabhängige Transformationen sind und $\approx$ als Gleichheitsrelation definiert ist.

$$T_1(T_2(s)) \approx T_2(T_1(s)) \tag{3.1}$$

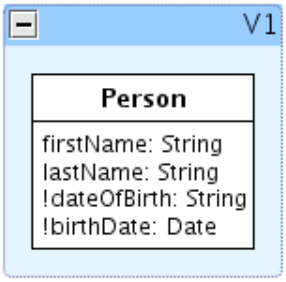

Abbildung 3.8: Ergebnis der Konflikterkennung: 3-Wegsvergleich.

Abbildung 3.8 zeigt das Ergebnis auf Basis eines 3-Wegsvergleichs ausgehend von der selben Ausgangssituation der letzten Sektion (Abbildung 3.6). Das Attribut *name* wird diesmal allerdings nicht übernommen, da es in Version *V0'* entfernt und in *V0"* nicht verändert wurde. Der *Deleted—NotChanged*-Fall kann also erkannt und richtig umgesetzt werden. Der *Changed—Changed*-Konflikt im Fall der Attribute *birthDate* und *dateOfBirth* wird, wie bereits erwähnt, von änderungsbasierten Vorgehensweisen immer erkannt. Beim statusbasierten 3-Wegsvergleich verhält es sich in diesem Fall ähnlich wie beim 2-Wegsvergleich. Dieser kann nur richtig entdeckt werden, wenn identifizierende Merkmale die Wiedererkennung der Attribute ermöglichen. Um diesen zu entdecken, muss erkannt werden, dass das Attribut *birthDate* der Version *V0* der Vorgänger des Attributs *dateOfBirth* ist. Beim statusbasierten 3-Wegsvergleich ist das ohne der Verwendung von eindeutigen Attributen nicht sichergestellt. Wenn diese Erkennung fehlschlägt, werden *Changed—Changed*-Konflikte fälschlich als *Deleted—Changed*-Konflikt und *Added*-Änderung erkannt.

3.4.6 Konflikterkennung und der operationsbasierte Vergleich

Beim operationsbasierten Vorgehen werden die Transformationen explizit, zumeist als sinnbehaftete und damit domänenspezifischen Operationen, abgebildet. Dadurch ist die Konflikterkennung auf Basis der feingranularen Änderungen, deren Bedeutung verstanden werden können, weitaus exakter. Die Genauigkeit und Qualität hängt stark von der Definition der Operationen ab. Je genauer und aussagekräftig diese sind, desto besser ist das Ergebnis – jedoch auf Kosten der Domänenunabhängigkeit. In unserem Beispiel von Abbildung 3.6 könnten diese Operationen wie in Listing 3.1 aussehen.

Die Konflikterkennung kann nun überprüfen, ob diese Transformationen lokal kommutativ sind. In diesem Fall führt die Ausführung der Operationen unabhängig von der Reihenfolge immer zum selben Ergebnis. Es besteht daher Kommutativität und damit kein Konflikt. Das Ergebnis aller Transformationen ist in Abbildung 3.9 dargestellt.

Es können also mit steigender Aussagekraft der Operationen viele *true-negative*-Konflikte, also Konfliktmeldungen wo keine sind, verhindert werden. Weiters werden aber auch *false-positive*-Konflikte, also Konflikte die nicht gemeldet werden, verringert. Ein Beispiel hierfür sind erkannte Refactorings. Eine Person könnte beispielsweise eine

Listing 3.1: Darstellung durchgeführter Operationen.

```
V0 —> V0'
  removeAttribute(Person/@name)
  addAttribute(Person/@firstName, String)
  addAttribute(Person/@lastName, String)
  changeAttributeName(Person/@birthDate, dateOfBirth)

V0 —> V0''
  changeAttributeType(Person/@birthDate, Date)
```

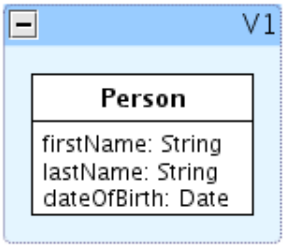

Abbildung 3.9: Ergebnis der Konflikterkennung: Operationsbasiert.

Methode umbenennen und eine andere Person einen Aufruf dieser Methode an einer anderen Stelle (mit altem Namen) einfügen. Eine Konfliktmeldung könnte verhindert werden, indem alle hinzugekommenen Methodenaufrufe überprüft und an die Methodenumbenennung angepasst werden.

3.4.7 Konflikterkennung und die vorherige Transformation der Objektstatus

In Kapitel 3.2 wurden Transformationen von Objektstatus vor dem Vergleich der beiden Versionen besprochen.

Dort wurden auch die beiden entgegengesetzten Gründe für eine solche Transformation erläutert. Der Grund der Verallgemeinerung auf eine gemeinsame abstrakte Sprache hat keinen direkten Einfluss auf die Konflikterkennung. Die Betrachtungsweise, mit der die Objektstatus analysiert werden, wird generalisiert und damit generischer. Dies führt auch zu einer generischen und damit allgemeineren Konflikterkennung, sodass auch nur generische Konflikte erkannt werden können [37]. Im Gegensatz dazu ist der andere Grund einer vorangestellten Transformation die Spezialisierung, also die Fokussierung auf sprachspezezifische Aspekte. Dies kann einen sehr großen Einfluss auf die Konflikterkennung haben und diese in hohem Maße verbessern.

In Bezug auf die Modellversionierung ist als Beispiel für die vorherige Transformation der Objektstatus das Modellversionierungssystem SMoVer [5] zu nennen. SMoVer wird in der Analyse bestehender Ansätze (Kapitel 4) näher beschrieben. Bei diesem Ansatz werden Modelle, die einem Metamodell folgen, in ein anderes Metamodell, eine sogenannte *Semantic View*, übersetzt. Diese ist speziell zur Berücksichtigung besonderer

Aspekte des ursprünglichen Metamodells aufgesetzt. Es werden nach der Übersetzung der Modellversionen diese übersetzten Modelle und nicht direkt die zu vergleichenden Modelle verglichen. So kann beispielsweise bei WSBPEL[3]-Prozessmodellen jede Modellversion in einen Abhängigkeitsgraphen der verwendeten Variablen übersetzt und bei dessen Vergleich konkurrierende Änderungen festgestellt werden, die eine nicht direkt geänderte Variable beeinflussen (z.B. die Summe anderer veränderer Variablen). Es wird also ein semantischer Konflikt erkannt, der bei herkömmlichen Vergleichen unbemerkt geblieben wäre.

3.4.8 Konflikterkennung und Refactorings

Refactorings sind programmtechnische Umgestaltungen bzw. Umstrukturierungen, die die Funktionalität nach außen nicht verändern, sondern eine Verbesserung der Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit zum Ziel haben [21]. Refactorings führen meistens zu sehr vielen Änderungen an vielen unterschiedlichen Stellen im Quellcode der Software. Viele integrierte Entwicklungsumgebungen unterstützen daher gängige Refactorings und machen diese zu einer teilautomatischen Operation.

Im Zusammenhang mit der Versionierung stellen Refactorings aufgrund der großen Änderungszahlen an unterschiedlichen Stellen eine große Herausforderung dar. Die Wahrscheinlichkeit, dass eine Änderung mit einer parallelen Änderung kollidiert, ist durch die starke Verteilung der Refactoring-Änderungen sehr hoch. Trotz der vielen kleinen Änderungen steht hinter einem Refactoring nur *eine* semantische Operation mit *einer* klaren Absicht. Aus diesem Grund ist es äußerst wünschenswert, dass ein Versionierungssystem Refactorings als solche erkennt und berücksichtigt. Derartige Systeme sind häufig operationsbasierte Systeme, indem sie Refactorings direkt bei der Durchführung in der Entwicklungsumgebung mitschreiben. Es gibt aber auch Ansätze, Refactorings aus einer statusbasierten oder operationsbasierten Basis ohne Refactoring-Auszeichnung zu erkennen (z.B. RefactoringCrawler [17]).

In Abbildung 3.10 ist ein Beispiel dargestellt, in dem das Erkennen von Refactorings eine Konfliktmeldung verhindern kann. Ausgangssituation (*V0*) ist eine einfache Klasse mit einer Methode *draw()*. Eine Person extrahiert die Superklasse (*Extract Superclass* [21]) *Drawable*, die die Methode ab jetzt implementiert. In der Klasse *Circle* befindet sich also keine Methode mehr. Eine zweite Person benennt nun in der Klasse *Circle* die Methode *draw()* um. Alle bisher vorgestellten Merge-Methoden würden nun einen Konflikt melden. Aus generischer Sicht wurde das gleiche Element verändert bzw. entfernt (Methode *draw()*). Aus sprachspezezifischer Sicht wurde nach der Entfernung einer Methode eine Änderung des Methodennamens durchgeführt. Ist allerdings bekannt, dass es sich hier um ein reines Refactoring handelt, kann auf die Absicht des Entwicklers bzw. der Entwicklerin, nämlich die Superklasse zu extrahieren, geschlossen werden. Üblicherweise können, wenn normale Änderungen vorgezogen und die Refactorings erst ganz zum Schluss durchgeführt werden, viele Konflikte beseitigt werden. Bei dem zuvor genannten Beispiel wäre das Ergebnis, wenn das Refactorings erst ganz zum Schluss durchgeführt werden würde, das selbe wie in *V0'*, mit dem Unterschied dass die Methode in *Drawable drawYourself()* hieße.

[3]WSBPEL - Web Service Business Process Execution Language.

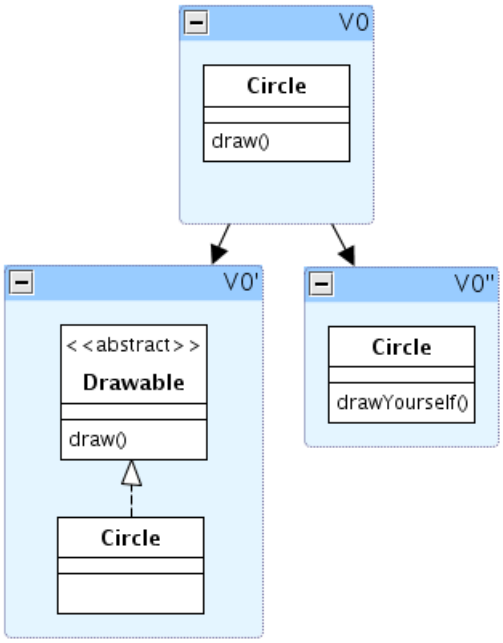

Abbildung 3.10: Beispiel der Konflikterkennung mit Refactorings.

4 Analyse bestehender Arbeiten

Der direkte Vergleich bestehender Arbeiten aus dem Modellversionierungsbereich ist kein leichtes Unterfangen. Die Herangehensweisen unterscheiden sich ebenso stark wie deren Fokus. Diese Analyse erfolgt daher in Form einer Kategorisierung der untersuchten Arbeiten nach *Vergleich*, *Deltarepräsentation* und *Konflikterkennung*. Diese decken sich mit den bereits erläuterten Konzepten des Kapitels 3. Der Vergleich umfasst die folgenden Arbeiten.

- Alanen und Porres [4]
- Blanc et al. [6]
- Cicchetti et al. [11, 12]
- DSMDiff [29]
- EMF Compare [8]
- Kögel [26]
- Mens et al. [32]
- MolhadoRef [18]
- Ohst et al. [33]
- SMoVer [5]
- Störrle [37]
- X-Diff [38]

Alanen und Porres haben in [4] eine der ersten Arbeiten im Modellversionierungsbereich verfasst. Sie stellen in dieser Arbeit drei metamodellunabhängige Algorithmen vor, die für MOF-basierte Modelle auch implementiert wurden. Bei diesen Algorithmen handelt es sich um die Errechnung von Differenzen zwischen zwei Modellen und um die Zusammenführung sowie die Vereinigung von Modellen.

Blanc et al. [6] präsentieren in ihrer Arbeit eine operationsbasierte Erkennung von Modellinkonsistenzen, wobei sowohl strukturelle als auch methodologische Inkonsistenzen durch logische Regeln abgebildet und anschließend in Operationsfolgen aufgedeckt werden können. Es werden hierfür Ecore basierte Modelle und Rational Software Architekt Modelle in logische Fakten übersetzt und die zuvor angesprochenen Regeln mittels Prolog geprüft.

Cicchetti et al. [11, 12] stellen einen metamodellunabhängigen Weg vor, Differenzen darzustellen. Sie übersetzen Metamodelle durch eine allgemeine Transformation in ein für das jeweilige Metamodell spezifisches Differenzenmetamodell. Anschließend können Modelle, als Instanzen des jeweiligen Metamodells, verglichen und deren Unterschiede als Instanz des erzeugten, metamodellspezifischen Differenzmetamodell abgebildet werden. Es liegt eine Implementierung unter der Verwendung von ATL [25] vor.

DSMDiff [29] ist ein Tool, um Unterschiede in domänenspezifischen Modellen zu ermitteln. Es übersetzt domänenspezifische Modelle in ein allgemeines, graphenbasiertes Modell, um diese anschließend miteinander zu vergleichen. Dieser Ansatz ist metamodellunabhängig, sofern eine Übersetzung in das allgemeine Metamodell für ein spezielles Domänenmodell vorliegt.

EMF Compare [8] ist ein Eclipse [2] Plug-In, das Ecore-basierte Modelle matchen, vergleichen und zusammenführen kann. Der Match wird als sprachunabhängiges, heuristisches Verfahren durchgeführt. Die Differenzen werden konform zu einem eigenen Differenzen-Metamodell abgebildet und können dem User in graphischer Form als symmetrische und gerichtete Deltas gleichermaßen angezeigt werden.

Kögel [26] skizziert ein Konfigurationsmanagement für Modelle. Es wird im Zuge dessen ein Differenz- und ein Versionsmodell eingeführt, mit dem die Evolution und die einzelnen Versionsunterschiede repräsentiert werden können. Die Änderungen selbst werden durch die Erweiterung der Editoren, also operationsbasiert, ermittelt und können in weiterer Folge manuell vom User zu Änderungspaketen zusammengefasst werden.

Mens et al. [32] stellen in ihrer Arbeit ein Verfahren vor, mit dem strukturelle Konflikte zwischen Refactorings mithilfe von Graphentransformationsmethoden erkannt werden können. Refactorings werden zu diesem Zweck als Graphtransformation abgebildet. Mit diesen Refactoring-Definitionen kann ermittelt werden, welche Refactorings, angewendet auf das selbe Modell, miteinander in Konflikt stehen.

MolhadoRef [18] ist eine Konfigurationsmanagement-Lösung für objekt-orientierte Programmiersprachen. Diese Sprachen werden in einen speziellen Graphen übersetzt, der als Grundlage für weitere Aktionen, wie beispielsweise die Konflikterkennung, dient. Refactorings werden direkt vom Editor, in diesem Fall Eclipse [2], mitgeschrieben und in den anschließenden Analysen mit eingeschlossen.

In der Arbeit von Ohst et al. [33] wird der Fokus auf die Visualisierung der Differenzen zweier Modelle gelegt. Es wird eine Coloring Technik vorgestellt, die die spezifischen Teile beider Versionen sowie die gemeinsamen Elemente mittels einer visuellen Hervorhebung abbildet.

In SMoVer [5] werden Modelle, die einem Metamodell folgen, in ein anderes Metamodell, eine sogenannte *Semantic View*, übersetzt. Diese ist speziell zur Berücksichtigung besonderer Aspekte des eigentlichen Metamodells aufgesetzt. Es werden nach der Übersetzung der Modellversionen diese übersetzten Modelle und nicht direkt die zu vergleichenden Modelle verglichen. Dadurch wird eine Erkennung besonderer, sprachspezifischer Konflikte erzielt.

Störrle [37] stellt einen formalen Ansatz zur metamodellunabhängigen Modellversionierung vor. Es werden die Modelle in einen einfachen, mathematischen Formalismus übersetzt, um diese unter der Verwendung von logischen Regeln zu analysieren. Dieser Ansatz unterstützt die Differenzermittlung in Form von Operationen sowie die Konflikterkennung und -behebung.

X-Diff [38] ist nicht direkt eine Arbeit aus dem Bereich der Modellversionierung. X-Diff zielt auf den Vergleich von XML-Dokumenten und nicht auf jenen von Modellen ab. Dennoch wurde es in diese Analyse mit eingeschlossen, da einerseits Modelle als XML (z.B. mit XMI) serialisiert werden können und andererseits XML-Dokumente im Gegensatz zu herkömmlichen Textdateien eine Baumform haben. Ein Baum ist eine

Spezialform eines Graphen und Modelle haben eine Graphenform. Daher ist es durchaus sinnvoll, X-Diff bei der Analyse von Modellvergleichsmethoden zu berücksichtigen.

4.1 Vergleich

In diesem Unterkapitel werden die jeweils verwendeten Vergleichstechnologien der analysierten Arbeiten einander gegenübergestellt. In Kapitel 3.2 wurden bereits Eigenschaften und Techniken des Vergleichs besprochen. Die selben Eigenschaften dienen teils auch als Kategorien für diese Analyse und sind in der folgenden Liste angeführt.

Match: Wie ist das Auffinden äquivalenter Elemente realisiert? Werden UUIDs verwendet oder verlässt man sich auf Heuristiken?

Basis: Welche Information dient als Basis des Vergleichs? Ist dieser status- oder operationsbasiert?

Transformation: Findet vor dem Vergleich eine Transformation der Objektstati statt?

Metamodellabhängigkeit: Hängt der Vergleich von der Sprache, in der die Modelle formalisiert sind, ab?

Arbeit	Match	Basis	Transf.	Metamodell
Alanen [4]	UUID	3-Weg, status	nein	unabhängig
Blanc et al. [6]	N/A	operation	ja	unabhängig
Cicchetti et al. [12]	N/A	N/A	nein	unabhängig
DSMDiff [29]	heuristisch	2-Weg, status	nein	unabhängig
EMF Compare [8]	heuristisch	3-Weg, status	nein	unabhängig
Kögel [26]	N/A	operation	ja	unabhängig
Mens et al. [32]	N/A	3-Weg, status	ja	unabhängig
MolhadoRef [18]	N/A	operation/3-Weg	ja	abhängig
Ohst et al. [33]	UUID	3-Weg, status	nein	abhängig
SMoVer [5]	UUID	3-Weg, status	ja	unabhängig
Störrle [37]	heuristisch	2-Weg, status	ja	unabhängig
X-Diff [38]	heuristisch	2-Weg, status	nein	unabhängig

Tabelle 4.1: Kategorisierung bestehender Arbeiten: *Vergleich*

.

In Tabelle 4.1 sind die zuvor beschriebenen Eigenschaften der untersuchten Systeme aufgelistet. In der ersten Spalte (*Match*) ist notiert, ob in den jeweiligen Arbeiten gleiche Elemente mittels UUIDs oder durch heuristische Methoden identifiziert werden. Bei Blanc et al., Kögel und MolhadoRef ist diese Kategorisierung nicht anwendbar (*N/A*), da es sich bei diesen drei Arbeiten um operationsbasierte Methoden handelt. Bei diesen ist ein Match nicht explizit notwendig, da die getrackten Operationen die betreffenden Elemente in irgendeiner Form identifizieren oder referenzieren müssen. Damit besteht eine Beziehung zwischen den Elementen der veränderten Version zu den Elementen in der Ursprungsversion. Folgedessen kann auch über die Ursprungsversion zwischen den Elementen beider unabhängig voneinander veränderten Versionen eine Relation hergestellt werden. Alle nicht veränderten Elemente müssen exakt gleich sein, da sonst

eine Operation stattgefunden hätte. Diese sind daher leicht über Namen und Typ oder Position in Beziehung zu setzen. Bei Mens et al. und Cicchetti et al. ist ebenfalls ein *N/A* notiert, da diese Arbeiten den Vergleich und damit auch den Match nicht explizit behandeln. Im Fall von Cicchetti et al. gilt dies auch bei der Basis (Spalte 2). Ansonsten wurde in der zweiten Spalte *Basis* vermerkt, ob ein statusbasierter Ansatz (2-Weg oder 3-Weg) verwendet wurde oder ob eine operationsbasierte Methode zur Anwendung kommt. MolhadoRef verwendet sowohl einen statusbasierten 3-Wegsvergleich für die Ermittlung der Änderungen von Java-Code als auch einen operationsbasierten Ansatz für das Tracking von Refactoring-Operationen. Diese werden direkt bei der Durchführung der Operationen in Eclipse [2] mitgeschrieben.

In der dritten Spalte (*Transf.*) ist vermerkt, ob eine vorherige Transformation der Objektstati vorgenommen wird. Es ist beispielsweise ersichtlich, dass kein operationsbasierter Ansatz eine vorherige Transformation verwendet. Diese dritte Spalte hängt aber in erster Linie mit der vierten Spalte *Metamodell* zusammen, die angibt, ob die in der Arbeit präsentierten Methoden abhängig von einem speziellen Metamodell sind. Wie weiter oben bereits besprochen (Kapitel 3.2), gibt es für eine vorherige Transformation die beiden Gründe *Spezialisierung* und *Generalisierung*. In den Arbeiten von Blanc et al., Kögel, Mens et al., Störrle und in MolhadoRef wurde eine vorherige Transformation genutzt (Spalte 3 = *ja*), um den Objektstatus in ein allgemeines Metamodell zu übersetzen und so die Methoden für mehrere Modellierungssprachen wiederverwendbar zu machen (Spalte 4 = *unabhängig*). Blanc et al. transformieren Modelle in Prolog-Fakten. Ähnlich ist es bei dem Ansatz von Störrle, in dem die Modelle in eine formale, logikbasierte Darstellungsform gebracht werden. Kögel konvertiert die Modelle folgend einem Metametamodell namens RUSE [39] und Mens et al. übersetzen die Modelle in einen getypten Graphen. MolhadoRef übersetzt Java-Code in ein eigenes, graphenbasiertes Modell. Refactorings werden jedoch als Operationen direkt geloggt und nicht transformiert. SMoVer verwendet ebenfalls eine Transformation und ist unabhängig von einem konkreten Metamodell. Jedoch findet keine Transformation in ein allgemeines Metametamodell statt, sondern in spezielle Semantic Views, durch deren Vergleich konkrete Aspekte zweier Modellversionen überprüft werden können. In Kapitel 3.4 wurde dies bereits kurz erläutert.

SMoVer realisiert also die Metamodellunabhängigkeit nicht durch eine Generalisierungstransformation, sondern durch die direkte Verwendung des Metametamodells, in der das Metamodell des zu vergleichenden Modells definiert ist. Es ist daher möglich das Modell über die Schnittstellen des Metametamodells zu verarbeiten, ohne das Metamodell direkt zu kennen. Im Fall von SMoVer handelt es sich um Ecore (EMF) [9] als Metametasprache. Die gleiche Methodik machen sich auch Alanen und Porres mit MOF [34], Cicchetti et al. mit KM3 [24], in DSMDiff mit GME [27] und in EMF Compare mit Ecore [9] zu nutze. Diese Herangehensweise kann mit jener des UNIX *diff* Befehls [23] verglichen werden. Dieser operiert nur auf Textebene und nicht auf der Sprache (vgl. Metamodell), die verwendet wird (z.B. Englisch, Deutsch, Java, ...), und schon gar nicht mit der Semantik, die mit dieser Sprache ausgedrückt wird. Text kann in diesem Zusammenhang mit einer Metametasprache verglichen werden. Das selbe gilt im Falle von X-Diff für XML.

Bei dieser Analyse von Vergleichstechniken sind im Bezug auf den *Match* vor allem jene interessant, die auf Heuristiken basieren. UUID-basiertes Matching ist relativ einfach durch ein Traversieren des Graphen möglich. Auch bei operationsbasierten Ansätzen ist der Verlgeich wenig herausfordernd, da alle Elemente, die nicht ohnehin direkt von

einer Operation referenziert sind, äquivalent sein müssen. Bei dieser Analyse der Vergleichsmethoden liegt der Schwerpunkt demnach bei den status- und heuristikbasierten Methoden. Besonders in DSMDiff, EMF Compare und X-Diff wird dem Vergleich und dem Auffinden von Äquivalenzen besondere Aufmerksamkeit geschenkt.

Wie bereits erwähnt, verwendet DSMDiff GME als Metametamodell. GME deckt in erster Linie Graphen ab und definiert daher Knoten und Kanten als zentrale Elementtypen. DSMDiff führt das Mapping auf Basis der syntaktischen Eigenschaften dieser beiden Typen durch. Zu Beginn werden die Knoten- und Kantensignaturen verglichen. Diese setzen sich aus einem Subset der syntaktischen Eigenschaften zusammen und beinhalten den Typ, die Art und den Namen der Knoten und zusätzlich die Signatur der Destination bei Kanten. Wenn einige Elemente nicht eindeutig gemappt werden können, also die Signatur nicht exakt dieselbe ist, dann wird ein strukturelles Matching für diese Elemente durchgeführt. Die strukturelle Gleichheit definiert sich durch die Gleichheit der Kanten, die zu diesem Element führen. Wird hierbei eine ausreichende Gleichheit ermittelt, werden jene Elemente als äquivalent betrachtet. DSMDiff unterstützt nur einen 2-Wegsvergleich und geht, wie das UNIX *diff*, davon aus, dass ein Modell der Nachfolger des anderen Modells ist. Daher ist die Modellreihenfolge (`dsmdiff(m1, m2)` oder `dsmdiff(m2, m1)`) entscheidend dafür, ob existierende bzw. nicht mehr existierende Elemente als neu bzw. gelöscht erkannt werden. Eine Änderung wird erkannt, wenn zwei eindeutig gemappte Elemente unterschiedliche Attribute haben. Dies ist natürlich unabhängig von der Richtung und Reihenfolge.

Ähnlich zu DSMDiff wird in X-Diff vorgegangen. Auch hier handelt es sich um einen 2-Wegsvergleich, bei dem angenommen wird, dass eine Version ein Vorgänger einer anderen ist und nicht eine parallel veränderte Version. Auch X-Diff führt einen kontextsensitiven Knotenvergleich durch, wobei ein Knoten entweder ein Element, Attribut oder Text ist. Für jeden Knoten dieser drei Typen wird im ersten Schritt eine Signatur erzeugt. Da XML einer Baumstruktur unterliegt, wo per Definition keine Zyklen existieren, kann der Pfad des Elements, also die Tiefenposition, in seine Signatur mit eingeschlossen werden. Bevor diese Signaturen mit einander verglichen werden, wird zuvor noch für jede Signatur aus Effizienzgründen ein Hash berechnet. Die Hashes werden anschließend verglichen, wobei nur Knoten des selben Typs verglichen werden, um keine unnötigen Vergleiche durchzuführen. Der Vergleich schließt außerdem Äste aus, deren gesamter Hash (inklusive Inhalt) äquivalent ist. Hierdurch kann die Laufzeit in vielen Fällen enorm verringert werden. Bei jedem Knoten, der nicht eindeutig gematcht werden kann, muss, um ein Distanzmaß zu errechnen, die Signatur direkt und nicht nur der Hash verglichen werden. Der Schwellwert, der festlegt, ab wann ein Treffer als solcher identizifiert werden kann, ist dynamisch definiert. Um diesen Schwellwert zu ermitteln, werden zu Beginn einige Knoten des ersten Dokuments ($\sqrt{n}$, wobei n die Anzahl aller Knoten des Dokuments ist) als Sampling zufällig ausgewählt. Für jeden dieser Knoten des Samplings wird die Distanz zu allen Kandidaten im zweiten Dokument errechnet. Der Durchschnitt aller kleinsten Distanzen, also aller besten Treffer, wird als Schwellwert für das folgende, tatsächliche Matching verwendet. So kann gewährleistet werden, dass sich das Matching dynamisch an der Heterogenität der Dokumente orientiert.

EMF Compare arbeitet auf dem Ecore Metametamodell und damit, im Vergleich zu den vorherigen Ansätzen, auf einer komplexeren Datenstruktur. EMF Compare wendet beim Vergleich von Ecore Elementen ein Gewichtungssystem der Informationstheorie an. Eine Instanz einer EMF-Klasse wird anhand *aller* ihrer Attributwerte verglichen. Folgedessen findet der Vergleich der Instanzen auf alle Eigenschaften also Namen, Elternklassen, Referenzen usw. statt. Die Distanz der Attributwerte selbst wird mit dem

Levenshtein-Algorithmus errechnet. Viele Attribute haben bei einem Großteil der Instanzen den selben Wert. Beispielsweise wird die Eigenschaft `isAbstract` vermutlich bei einem großten Teil der Klassen den Wert `false` aufweisen. Aus diesem Grund wird das Gewicht jedes Attributs, also das Maß, wie sehr der Unterschied dieses Attributs zur Gesamtdistanz beiträgt, anhand der Variabilität der Werte angepasst. Bool'sche Werte dienen daher als eher schwache Kriterien im Vergleich zu Zeichenketten, die bei fast allen Instanzen unterschiedlich sind.

Der Vorteil der Flexibilität von heuristischen Methoden ist unbestritten. Der hierfür zu zahlende Preis ist die weitaus schlechtere Performance im Vergleich zu jener bei der Verwendung von UUIDs und die entstehende Unschärfe. Im Fall von EMF Compare kann jedoch das Argument der Unschärfe relativiert werden. Da EMF Compare auf alle Attribute einer Ecore Instanz Rücksicht nimmt, wären auch identifizierende Attribute, wo auch immer diese IDs untergebracht sind, in den Vergleich mit eingeschlossen. Da ein identifizierendes Attribut bei jedem Element einen anderen Wert hat, würden diese auch in besonders hohem Maße Einfluss auf das Matching nehmen und vermutlich in jedem Fall zu einem richtigen Mapping führen. Das bedeutet, dass, sofern IDs vorhanden sind, diese genutzt werden, ohne eine direkte Abhängigkeit von den Editoren zu erzeugen.

4.2 Deltarepräsentation

Die Grundlagen der Deltarepräsentation wurden bereits im Kapitel 3.3 besprochen. Die dort genannten Eigenschaften stellen auch die folgenden Kategorien dieser Analyse dar.

Form: In welcher Form liegen die Differenzen vor? Werden diese symmetrisch oder gerichtet dargestellt?

Modellbasiert: Ist die Deltapräsentation modellbasiert? Gibt es also hierfür ein gemeinsamens Metamodell, sodass die Deltas als Modelle weiterverarbeitet werden könnten?

Metamodellabhängigkeit: Können mit der Deltarepräsentationsform nur die Unterschiede für eine Sprache dargestellt werden oder ist diese Darstellungsform metamodellunabhängig?

Transformativ: Sind die Deltas auf Objektstati ausführbar und können daher zur Transformation der Objektstati verwendet werden? Dies ist mehr eine Funktion, die von einem System implementiert wurde, als eine Eigenschaft der Darstellungsform selbst. Letzlich könnte für praktisch jede Deltarepräsentation eine Art Interpreter implementiert werden, der die Deltas auf Modelle ausführen kann.

Tabelle 4.2 zeigt eine Gegenüberstellung der untersuchten Arbeiten in Bezug auf die oben beschriebenen Kategorien. Die erste Spalte (*Form*) gibt an, ob die Deltas in gerichteter oder symmetrischer Form abgebildet werden. Es wurde nur auf die interne Struktur Bezug genommen und nicht wie etwaige Differenz-Visualisierungen für BenutzerInnen dargestellt werden. Diese Visualisierung für Benutzer ist unabhängig von der internen Repräsentation. Eine symmetrische Abbildung ließe sich schließlich auch aus gerichteten Deltas ableiten. Bei allen Arbeiten, bis auf der von Ohst et al., wurde ein gerichteter Aufbau gewählt. Ohst et al. legen jedoch auch den Schwerpunkt auf der Visualisierung und nicht die Weiterverarbeitung der Deltamodelle. Sie schlagen

Arbeit	Form	Modellb.	Metamodellabh.	Transf.
Alanen [4]	gerichtet	nein	unabhängig	ja
Blanc et al. [6]	gerichtet	nein	unabhängig	ja
Cicchetti et al. [12]	gerichtet	ja	unabhängig	ja
DSMDiff [29]	gerichtet	nein	unabhängig	nein
EMF Compare [8]	gerichtet	ja	unabhängig	ja
Kögel [26]	gerichtet	ja	unabhängig	ja
Mens et al. [32]	gerichtet	nein	unabhängig	ja
MolhadoRef [18]	gerichtet	nein	abhängig	ja
Ohst et al. [33]	symmetrisch	ja	abhängig	nein
SMoVer [5]	gerichtet	nein	unabhängig	nein
Störrle [37]	gerichtet	nein	unabhängig	ja
X-Diff [38]	gerichtet	nein	unabhängig	nein

Tabelle 4.2: Kategorisierung bestehender Arbeiten: *Deltarepräsentation*.

die Anwendung einer *Coloring*-Technik vor. Diese symmetrische Form ist für die Visualisierung sicher passender als gerichtete Darstellungen, da die Differenzmodelle so in der gleichen Syntax wie die ursprünglichen Modelle vorliegen. Außerdem sind alle Unterschiede parallel visualisiert und nicht sequentiell, wie dies bei gerichteten Formen der Fall ist. Die Differenzen sind dadurch für BenutzerInnen leichter und schneller erfassbar. Bei der internen Repräsentation überwiegen jedoch die Vorteile gerichteter Repräsentationsformen. Diese Vorteile wurden bereits in Kapitel 3.3 beschrieben und zeigen sich auch durch die dominante Anwendung in den Ergebnissen dieser Analyse.

In der zweiten Spalte (*Modellb.*) ist ein *ja* vermerkt, wenn die Deltas modellbasiert sind und damit einem gemeinsamen Differenzmetamodell entsprechen. Dies ist bei vier Arbeiten der Fall: Cicchetti et al., EMF Compare, Kögel und Ohst et al. Ohst et al. stellt Deltas, wie bereits beschrieben, in symmetrischer Form als eingefärbte Modelle dar. Das Differenzmodell folgt daher letztlich dem Metamodell der verglichenen Modelle selbst. Cicchetti et al., EMF Compare und Kögel verwenden im Gegensatz dazu für ihre gerichteten Deltas eigens definierte Metamodelle.

In diesem Zusammenhang ist auch die Metamodellabhängigkeit der Deltarepräsentation von Interesse. Diese wird in Spalte 3 notiert und ist identisch mit der Spalte über die Metamodellabhängigkeit im vorangegangenen Kapitel 4.1. Blanc et al., Kögel, Mens et al. und Störrle übersetzen die Ursprungsmodelle entsprechend eines Metametamodell. Alanen und Porres, Cicchetti et al., DSMDiff, EMF Compare und SMoVer unterstützen nur Sprachen, die auf einem gemeinsamen Metametamodell aufbauen. Die Deltarepräsentation ist bei allen Arbeiten, die metamodellunabhängige Repräsentation aufweisen, auf diese Metametasprachen angepasst und kann Änderungen auf dieser Ebene abbilden. Natürlich sind die Differenzen dementsprechend abstrakt und sprachenunabhängig.

Die Arbeit von Cicchetti et al. sticht in diesem Zusammenhang hervor, da das verwendete Differenzmetamodell aus dem Metamodell der Vergleichsmodelle selbst abgeleitet wird und damit Änderungen nicht nur auf der abstrakten Metametaebene, sondern direkt auf der Metaebene repräsentieren kann. Eine ATL-Transformation [25] erzeugt für jede Metaklasse eine `Added*`, `Deleted*` und `Changed*`-Klasse, wobei der Stern der jeweilige Klassenname ist. Durch diese Transformation ist das generierte Differenzme-

tamodell selbst zwar metamodellabhängig, die Methodik als Ganzes allerdings metamodellunabhängig (siehe Spalte 3 *Metamodellabh.*), da die Transformation auf jedes Metamodell angewendet werden kann. Die abgebildeten Unterschiede sind aber dennoch generisch und nicht sprachspezifisch.

Das Differenzmetamodell von EMF Compare und Kögel ist rein generisch und kann für alle Modelle verwendet werden, die dem Ecore Metamodell [9] bei EMF Compare und dem RUSE Metamodell [39] im Fall von Kögel entsprechen. In Kögel werden die Änderungen jedoch außerdem auf den übereinander liegenden Ebenen *Meta Model Layer*, *Model Layer* und *Logical Layer* abgebildet (siehe Bild 4.1).

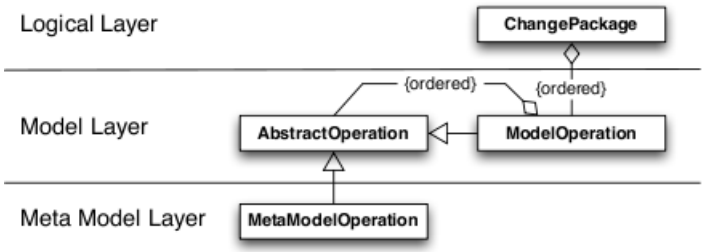

Abbildung 4.1: Die drei Änderungs-Ebenen aus [26].

Da Kögel einen operationsbasierten Ansatz verfolgt (siehe Kapitel 4.1) können die Änderungen auf der Metamodellebene mitgeloggt werden. Eine spezielle Implementierung im Editor für die jeweilige Sprache (z.B. UML Use Case Diagramm) ermöglicht außerdem auch die automatische Erfassung auf Modellebene. Im *Logical Layer* müssen die Änderungen vom User manuell zusammengefasst und auf dieser semantisch höchsten Ebene annotiert werden. Dies gewährleistet eine nachvollziehbare Historie der Änderungen.

Annotationen auf der logischen Schicht werden von EMF Compare nicht unterstützt. Jedoch können ähnlich wie bei Kögel auf dem *Model Layer* sprachspezifische Differenzen durch eine Kombination aus sprachunabhängigen Unterschieden (auf Ecore Metamodellebene) definiert werden. Am Beispiel vom UML 2 Klassendiagramm würde eine Erzeugung einer Assoziation in viele Änderungen auf Metametamodellebene resultieren. Der User würde bei der Betrachtung des Differenzenmodells von den vielen Unterschieden vermutlich erschlagen werden. Erschwerend kommt hinzu, dass diese Unterschiede in einer anderen Sprache (Ecore Metamodell) formuliert sind, die mit der eigentlichen Sprache (UML 2 Klassendiagramm) nichts direkt zu tun hat. Um dieses Problem zu lösen, können Transaktionen auf einer höheren Ebene definiert werden. In unserem Beispiel wäre diese Transaktion `AddNavigableAssociation`, die aus drei `AddModelElement`-Differenzen (eine Assoziation und zwei Eigenschaften) besteht.

Die vierte Spalte (*Transf.*) betrifft die Transformativität. Ein Deltadokument ist transformativ, wenn es auf ein Modell angewendet werden kann, um ein Modell von einer Version in eine andere überzuführen. Letztlich trifft das in gewisser Weise auf alle Systeme mit gerichteten Deltas zu, die eine Merge-Operation implementieren, da bei einem Merge die ermittelten Änderungen auf Objektstati angewendet werden müssen. Eine hohe Wiederverwendbarkeit der Differenzmodelle ist dadurch aber noch nicht sichergestellt. Diese hängt in erster Linie davon ab, ob die Deltas auch *self-contained* und *compositional* sind. Um festzustellen, ob diese Eigenschaften in den Systemen erfüllt sind, bedarf es einer Analyse derer Implementierungen, da diese Aspekte in den Artikeln selbst meist nicht ausreichend beschrieben sind.

Neben den bisher genannten Eigenschaften ist auch die Minimalität bzw. Kompaktheit ein wünschenswertes Charakteristikum von Deltas. Speziell bei operationsbasierten Systemen, wo Änderungen der BenutzerInnen direkt getrackt werden, weisen die ermittelten Deltas mit hoher Wahrscheinlichkeit redundante und sich gegenseitig aufhebende Aktionen auf. Alanen und Porres stellen eine Minimierungsfunktion vor, die die Minimalität von Änderungslisten gewährleistet. Diese berücksichtigt die Erzeugung und anschließende Entfernung von Elementen, die Änderung von Attributwerten, sowie die Manipulation geordneter und ungeordneter Listen.

4.3 Konflikterkennung

Alle untersuchten Arbeiten, die die Konflikterkennung thematisieren, finden generische Konflikte. Diese generische Konfliktfindung entspricht meist den Regeln, die in Tabelle 3.1 aufgelistet sind. Für diese Analyse ist darüber hinaus interessant, ob auch sprachspezifische Konflikte gefunden werden können. Wenn dies der Fall ist, kann ferner die Abhängigkeit der Algorithmen von Sprache und Domäne, also mit welchem Aufwand diese auch für andere Sprachen anwendbar gemacht werden können, als bedeutendes Qualitätskriterium dienen. Ist ein programmtechnischer Eingriff erforderlich oder lässt sich eine neue Sprache durch deskriptive Artefakte konfigurieren? Es stellt sich außerdem die Frage, ob Änderungen auf höherer Ebene wie beispielsweise Refactorings bei der Konflikterkennung berücksichtigt werden können. Dieser Teil der Analyse verwendet daher die folgenden Kategorien, um diese Eigenschaften in den untersuchten Arbeiten zu bewerten.

Sprachspezifische Konflikterkennung: Werden sprachspezifische Aspekte bei der Konflikterkennung berücksichtigt?

Refactorings: Werden Refactorings als solche erkannt und bei der Konflikterkennung beachtet?

Programmeingriff: Mit welchem Aufwand können die Methoden auf andere Sprachen ausgeführt werden? Erfordert dieser einen Programmeingriff oder kann die sprachenabhängige Schicht deklarativ konfiguriert werden?

In Tabelle 4.3 wurde vermerkt, welche Arbeiten sprachspezifische Konflikte und Refactorings unterstützen. Die erste Spalte (*Sprachspez.*) bezieht sich darauf, ob die jeweiligen Arbeiten bei der Konflikterkennung spezifische Aspekte der Modellierungssprache berücksichtigen. Dies trifft nur zu, wenn Konzepte der Sprache (Metamodell) selbst und nicht nur jene der Metasprache (Meta-Metamodell) bei der Erkennung eine Rolle spielen. In der zweiten Spalte (*Refactorings*) ist in weiterer Folge ein *ja* notiert, wenn die jeweilige Arbeit Refactorings erkennt und bei der Konflikterkennung berücksichtigt. Ist dies der Fall, können viele Konflikte sofort aufgelöst und eine Konfliktmeldung verhindert werden. Da sowohl die sprachspezifischen Aspekte (Spalte 1), als auch Refactorings (Spalte 2) sprachabhängig sind, stellt sich die Frage, wie stark die jeweilige Lösung an die Sprache gebunden ist und wie hoch der Aufwand für eine Erweiterung auf andere Sprachen zu Buche schlägt. Diese Bindung an die verwendete Sprache wird in Spalte 3 (*Programmeingriff*) vermerkt.

Wie bereits weiter oben erwähnt, verwenden Alanen und Porres MOF als Metametamodell. Jede darin abgebildete Sprache kann daher behandelt werden. Der Konflikterkennungsprozess berücksichtigt jedoch nur Konzepte, die in MOF definiert sind und

Arbeit	Sprachspez.	Refactorings	Programmeingriff
Alanen [4]	nein	N/A	N/A
Blanc et al. [6]	ja	N/A	Logische Regel
Cicchetti et al. [12]	nein	N/A	N/A
DSMDiff [29]	nein	N/A	N/A
EMF Compare [8]	nein	N/A	Spez. Implementierung
Kögel [26]	nein	N/A	N/A
Mens et al. [32]	ja	ja	Graphtransformation
MolhadoRef [18]	ja	ja	Programmeingriff
Ohst et al. [33]	ja	N/A	Programmeingriff
SMoVer [5]	ja	N/A	Semantic View
Störrle [37]	nein	N/A	N/A
X-Diff [38]	nein	N/A	N/A

Tabelle 4.3: Kategorisierung bestehender Arbeiten: *Konflikterkennung.*

nicht jene des in MOF dargestellten Metamodells selbst. Dies macht die Konflikterkennung naturgemäß allgemeiner und ungenauer. Ebenso wird von Cicchetti et al., Kögel und Störrle, sowie in DSMDiff, EMF Compare, und X-Diff nur das jeweilige Metametamodell in die Konflikterkennung mit einbezogen. EMF Compare sieht jedoch die Möglichkeit vor, eigene Operationen und sprachspezifische Vergleichsalgorithmen zu implementieren.

Im Gegensatz zu den bisher genannten Arbeiten stehen in diesem Zusammenhang die Herangehensweisen von Blanc et al., Mens et al., MolhadoRef, Ohst et al. und SMoVer. Diese ermöglichen eine sprachspezifische Erkennung. Die Vorteile dieser Erkennungsmethode wurde bereits im Kapitel 3.4 erläutert. An dieser Stelle können mit MolhadoRef diese Vorzüge anhand eines Beispiels erneut hervorgehoben werden. Angenommen zwei Personen ändern den Namen von Methoden, sodass zwei Operationen `renameMethod(`m_1`, `m_2`)` und `renameMethod(`m_3`, `m_4`)` vorliegen. Generische Konflikterkennung würde einen Konflikt melden, sofern $m_1 = m_3$, also wenn die geänderte Methode die selbe ist. Durch die sprachspezifische Erkennung kann jedoch zusätzlich ein Konflikt erkannt werden, wenn $m_2 = m_4$ und beide Methoden in der selben Klasse sind. In diesem Fall hätte eine Klasse zwei Methoden mit der selben Signatur, was zu einem inkonsistenten Zustand führen würde. Ohne Kenntnis der Regeln, die die Sprache vorgibt, bliebe dieser Inkonsistenzkonflikt unerkannt. Wenn $m_2 = m_4$ wird in MolhadoRef nicht nur geprüft, ob sich beide Methoden in der selben Klasse befinden, sondern auch, ob eine ungewollte Überschreibung im Sinne der Objektorientierung stattfindet. Dies ist der Fall, wenn sich eine Methode (z.B. m_2) in einer Superklasse jener Klasse, in der die andere Methode (z.B. m_4) platziert ist, befindet.

In Bezug auf Refactorings (Spalte 2) sind Mens et al. hervorzuheben. Refactorings werden dort als Graphtransformationen des Metamodells abgebildet. Durch die Definition von Vorbedingungen wird für jedes Refactoring festgelegt, ob das Refactoring auf einen Objektstatus anwendbar ist oder nicht. Aus diesen expliziten Refactoring-Definitionen ist ebenfalls ableitbar, welche Refactorings, angewendet auf das selbe Modell, miteinander in Konflikt stehen. Ein Konflikt tritt auf, wenn ein Refactoring ein Objekt entfernt oder eines jener Attribute ändert, die von einem anderen Refactoring als Vorbedingung definiert sind. In diesem Fall wird von einem kritischen Paar (*critical pair*) im Sinne

der Graphersetzung (*graph rewriting*) gesprochen.

Bei sprachspezifischen Methoden ist weiterhin interessant, wie stark deren Bindung an das unterstützte Metamodell ist (Spalte 3 *Programmeingriff*). Am stärksten ist diese Bindung wohl bei MolhadoRef und der Arbeit von Ohst et al. Für diese Herangehensweisen müsste vermutlich eine umfassende Implementierung vorgenommen werden.

Beim Ansatz von Blanc et al. ist die sprachabhängige Schicht weitaus geringer. Es können über logische Regeln sprachspezifische Konsistenzprüfungen definiert werden, die die Konsistenz von Modellen überprüfen. Wenn auch im referenzierten Artikel nicht explizit angegeben, könnten mit dieser logikbasierten Technologie auch Refactorings logisch definiert und in weiterer Folge erkannt werden. Die allgemeine Methodik von Blanc et al. ist also metamodellunabhängig und wird durch metamodellabhängige Regeln für sprachspezifische Anforderungen erweitert. Die metamodellabhängige Schicht ist daher auf deskriptive Regeln beschränkt. Blanc et al. diskutiert jedoch nur die Aspekte der Konsistenzprüfung. Konflikterkennung im Sinne der Versionierung bleibt unangetastet.

In SMoVer wird der Fokus hingegen rein auf die Konflikterkennung gelegt. Die Sprachenbindung fällt, wenn auch etwas mehr als bei der Arbeit von Blanc et al., ebenfalls relativ gering aus. Es müssen für jede Sprache bzw. deren zu überprüfende Aspekte nur Metamodelle (Semantic Views Definition) und Transformationen (Modell in Semantic View) entwickelt werden.

Im Zusammenhang mit Metamodellabhängigkeit und Refactorings kann erneut die Arbeit von Mens et al. genannt werden. Die sprachspezifischen Refactorings sind durch getypte Graphtransformationen definiert. So können Refactorings für praktisch jede Modellierungssprache abgebildet werden, ohne eine erneute Implementierung erforderlich zu machen. Es handelt sich also um eine äußerst minimale Art und Weise, wie sprachspezifische Aspekte definiert werden.

In diesem Kapitel wurden einige bestehende Systeme vorgestellt und deren Vor- und Nachteile diskutiert. Auch wenn viele der vorgestellten Arbeiten hervorragende Aspekte enthalten und einen wertvollen Beitrag leisten, bietet keines dieser Systeme eine ganzheitliche Lösung für eine adäquate Konflikterkennung in der Modellversionierung. Im folgenden Kapitel 5 werden daher die Anforderungen an die Konflikterkennung abgeleitet und eine mögliche Umsetzung bezugnehmend auf die diskutierten Vor- und Nachteile präsentiert.

5 Anforderungen an den Modellvergleich und seine Realisierung

Nachdem in den letzten Kapiteln die unterschiedlichen Methoden und Eigenschaften des Modellvergleichs, der Deltarepräsentation und der Konflikterkennung besprochen und bestehende Arbeiten in diesem Bereich analysiert wurden, folgt in diesem Kapitel eine Beschreibung der Anforderungen an ein neues, im Zuge dieser Arbeit zu entwickelndes, Modellvergleichs- und Konflikterkennungssystem. Anschließend werden die zum Großteil bereits vorgestellten Techniken in Hinsicht auf die Erfüllung der Anforderungen diskutiert.

5.1 Anforderungen

Im Kontext der Modellversionierung gehören *Zuverlässigkeit*, *Vollständigkeit* und *Genauigkeit* zu den wichtigsten Anforderungen an den Modellvergleich. Das Ergebnis des Modellvergleichs, also der Änderungs- oder Differenzbericht, dient nicht nur als Report für User, sondern ist auch die Grundlage für eine anschließende Konflikterkennung, Konfliktresolution und Zusammenführung. Unzuverlässige, ungenaue oder unvollständige Differenzen verhindern eine qualitativ hochwertige und intelligente Konflikterkennung, -resolution und Zusammenführung der Modellversionen. Damit ist die hohe Qualität der Ergebnisse einer Vergleichs- und Konflikterkennungskomponente ein erfolgsbestimmendes Kriterium für das gesamte Modellversionierungssystem. Zuverlässigkeit und Genauigkeit bedeuten in diesem Zusammenhang, dass Unterschiede exakt so erkannt und abgebildet werden, wie sie vorliegen. Beispielsweise ist es erforderlich, dass das Verschieben und die starke Veränderung eines Elements nicht als Löschen und anschließendes Hinzufügen eines neuen, anderen Elements missverstanden werden. Denn das kann für die Erkennung spezifischer Konflikte und in weiterer Folge für die Anwendung eines Resolutionsmusters von großer Bedeutung sein.

Besonders seit der weiten Verbreitung domänenspezifischer Modelle (*Domain Specific Languages*) ist oft eine Vielzahl unterschiedlicher Metamodelle innerhalb eines Softwareprojekts in Verwendung. Ein Modellversionierungssystem darf daher nicht nur die Aspekte einer Modellierungssprache berücksichtigen, sondern muss ein maximales Maß an Funktionen für viele Sprachen bieten und für besondere sprachspezifische Eigenschaften adaptierbar sein. Die Ermittlung eines generischen Änderungsberichts und die Erkennung generischer Konflikte sollte ohne jegliche Anpassung des Systems möglich sein. Grundvoraussetzung für diese Anforderung ist die Metamodellunabhängigkeit des gesamten Systems. Die Algorithmen dürfen daher nicht sprachspezifische Schnittstellen verwenden, sondern müssen auf generische Art und Weise mit den Modellen umgehen

können. Eine rein generische Logik ermöglicht jedoch keine Berücksichtigung sprachspezifischer Aspekte des verwendeten Metamodells und bietet daher oftmals keine ausreichende Unterstützung bei der Konfliktbeseitigung. Das System muss also Schnittstellen anbieten, die zur Erweiterung der Grundfunktionalität um Sprachspezifika verwendet werden können. Dieser Erweiterungsmechanismus sollte aber, soweit möglich, nicht durch programmatische Eingriffe oder Erweiterungen stattfinden, sondern aus deskriptiven Artefakten bestehen, die in das System eingebunden werden können. Durch diese deskriptiven Artefakte soll die Möglichkeit für BenutzerInnen geschaffen werden, das System insbesondere in Hinsicht auf die Erkennung zusammengesetzter Operationen, Refactorings und sprachspezifischer Konflikte zu erweitern.

Die Notwendigkeit für die Definitionsmöglichkeit zusammengesetzter Operationen liegt auch in der Erzielung der Metamodellunabhängigkeit durch die Verwendung der Schnittstellen des Meta-Metamodells. Wenn der Vergleich und damit die Differenzermittlung rein generisch, also über die Schnittstellen des Meta-Metamodells geschehen, sind die ermittelten Operationen, auf der Meta-Metaebene, oftmals abstrakt und komplex. Eine Änderung auf Metaebene, also auf der Ebene der verwendeten Sprache selbst, erzeugt oftmals viele Änderungen aus Sicht der Meta-Metaebene. Beispielsweise führt das Hinzufügen einer *Association* im UML 2 Klassendiagramm (auf Ecore-Basis) insgesamt zu drei Operationen in der Meta-Metasicht. Diese sind das Hinzufügen der Assoziation als Modellelement selbst und für jede referenzierte Klasse jeweils das Hinzufügen einer *Property*, die die Assoziation referenziert. Das ist natürlich davon abhängig, wie das jeweilige Metamodell auf dem Meta-Metamodell abgebildet ist. Die Differenzen werden demzufolge nicht in der Modellierungssprache des Users dargestellt und sind daher für diesen teils unklar und schwer erfassbar. Die Erkennung und Darstellung sprachspezifischer, zusammengesetzter Operationen kann dieses Problem lösen. Außerdem können zusammengesetzte Operationen als *Information Hiding* [36] im Sinne der Softwareentwicklung verstanden werden. Sie kapseln spezifisches Wissen und erleichtern die weiterführende Verarbeitung, wie beispielsweise die Definition sprachspezifischer Konflikte. Ein weiterer Grund für die Notwendigkeit der Erkennung sprachspezifischer Operationen ist die Berücksichtigung von Refactorings. Ein Refactoring ist eine besondere Form einer zusammengesetzten Operation, da es viele kleinere Änderungen auslöst, obwohl hinter ihm nur eine einzige Intention des Users steht. Die Erkennung von Refactorings kann, wie bereits im Kapitel 3.4 beschrieben, die Qualität der Konflikterkennung und -resolution maßgeblich erhöhen.

Neben den zusammengesetzten Operationen und Refactorings gibt es einen weiteren sprachspezifischen Aspekt, der über deskriptive Artefakte konfigurierbar sein muss. Hierbei handelt es sich um die Definition von Konflikten, die nicht durch die rein generische Betrachtung auf Meta-Metaebene erkennbar ist (siehe Kapitel 3.4.2). Wie bereits im Kapitel 3.4 beschrieben und in Abbildung 3.4 dargestellt wurde, können Konflikte in Hinsicht auf deren Definition operationsbasiert und ergebnisbasiert abgebildet werden. Operationsbasierte Konfliktdefinitionen decken jene Konflikte ab, die durch widersprüchliche Operationsszenarien entstehen. Als Beispiel kann hier der Fall dienen, dass eine Person ein *Interface* aus einer Klasse abstrahiert und eine andere Person eine abstrakte Klasse (siehe Abbildung 9.3). Theoretisch liegt kein sprachspezifischer Verstoß im resultierenden Modell vor. Im üblichen Fall sind diese beiden Operationen allerdings widersprüchlich und sollten folglich als Konflikt gemeldet werden. Das System muss daher eine Schnittstelle bieten, um widersprüchliche Operationsszenarien als Konflikt zu definieren und diese Definition in das System einzubinden. Die zweite Möglichkeit, Konflikte zu definieren, ist ergebnisorientiert. Das bedeutet, dass die

Ausführung zweier Operationsfolgen zu einem invaliden Modell führt, ohne dass dies bereits bei der Analyse der Operationen ersichtlich war. Derartige Regeln entsprechen im Prinzip den Validationsregeln für Modelle. Das System muss daher das resultierende Modell testweise erzeugen, es gegen zuvorgenannte Validationsregeln testen und, sofern ein Verstoß gegen eine dieser Regeln vorliegt, einen Rückschluss auf eine oder mehrere, verursachende Operationen ziehen. Diese Operationen sind in weiterer Folge als Konflikt auszuzeichnen und in den Konfliktbericht aufzunehmen.

Da mehrere Sprachen in Projekten Verwendung finden, kommen oftmals auch unterschiedliche Editoren zur Erstellung dieser Modelle zum Einsatz. Ebenso wie die alleinige Unterstützung für eine Sprache ist daher auch die Abhängigkeit von der Verwendung eines speziellen Editors nicht wünschenswert. Die Anwendung eines Editors sollte daher zumindest kein Ausschlusskriterium für die Verwendung des gesamten Versionierungssystems sein.

In Hinsicht auf den Änderungsbericht können neben den genannten Anforderungen weitere wünschenswerte Eigenschaften identifiziert werden. Im Kontext der Modellversionierung spielen Modelle eine zentrale Rolle. Die User sind gewohnt, mit Modellen umzugehen und diese beispielsweise für Transformationen weiter zu verwenden. Der Änderungsbericht sollte daher ebenfalls modellbasiert sein und damit einem definierten Metamodell für Differenzen folgen. Natürlich ist bei diesem Metamodell, wie beim Gesamtsystem ein wichtiges Kriterium, dass damit Differenzen zwischen Modellen, unabhängig der verwendeten Sprache, abgebildet werden können. Beim Umgang mit Änderungen mehrerer EntwicklerInnen ist es oftmals von Vorteil, Änderungen exportieren und speichern sowie diese Änderungsskripte selbst zusammenführen zu können (*Compositionality*). Ein einzelnes oder kombinierte Differenzmodelle müssen in diesem Zusammenhang auch transformativ, also auf Modelle ausführbar sein, sodass eine Version eines Modells in eine nachfolgende oder komplett neue Version überführt werden kann. Kompositionelle und transformative Differenzmodelle sind besonders im Zusammenhang mit dem *Branching* in Versionierungssystemen nützlich. Denn nur so können Änderungen eines Branches in einen anderen selektiv übernommen werden.

In der folgenden Liste werden die zuvor erläuterten Anforderungen zur besseren Übersicht noch einmal zusammengefasst.

- Metamodellunabhängigkeit
- Zuverlässigkeit, Vollständigkeit und Genauigkeit
- *Out-of-the-box*-Erkennung generischer Operationen und Konflikte
- Deskriptive Erweiterungsmechanismen für sprachspezifische Aspekte
  - Definition und Erkennung zusammengesetzter Operationen
  - Definition und Erkennung von Refactorings
  - Operationsbasierte Definition und Erkennung sprachspezifischer Konflikte
  - Ergebnisbasierte Definition und Erkennung sprachspezifischer Konflikte
- Unabhängigkeit vom verwendeten Editor
- Modellbasierter Änderungsbericht
- Transformativer (ausführbarer) Änderungsbericht
- Kompositioneller (kombinierbarer) Änderungsbericht

5.2 Diskussion verschiedener Techniken

Im vorigen Abschnitt wurden die Anforderungen an eine zuverlässige Konflikterkennung im Rahmen eines Versionierungssystems für Modelle besprochen. Nun werden diese Anforderungen in Bezug auf deren Realisierung anhand der bereits vorgestellten Techniken aus dem Kapitel 3 diskutiert.

Die erste Anforderung, die *Metamodellunabhängigkeit*, wird üblicherweise mit Hilfe einer von zwei Techniken erreicht. Das ist einerseits die Durchführung einer vorherigen Transformation der Status und andererseits die Nutzung der *generischen Schnittstellen des Meta-Metamodells*. Die vorherige Transformation übersetzt die Modelle auf ein allgemeines Metamodell. Üblicherweise hat dieses allgemeine Metamodell eine graphenähnliche Struktur, mit der die Vergleichs- und Konflikterkennungsalgorithmen arbeiten. Somit kann jede Modellierungssprache verarbeitet werden, für die diese Transformation existiert. Auch wenn es von Vorteil ist, dieses allgemeine Metamodell nach den eigenen Bedürfnissen zu designen, müsste für jede existierende bzw. neu entstandene Sprache eine solche Transformation entworfen werden. Dies kann ein zeitintensiver Prozess sein. Verlässt man sich hingegen auf die Schnittstellen eines Meta-Metamodells, können ohne zusätzlichen Aufwand, alle Sprachen unterstützt werden, deren Definition auf Basis dieses Meta-Metamodells erfolgt. Mit der Verwendung eines populären Meta-Metamodells, wie beispielsweise Ecore, hat man so von Beginn an eine breite Unterstützung für viele Sprachen. Sollten auch Sprachen unterstützt werden müssen, deren Meta-Metamodell nicht dem gewählten entspricht, kann man sich über die Implementierung eines Model-to-Model-Mappings auf der Meta-Metaebene helfen. Im Vergleich zum Ansatz der vorherigen Transformation wird der Bedarf einer Transformation bzw. eines Mappings deutlich seltener auftreten. Die Nutzung der Meta-Metamodellschnittstellen hat damit einen großen Vorteil. Der einzige Nachteil ist, dass die Meta-Metamodellschnittstellen nicht auf die Verwendung innerhalb des Versionierungssystems optimiert und damit teils etwas komplexer und abstrakter sind. Dieser Nachteil ist allerdings in Abwägung zum zuvor genannten Vorteil vernachlässigbar.

Weitere, besonders wichtige Anforderungen sind *Zuverlässigkeit*, *Vollständigkeit* und *Genauigkeit*. In Bezug auf diese Anforderung steht besonders die verwendete Technik des Matchs im Vordergrund. Hierzu wurden im Kapitel 3 insbesondere zwei Herangehensweisen vorgestellt. Diese sind einerseits die Verwendung von UUIDs oder andererseits die von Heuristiken, mittels derer die Ähnlichkeit zweier Modellelemente geschätzt wird. Heuristiken, die meist mithilfe von Techniken der Informationstheorie und verbreiteten Methoden der Distanzberechnung arbeiten, können eine beeindruckende Qualität erzielen. Dennoch ist das Ergebnis von Heuristiken stets eine Schätzung und keine verlässliche Aussage. Insbesondere, wenn man davon ausgeht, dass Fälle existieren, bei denen keine Heuristik einen zuverlässigen Match erzielen kann. Wenn ein Modellelement beispielsweise stark verändert und zudem noch verschoben wurde, ist es praktisch unmöglich, den Ursprung des Element wiederzuerkennen. Nachdem in diesem Fall ein heuristikbasierter Match den tatsächlichen Ursprung nicht erkennen kann, würde die Änderung und die Verschiebung des Elements als Entfernung und neu hinzugefügtes Element erkannt werden. Eine derartige Unschärfe ist jedoch, wie bereits im letzten Abschnitt besprochen, bei der Modellversionierung nicht erstrebenswert. Letztlich muss also die Verwendung von Heuristiken aus diesen Gründen abgelehnt werden. Die Alternative, also der Einsatz von eindeutigen Attributen zur Elementidentifizierung, bringt jedoch wiederum eine Abhängigkeit vom verwendeten Editor mit sich, was in weiterer Folge die Anforderungserfüllung der Editorunabhängigkeit erschwert. Es lassen sich

daher keine dieser beiden Techniken direkt anwenden. Es können jedoch Lösungen gefunden werden, wenn diese Entscheidung nicht isoliert, sondern in Kombination mit der Basis des Vergleichs, betrachtet wird.

In Bezug auf die Basis des Vergleichs wurden in Kapitel 3 zwei unterschiedliche Techniken besprochen. Der Vergleich ist demnach entweder operationsbasiert oder statusbasiert. Bei einem operationsbasierten Vergleich werden alle Änderungen des Users direkt vom Editor mitgeschrieben und zur weiteren Verarbeitung gespeichert. Die Modifikation eines veränderten Elements wird durch eine Operation ausgedrückt, die den Unterschied vom Ursprung zum modifizierten Element abbildet. Veränderte Elemente sind daher durch eine Operation mit ihrem Ursprung verbunden. Der Match veränderter Elemente ist daher implizit über die Operationen gegeben. Unveränderte Elemente, die dementsprechend nicht durch eine Operation mit ihrem Ursprung verknüpft sind, sind ident mit dem ursprünglichen Element und können daher einander leicht zugeordnet werden. Wenn diese jedoch nicht gematcht werden können, wurden diese Elemente vermutlich neu hinzugefügt. Folglich liegt beim operationsbasierten Ansatz immer ein perfekter Match vor. Die zuvor angesprochene Entscheidung, ob IDs oder Heuristiken zu Match-Erstellung verwendet werden sollen, stellt sich also in diesem Fall gar nicht. Dennoch bleibt beim operationsbasierten Ansatz im Allgmeinen das selbe Problem wie bei der Verwendung von IDs bestehen – nämlich die *Editorabhängigkeit*. Der Editor muss schließlich die Änderungen mitschreiben. Es wird jedoch in Kapitel 7 eine Implementierung des operationsbasierten Vergleichs vorgestellt, die diese Abhängigkeit auf die Plattform (Eclipse) beschränkt.

Der operationsbasierte Vergleich hat einen weiteren großen Vorteil. Durch das direkte Mitschreiben der Änderungen können auch zusammengesetzte Operationen und Refactorings direkt erkannt werden und müssen nicht erst im Nachhinein abgeleitet werden. Dies setzt allerdings voraus, dass eine Implementierung für einen konkreten Editor und für eine spezielle Sprache existiert. Diese Implementierung, zumindest jene der Erkennung von sprachspezifischen Operationen, ist daher in höchstem Maße editorabhängig und nur schwer wiederverwendbar. Die Erkennung muss schließlich für jeden Editor speziell angepasst werden, da jeder Editor beispielsweise die Ausführung von Refactorings andersartig zur Verfügung stellt und auslöst. Außerdem stellen Editoren nicht nur die Ausführung in unterschiedlicher Art bereit, sondern führen zusätzlich die Refactorings selbst auf teilweise unterschiedliche Weise durch. Eine allgemeine, deskriptive Definition sprachspezifischer Operationen, wie sie in den Anforderungen motiviert wurde, ist daher schwer realisierbar, da die Logik eng mit dem Editor verknüpft ist und deren Erkennung in deren spezifischer Implementierung versteckt wird. Um außerdem die Refactorings bei der Konflikterkennung effizient zu nutzen, müssten diese auf eine einheitliche Art und Weise definiert sein. Es wäre daher ein Mapping der editorspezifischen Implementierung der Operationen zu den allgemein definierten Operationen erforderlich. Dieses Problem könnte gelöst werden, indem ein allgemeines Metamodell für generische sowie sprachspezifische Operationen definiert wird, das die Editoren implementieren. Diese Lösung wurde beispielsweise in [26] vorgeschlagen. In diesem Fall ist das Problem jedoch nur auf die Editorenhersteller abgewälzt, nicht aber gelöst. Eine zuverlässige Funktion des Versionierungssystems wäre so nicht sichergestellt, da ein kritischer Teil des Systems von Drittanbietern korrekt implementiert werden müsste. Außerdem ist nicht gewährleistet, dass ein Refactoring innerhalb eines Editors und damit die Erkennung des Refactorings überhaupt zur Verfügung steht. Das bedeutet, dass möglicherweise ein Refactoring manuell durchgeführt wird und dieses daher nicht als solches im Differenzmodell erkannt und ausgezeichnet wird. Zusammenfassend kann ge-

folgert werden, dass ein allgemein einsetzbares Versionierungssystem zumindest nicht ausschließlich von einem Operationstracking eines Editors abhängig sein sollte. Dieses Tracking stellt schließlich ein Ausschlusskriterium für die weitere Funktionsweise des Systems dar. Wurden keine Änderungen mitgeschrieben oder liegen diese in einer unzuverlässigen Qualität vor, ist der gesamte Nutzen des Systems in Gefahr.

Die Alternative zum operationsbasierten Ansatz ist die im Nachhinein durchgeführte Ermittlung der Änderungen auf Basis des Ursprungs und der modifizierten Version. Der Vergleich basiert daher auf den Status der Modelle und ist damit statusbasiert. Wählt man diese Herangehensweise zur Implementierung eines Versionierungssystems, ist man zwar unabhängig von den eingesetzten Editoren, muss allerdings zu allererst, das Problem des Matchs lösen. Dieser muss, wie bereits beschrieben, im Kontext der intelligenten Konflikterkennung und -resolution perfekt und vollständig sein. Daher kann man sich nicht auf Heuristiken verlassen, die die Äquivalenz der Modellelemente nur schätzen. Im Versionierungskontext existiert allerdings auch ein Zeitpunkt, wo die Ursprungsversion und die Version, auf der gearbeitet wird, exakt identisch sind. Dieser Zeitpunkt ist gleich nach dem *Checkout*. Sind die beiden Versionen identisch, kann ein perfekter 1:1-Match durchgeführt werden. Dieser muss allerdings in irgendeiner Form während dem gesamten Veränderungsprozess beibehalten werden. Das kann auf zwei Arten realisiert werden. Entweder behält man sich Referenzen auf äquivalente Elemente im Speicher oder annotiert diese in Form von *transienten* IDs, die nur während der Bearbeitungssession ihre Gültigkeit behalten. Es könnte daher direkt nach dem *Checkout* des Modells und dem 1:1-Match vom Versionierungssystem selbst jedes Element mit einer ID versehen werden. Damit wird die Editorabhängigkeit des ID-basierten Matchs minimiert. Der Editor darf die eben hinzugefügten IDs nur nicht entfernen, was sich nicht als großes Problem erweisen sollte. Im Gegensatz zum konventionellen ID-basierten Ansatz stellt das eine grundlegende Verbesserung dar. Bei diesem musste der Editor IDs bei jeder Erstellung von Elementen erzeugen und vor allem diese an exakt jene Position platzieren, wo es das Versionierungssystem erwartet. Eine mögliche Implementierung der transienten IDs wird in Kapitel 7 beschrieben.

Aus der Verwendung der statusbasierten Methode ergeben sich, neben den bereits genannten Vorzügen, einige zusätzliche Vorteile. Die Erkennung von zusammengesetzten, sprachspezifischen Operationen findet nicht während deren Durchführung statt, sondern muss im Nachhinein vorgenommen werden. Wie bereits weiter oben erwähnt, sollen sprachspezifische Operationen durch deskriptive Definitionen vom Benutzer spezifizierbar sein. Diese Definition dient bei dieser Methode daher als Erkennungsmuster, das vom Versionierungssystem alleine genutzt wird, um generische Differenzen um zusätzliche Informationen zu erweitern. Sowohl die Verwendung als auch die Lagerung der Definitionen ist damit im Versionierungssystem zentralisiert und an einem logischen Ort vereint. Dies erleichtert die Wartbarkeit und gewährleistet die sofortige Wirkung von Änderungen der Definitionen. Beim operationsbasierten Ansatz müsste bei einer solchen Änderung diese entweder in allen Editoren nachgetragen werden oder auf deren Realisierung durch die Editorhersteller gewartet werden, bevor diese neuen oder geänderten Definitionen Nutzen und Wirkung erzeugen. Diese zentralen und expliziten Operationsdefinitionen sind weitaus wiederverwendbarer, als wenn diese in der Implementierung des Editors verankert sind. In Kapitel 8 wird neben der detaillierteren Beschreibung dieser Vorteile ein Ansatz vorgestellt, der die benutzerfreundliche und beispielgetriebene Definition zusammengesetzter, sprachspezifischer Operationen und Refactorings erlaubt.

Schlussendlich stellt sich noch die Frage nach der Repräsentationsform der Deltas. Diese ist jedoch einfach zu beantworten. Da laut Anforderungsanalyse beide Formen realisierbar sein sollten, muss eine gerichtete, modellbasierte Deltarepräsentation verwendet werden, die sich auf symmetrische Art und Weise darstellen lässt. In Kapitel 7 wird hierfür das Metamodell für Differenzen von EMF Compare verwendet.

Die folgende Auflistung fasst daher die aus dieser Diskussion resultierende Wahl der Techniken noch einmal kurz zusammen.

Perfekter Match: Der perfekte Match wird durch einen 1:1-Match zum Zeitpunkt des *Checkout*s durchgeführt und über Referenzen oder *transiente* IDs während der Änderungssession beibehalten.

Vergleich: Statusbasierter Vergleich und *retrospektive* Erkennung von sprachspezifischen, zusammengesetzten Operationen.

Repräsentationsform: Verwendung des Metamodells für Differenzen von EMF Compare und damit gerichteter, modellbasierter Repräsentationsform, die auch zur symmetrischen Darstellung verwendet werden können.

Metamodellunabhängigkeit: Verwendung der Meta-Metamodellschnittstellen und Ecore als Meta-Metamodellsprache.

6 Infrastruktur

Das folgende Kapitel gibt eine Übersicht über die Plattformen und Softwaresysteme, die als Infrastruktur der im folgenden Verlauf dieser Arbeit beschriebenen Implementierungen und Konzepte dienen. Diese sind allesamt quelloffen und auf mehreren Betriebssystemen, darunter Linux, Windows und Mac OS X, lauffähig. Der folgende Überblick umfasst jedoch nur jene Aspekte der besprochenen Systeme, die für das Verständnis im weiteren Verlauf der Arbeit nötig sind.

6.1 Eclipse und Eclipse Plug-Ins

Eclipse [2] ist ein Open-Source Projekt (CPL[1]), welches sich zur Aufgabe gemacht hat, eine plattform-unabhängige, integrierte Tool-Plattform zur Verfügung zu stellen. Die wohl populärste Anwendung dieser generischen Plattform ist die Entwicklungsumgebung (IDE[2]) für Java. Neben dieser gibt es aber noch unzählige andere Module, die Eclipse-Anwendungsbereiche vom Reporting bis zur Entwicklungsumgebung für diverse Programmiersprachen, wie beispielsweise C++, Ruby und PHP erschließen.

Hinter der Entwicklung von Eclipse steht ein Konsortium vieler Firmen, die das Projekt mit Arbeitszeit, Wissen und Technologie versorgen und somit die Featurevielfalt und Qualität vorantreiben. Da ein Projekt derartiger Größe nicht zentral organisiert werden kann, wurden drei Hauptprojekte mit eigenen Verantwortlichkeiten und Themenbereichen gegründet. Diese sind *(i)* das Eclipse Project, *(ii)* das Tools Project und *(iii)* das Technology Project. Jedes dieser Hauptprojekte besteht wiederum aus einer Vielzahl an Unterprojekten, wie beispielsweise das Java Developement Tools Projekt (JDT) oder das C/C++ Development Tools Projekt (CDT) [9].

Da es nicht das Ziel von Eclipse ist, eine spezifische IDE zur Verfügung zu stellen, sondern vielmehr eine Plattform darstellt, um jede mögliche IDE zu entwickeln, ist die gesamte Eclipse Plattform nur aus Plug-Ins zusammengestellt, die alle ein gewisses Featurespektrum zur Plattform beisteuern. Ein Plug-In beinhaltet all das, was zur Ausführung seines Beitrags nötig ist. Dies umfasst üblicherweise alles von Java-Code über Bildressourcen bis hin zu den Abhängigkeiten zu anderen Plug-Ins. Die zentrale Beschreibung eines Plug-Ins ist die Datei *plugin.xml*, die die folgenden Informationen beinhaltet [9]:

- **Requires** – Abhängigkeiten zu Bibliotheken und anderen Plug-Ins.
- **Exports** – Die Sichtbarkeit eigener, öffentlicher Klassen, die das Plug-In anderen Plug-Ins zur Verfügung stellt.

[1] CPL – Common Public License.
[2] IDE – Integrated Development Environment.

- **Extension Points** – Öffentliche Deklarationen der Funktionalität, die das Plug-In anderen Plug-Ins anbietet.
- **Extensions** – Öffentliche Deklarationen der eigenen Verwendung von Funktionalitäten anderer Plug-Ins.

Innerhalb dieser Datei werden also die Abhängigkeiten, der Beitrag, die Punkte, wo dieses Plug-In innerhalb der Plattform eingebunden werden soll (*Extensions*) und die Punkte, die andere Plug-Ins nutzen können, um innerhalb dieses Plug-Ins eingebunden zu werden (*Extension Points*), beschrieben.

6.2 EMF und Ecore

Das Eclipse Modeling Framework (EMF) [9] ist eine Sammlung an Eclipse Plug-Ins, die ein mächtiges Rahmenwerk für die Modellierung innerhalb der Eclipse Plattform bietet. Es beinhaltet mit dem Ecore Metamodell als Herz dieses Frameworks ein zentrales, allgemeines Meta-Metamodell, das zur Definition verschiedenster Sprachen verwendet werden kann. Beispielsweise wurde Java, UML und XML Schema in Ecore abgebildet und ermöglicht daher eine vereinheitlichte Sicht auf diese (Modellierungs-)Sprachen. Ecore selbst ist wiederum in Ecore definiert. Um die Methodik hinter Ecore und dem gesamten EMF zu verstehen, ist die Kenntnis der Konzepte der Metamodellierung nötig. Diese werden jedoch nicht innerhalb dieser Arbeit besprochen.

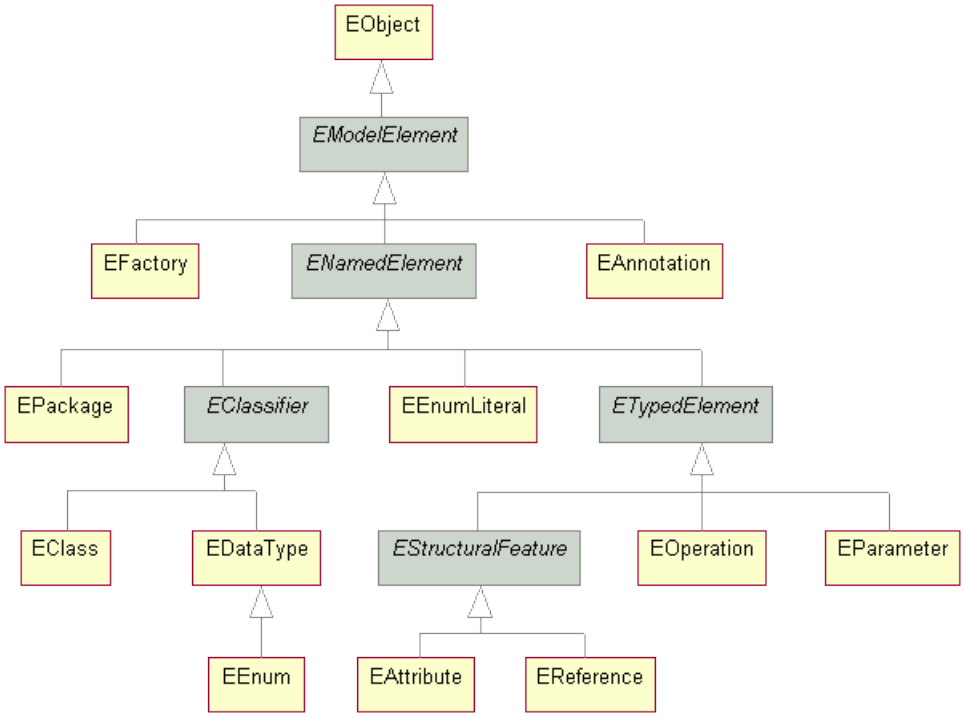

Abbildung 6.1: Das Ecore Metamodell aus [9].

In Abbildung 6.1 ist ein Überblick über das Ecore Metamodell dargestellt. Auf den ersten Blick erinnern die Namen der Klassen dieses Metamodells an jene der UML. Dies ist kein Zufall. Ecore ist ein minimales, vereinfachtes Subset des UML Klassendiagramms

der OMG, ähnlich wie MOF [9]. MOF und Ecore bilden beide die Konzepte einer Klasse, ihrer Beziehungen und ihrer strukturellen Eigenschaften ab. Es wurde jedoch bei der Definition von Ecore versucht, unnötige Komplexität zu verhindern und die Abbildung in Hinsicht auf deren Implementierung zu optimieren. Dies hat zur relativ einfachen und dennoch mächtigen Form von Ecore geführt.

Neben der Abbildung unterschiedlicher Sprachen in Ecore und der Abbildung, sowie Implementierung von Ecore selbst, bietet das EMF auch die Serialisierung eines jeden Ecore-basierten Modells in XMI, einem standardisierten Austauschformat für Modelle. Ecore-basierte Modelle können neben der direkten Erstellung in einem Baumeditor oder einem graphischen Editor auch aus annotiertem Java-Code, aus UML, XML Schema und anderen Sprachen abgeleitet werden. Auf der anderen Seite sind Ecore-basierte Modelle wiederum in Java-Code durch Codegenerierung und Template-Technologien übersetzbar. Es werden auch Aspekte der Evolution und Regenerierung der Modelle berücksichtigt, um auch Änderungen im Code bei erneuter Codegenerierung nicht zu verlieren. Nicht zuletzt durch den starken Einfluss von Persönlichkeiten, wie beispielsweise Erich Gamma [22], fanden viele Entwurfsmuster ihren Einzug in das Framework. Beispielsweise können auf alle Instanzen von EMF-Modellen Adapter gesetzt werden, die mittels der Notification API über die Veränderung der Modellinstanzen im Laufe der Zeit benachrichtigt werden. Die Erzeugung von Instanzen geschieht stets über Factories und die Verwendung derselben über Interfaces. Jedes Objekt implementiert das Interface eines *EObject*s und kann damit über diese rein generische, reflektive API mit Daten befüllt, verändert und ausgelesen werden. Mit Hilfe dieser API können auch Metamodelle ohne die Generierung von Code programmatisch erstellt und Instanzen dieses Metamodells erzeugt werden.

6.3 EMF Compare

EMF Compare [8] ist ein Unterprojekt des Eclipse Modeling Framework Technology (EMFT) Projekts und erweitert EMF um Modellvergleichsfunktionen. Es wurde daher auch als bestehende Arbeit aus diesem Bereich in die Analyse in Kaptel 4 mit eingeschlossen. EMF Compare ist durch eine Reihe an Plug-Ins realisiert und bietet vor allem die folgenden Funktionalitäten.

- Match-Metamodell
- Diff-Metamodell
- Heuristischer Match für Ecore-basierte Modelle
- Statusbasierter 2-Wegs- und 3-Wegsvergleich
- Visualisierung der Unterschiede (Instanzen des Diff- und Match-Metamodells)
- Ausführung von Unterschieden (Merge)

EMF Compare überschreibt die normale Vergleichsfunktion von Eclipse, sodass der Modellvergleich aktiv wird, sobald eines der konfigurierten Modelltypen verglichen werden soll. Welche Modelltypen den Vergleich von EMF Compare auslösen sollen und welche nicht, kann in *Preferences/General/Content Types* in der Eclipse Workbench konfiguriert werden. Wenn beispielsweise im Navigationsbereich von Eclipse zwei Ecore-Modelle markiert sind und man im Kontextmenü *Compare With/Each other* wählt,

werden die markierten Modelle, unabhängig davon, ob sie den selben Ursprung haben oder nicht, miteinander verglichen. Wenn drei Modelle ausgewählt werden, wird man vor dem Vergleich gefragt, welche der drei Dateien, der Ursprung der anderen ist. Wird hier das richtige Modell ausgewählt, kann der statusbasierte 3-Wegsvergleich durchgeführt und das Ergebnis visualisiert werden.

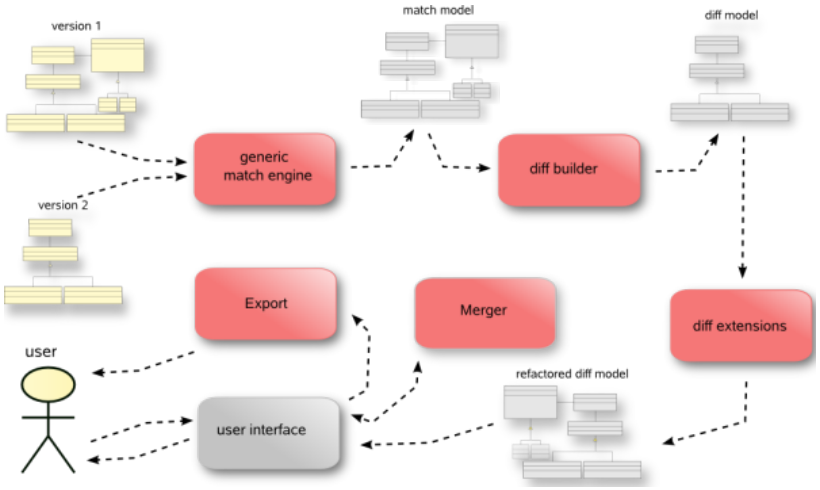

Abbildung 6.2: EMF Compare Architektur aus [7].

In Abbildung 6.2 wird ein Überblick über die API-Architektur von EMF Compare gegeben. Alle roten Rechtecke stellen austauschbare bzw. erweiterbare Teile von EMF Compare dar. Den Beginn bei einem Vergleichsprozess macht die generische Match-Engine. Diese vergleicht zwei Modelle, unabhängig von ihrem Metamodell, und erzeugt ein Match-Modell, das die gefundenen Äquivalenzen beinhaltet. Da innerhalb dieser Arbeit versucht wurde, heuristische Methoden wegen ihrer Unschärfe zu vermeiden, wurde dieses Modul in den vorgestellten Implementierungen durch ein exakteres Verfahren ersetzt. Bezüglich des Match-Modells kann noch angemerkt werden, dass es jeweils eine Ausprägung dieses Modells für zwei Modelle bzw. für drei gibt. Bei drei Modellen ist eines immer als Ursprungsmodell auszuzeichnen. Der nächste Schritt, der das Match-Modell weiterverarbeitet, ist der Diff-Builder. Dieser kann mithilfe des Match-Models einen statusbasierten 2-Wegs- und 3-Wegs-Vergleich durchführen und ein Diff-Modell erzeugen. Auch hier sieht EMF Compare vor, dieses Modul durch Ableitung einer Klasse erweiterbar zu halten. So kann beispielsweise für jede Sprache ein spezialisierter Vergleich durchgeführt werden. Sobald das Diff-Modell vorliegt, können sogenannte Diff-Extensions erneut Einfluss auf das resultierende Diff-Modell nehmen. Der Gedanke hinter diesem Mechanismus ist, dass oftmals eine Änderung auf Metaebene viele Änderungen auf Meta-Metaebene verursacht. Um den User nicht mit diesen abstrakten Details belasten zu müssen, kann eine Diff-Extension eingesetzt werden, die mehrere Diff-Elemente zusammenfasst. Letztlich wurde hierfür eine Möglichkeit geschaffen, eine Klasse in den Vergleichsprozess einzuhängen, die das Diff-Modell übergeben bekommt und es nach eigenen Vorstellungen verändern kann, also beispielsweise mehrere Diff-Elemente zusammen zu fassen. Es wird allerdings nur das Diff-Modell übergeben. Ein Zugriff auf das Match-Modell und damit die Modellversionen selbst ist derzeit nicht möglich. Diese Diff-Extension-Funktion ist allerdings auch noch in den Kinderschuhen. Nun kann dieses veränderte Diff-Modell im User-Interface visualisiert, über den Export exportiert und über den Merger ausgeführt werden. Die Abbildung 7.3 stellt einen Screenshot der EMF Compare Benutzerschnittstelle dar.

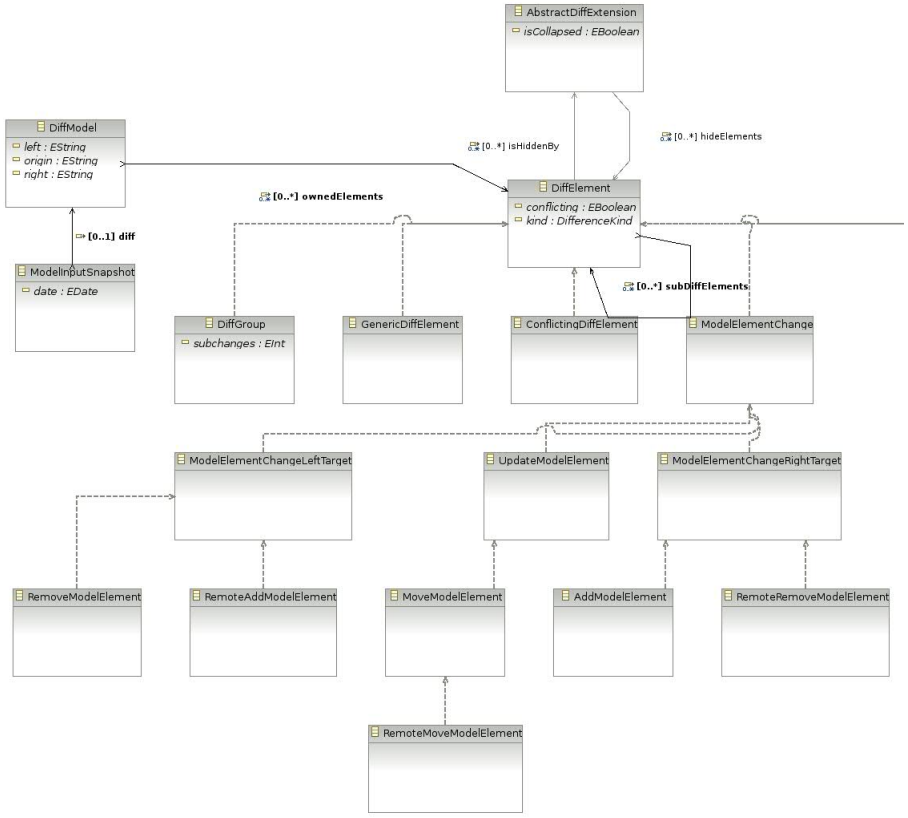

Abbildung 6.3: Ausschnitt aus dem EMF Compare Diff-Metamodell aus [7].

Abbildung 6.3 zeigt einen Ausschnitt aus dem Diff-Metamodell von EMF Compare. Das zentrale Element ist das *DiffElement*, das als Mutterklasse aller unterschiedlichen Differenzarten fungiert. Differenzen werden innerhalb eines Deltadokuments nach Elementen mithilfe der Klasse *DiffGroup* gruppiert, die die beinhaltenden Klassen über die Eigenschaft *subchanges* zusammenfasst. Auf der untersten Ebene befinden sich die konkreten Änderungstypen, wie beispielsweise *RemoveModelElement*, *MoveModelElement* und *AddModelElement*. Die Klassen *Remote** werden nur beim 3-Wegsvergleich instanziert und repräsentieren die jeweilige Änderung in der entfernten Version.

In Kapitel 7 wird im Listing 7.2 die Verwendung der EMF Compare Diff-API und in Listing 7.3 die Durchführung der Serialisierung eines Deltadokuments präsentiert.

7 Implementierung des Modellvergleichs

Dieses Kapitel beschreibt die im Zuge dieser Arbeit durchgeführte, prototypische Implementierung des Modellvergleichs. Diese Implementierung orientiert sich an den Anforderungen, die in Kapitel 5 besprochen wurden und findet auf Basis der in Kapitel 6 eingeführten Technologien statt.

7.1 Editorunabhängiges und sprachunabhängiges Model-Operation-Tracking

In Kapitel 5 wurde der Einsatz eines statusbasierten Vergleichs motiviert. Dennoch beschreibt dieser Abschnitt die Implementierung eines editor- und sprachunabhängigen Model-Operation-Trackings, das den operationsbasierten Modellvergleichen zugeordnet werden kann. Es soll hiermit einerseits gezeigt werden, wie ein editorunabhängiger, operationsbasierter Vergleich im Rahmen der Eclipse Plattform implementiert werden kann und andererseits, dass letztlich die Qualität der Deltas, die mit einem statusbasierten Vergleich ermittelt werden, die selbe ist, wie jene des operationsbasierten, generischen und editorunabhängigen Vergleichs.

Bevor nun die Implementierung im Detail besprochen wird, soll das allgemeine Konzept erläutert werden. In EMF ist es relativ einfach über Änderungen, die an einer Modellinstanz vorgenommen werden, informiert zu werden. Das EMF bildet hierfür ein hervorragendes Framework. Ecore-basierte Modelle in EMF implementieren alle die Schnittstelle *Resource* (`org.eclipse.emf.ecore.resource.Resource`) und sind damit auch *Notifier*. Daher ermöglichen sie auch die Registrierung eines *Adapter*s (`eAdapters()`). Ist ein Adapter bei einer Resource registriert, wird dieser Adapter über jede Änderung, die an der Resource vorgenommen wurde, in Form von *Notification*s informiert. Der Einfachheit halber kann man nun zur Implementierung des *Adapter*-Interfaces, das bei der Resource registriert wird, die Klasse *EContentAdapter*, eine fertige Implementierung des *Adapter*-Interfaces, ableiten und die Methode `notifyChanged(Notification notification)` überschreiben, um die Änderungen in Form von *Notification*s zu erhalten. Die übergebenen Notifications (`org.eclipse.emf.common.notify.Notification`) enthalten alle Informationen, die zur genauen Aufzeichnung der Änderungen nötig sind.

Änderungen werden jedoch nur dann an die Adapter weitergeleitet, wenn die Änderungen an genau jener Instanz einer Resource vorgenommen werden, an der die Adapter registriert sind. Da das Ziel die Implementierung eines editorunabhängigen Operationstrackings ist, muss unser Adapter, der die Änderungen mitschreibt, an genau jene

Instanz einer Resource gebunden werden, auf der ein beliebiger und unbekannter Editor arbeitet. Dieser Umstand erschwert die vorerst scheinbar einfache Aufgabe erheblich. Unsere Implementierung muss schließlich von jedem laufenden Editor die konkrete Instanz der Resource, auf der der Editor arbeitet, finden, um ihren Adapter anzubringen. Das erfordert eine genauere Betrachtung der Funktionsweise der Eclipse-Workbench und der in ihr laufenden Editoren. In Abbildung 7.1 ist daher auf der linken Seite der Aufbau hinsichtlich der Workbench und der Editoren skizziert.

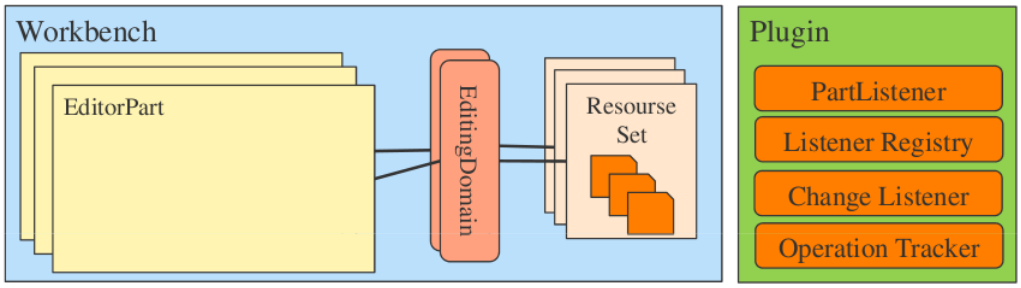

Abbildung 7.1: Überblick über die Architektur des Model-Operation-Tracker.

Eine Workbench (`IWorkbenchWindow`) beinhaltet eine beliebige Anzahl an *EditorParts*. Diese sind Teile eines Editors, die eventuell zur Bearbeitung einer Resource fähig sind. Um über ihre Existenz benachrichtigt zu werden, kann ein *EditorPartListener* an das PartService (`IPartService`) des *WorkbenchWindows* gebunden werden. Dieser *EditorPartListener* wird nun über die Erzeugung, das Schließen und den Fokuswechsel bezüglich dieses *EditorParts* informiert. Für diese Implementierung ist in diesem Zusammenhang jedoch nur interessant, ob es sich bei einem geöffneten oder aktiven *EditorPart* um einen Editor handelt, der Resourcen verändern kann. Wenn ein Editor eine Resource bearbeitet, hat er einen `IEditingDomainProvider` als Adapter registriert. Dieser `IEditingDomainProvider`, sofern existent, beinhaltet eine *EditingDomain*, die wiederum ein *ResourceSet* und damit mehrere Resourcen umfasst. Folglich kann bei jedem *EditorPart* auf diese Weise geprüft werden, ob diesem ein `IEditingDomainProvider` als Adapter zugewiesen wurde und damit ob und wenn ja, welche Resources dieser Editor bearbeiten kann. Diese Instanzen der Resourcen sind exakt jene, die der Editor verwendet, um sie zu manipulieren. Das bedeutet, dass ein Adapter an genau diesen Instanzen über die Änderungen, die dieser Editor vornimmt, informiert wird.

Das editorunabhängigen Operationstrackings ist als Eclipse Plug-In realisiert und läuft nur in Verbindung mit der Java Runtime Environment 1.6 und Eclipse 3.3. Außerdem ist diese Implementierung abhängig von insgesamt sechs anderen Eclipse-Plug-Ins. Diese sind in der folgenden Liste angeführt.

- *org.eclipse.emf*: Beinhaltet das Eclipse Modeling Framework und damit die Schnittstellen und Implementierungen von Ecore.
- *org.eclipse.emf.ecore.xmi*: Stellt Funktionalitäten für das Serialisieren und Deserialisieren von Ecore-basierten Modellen in XMI zur Verfügung.
- *org.eclipse.emf.compare*: Dieses Plug-In wird benötigt, um die EMF Compare API verwenden zu können.
- *org.eclipse.emf.compare.diff*: Innerhalb dieses Plug-Ins befindet sich das Diff-Metamodell von EMF Compare.

- *org.eclipse.emf.compare.match*: Neben dem Diff-Metamodell wird auch das Match-Metamodell von EMF Compare benötigt, dessen Implementierung teil dieses Plug-Ins ist.
- *org.eclipse.core.filesystem*: Dieses Plug-In enthält Funktionen zur Erstellung und Bearbeitung von Dateipfaden sowie zum Umgang mit dem Dateisystem.

Das hier vorgestellte Plug-In selbst umfasst vier Komponenten, die in Abbildung 7.1 im rechten Bereich dargestellt sind. Es erweitert die Eclipse-Umgebung um einen *PartListener* sowie ein Popup-Menü über die Extension `org.eclipse.ui.popupMenus`. Dieses Kontextmenü erscheint nach einem Rechtsklick auf eine Datei und bietet die Möglichkeit, die Änderungen der ausgewählte Datei zu überwachen (siehe Abbildung 7.2).

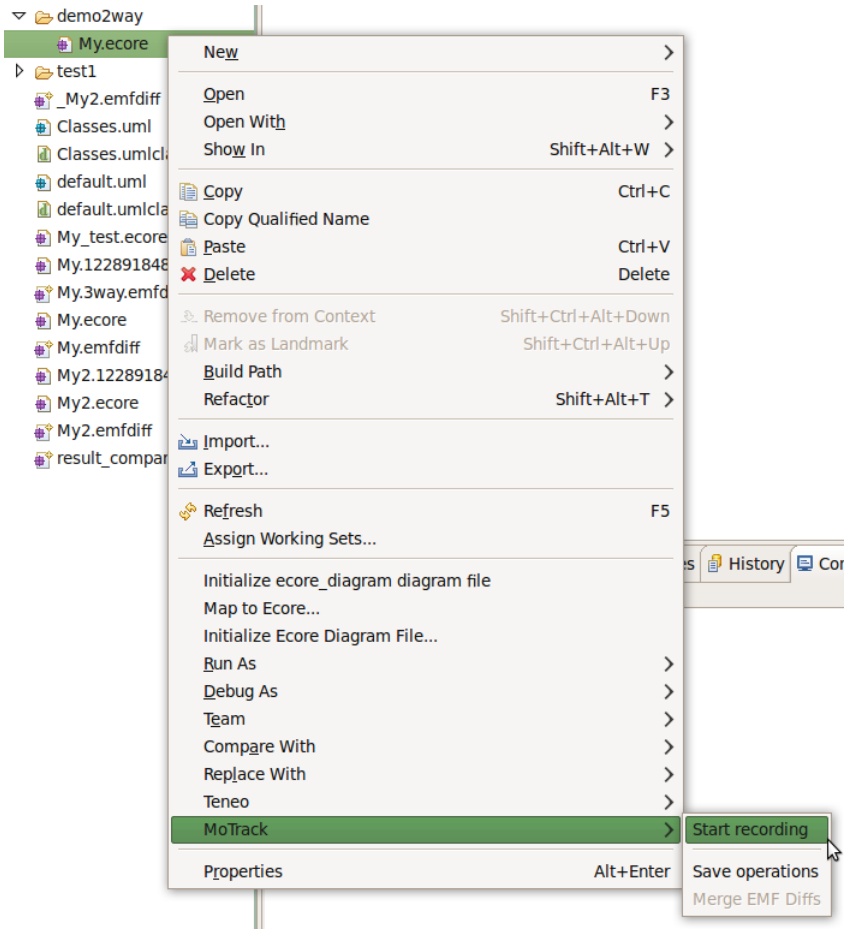

Abbildung 7.2: Kontext-Menü des implementierten Plug-Ins.

Wird im Kontextmenü *Start recording* gewählt, prüft das Plug-In, ob es sich bei der ausgewählten Datei um ein Ecore-basiertes Modell handelt und legt in diesem Fall eine Kopie der ausgewählten Datei an. Die kopierte Datei simuliert innerhalb dieses Prototyps nun die Version des Repositories und die ausgewählte Datei stellt von nun an die Arbeitskopie dar. Nachdem die Kopie erstellt wurde, wird für die Arbeitskopie ein *ChangeListener* (Klasse `at.ac.tuwien.big.motrack.listener.impl.ModelOperationListenerImpl`) erstellt und in der *Listener Registry* registriert. Die *Listener Registry* (Klasse `at.ac.tuwien.big.motrack.listener.impl.ModelOperationListenerRegistryImpl`) ist gleichzeitig auch der *PartListener*, der über die Aktionen der *EditorParts*

informiert wird. Dieser prüft, ob es sich bei jedem *EditorPart* um einen Editor handelt, der ein zu überwachendes Modell bearbeitet. Ist dies der Fall, registriert sich der *ChangeListener* als Adapter bei der jeweiligen Resource-Instanz dieses Editors, um in weiterer Folge über Änderungen, die dieser Editor durchführt, benachrichtigt zu werden. Dieser *ChangeListener* schreibt diese Änderungen jedoch nicht selbst mit, sondern dient nur als Sammelstelle aller eintreffenden *Notification*s und leitet diese an den *Operation Tracker* (Klasse `at.ac.tuwien.big.motrack.diffbuilder.impl.GenericDiffModelBuilder`) weiter. Dieser könnte nun die empfangenen *Notification*s direkt filtern, interpretieren und in ein EMF Compare Diff-Modell übersetzen. In einer frühen Version der Implementierung war dies auch der Fall. Jedoch musste bei der direkten Übersetzung der *Notification*s manuell berücksichtigt werden, ob sich Änderungen im Laufe der Zeit gegenseitig wieder obsolet machen. Dies war beispielsweise der Fall, wenn eine Klasse verändert wurde, die später gelöscht wurde. Um sich dies zu ersparen, kann auch der in EMF enthaltene *ChangeRecorder* (`org.eclipse.emf.ecore.change.util.ChangeRecorder`) verwendet werden, der diese Arbeit übernimmt. Dieser erzeugt eine sogenannte *ChangeDescription*(`org.eclipse.emf.ecore.change.ChangeDescription`), die nunmehr keine überflüssigen Operationen mehr enthält. Dementsprechend muss diese *ChangeDescription* nur mehr in ein EMF Compare Diff-Modell übersetzt werden. Da das EMF Compare Diff-Modell jedoch komplexere und daher aussagekräftigere Klassen und Eigenschaften enthält als dies bei der *ChangeDescription* der Fall ist, muss diese im Zuge dieser Übersetzung etwas genauer interpretiert werden. Diese Übersetzung geschieht, nachdem der User *Save operations* im Kontextmenüs des Plug-Ins (siehe Abbildung 7.2) gedrückt hat.

Um ein vollständiges EMF Compare Diff-Modell erzeugen zu können, das auch innerhalb des grafischen Editors von EMF Compare dargestellt werden kann, muss auch ein EMF Compare Match-Modell generiert werden. Da zum Zeitpunkt der Erstellung der Kopie (siehe weiter oben) die Modelle 100 %ig ident waren, konnte eine `java.util.Map` erzeugt werden, die alle äquivalenten Elemente aus der Arbeitskopie mit ihren Ursprungselementen der Sicherungskopie verknüpft. Auf Basis dieser Map kann nun ein exaktes EMF Compare Match-Modell erzeugt werden.

In Abbildung 7.3 ist ein Beispielszenario abgebildet, anhand dessen die Funktionsweise dieses Plug-Ins demonstriert werden soll. In der Ausgangssituation (linker Bereich) sind zwei Klassen, die Klasse *Person* und die Klasse *Address*, abgebildet.

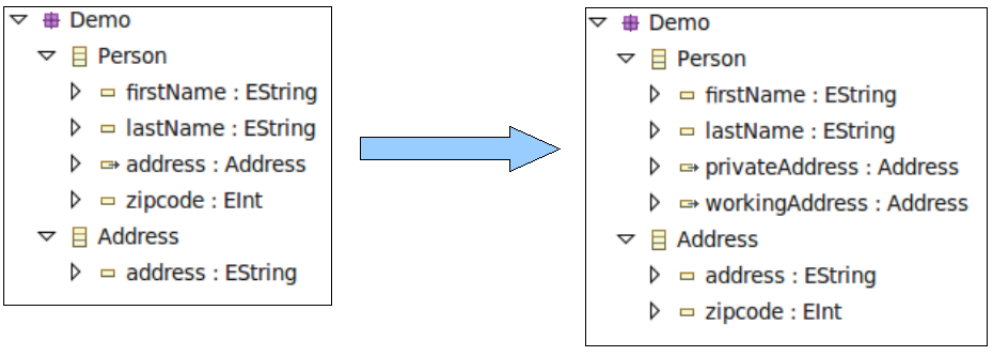

Abbildung 7.3: Demonstration der Implementierung: Modelle.

Der User aktiviert nun durch Auswählen von *Start recording* im Kontextmenü die Aufzeichnung der Änderungen, wodurch, wie zuvor beschrieben, eine Sicherungskopie

(Repository-Version) angelegt und ein *Change Listener* für diese Datei in der *Listener Registry* angelegt wird. Nun führt der User in einem Editor seiner Wahl einige Änderungen durch. Indem der User in irgendeiner Form den Editor verwendet, erhält dieser den Fokus und aktiviert damit den *PartListener* und damit ebenfalls die *Listener Registry*, die feststellt, dass dieser Editor eine zu überwachende Resource bearbeitet. Die *Listener Registry* fügt nun der konkreten Instanz der Resource den *Change Listener* hinzu, der die Änderungen an den *OperationTracker* weiterleitet. In dem Beispiel verschiebt der User nun das Attribut *zipcode* in die Klasse *Address*, benennt die Referenz *address* in *privateAddress* um und fügt eine weitere Referenz auf die Klasse *Address* mit dem Namen *workingAddress* hinzu. All diese Änderungen wurden nun vom *ChangeRecorder* aufgezeichnet und gespeichert. Klickt der User nun auf *Save operations*, wird über den *ChangeRecorder* eine *ChangeDescription* erzeugt und in ein EMF Compare Diff-Modell übersetzt. Das resultierende Diff-Modell ist in Abbildung 7.4 abgebildet. Dort wurden alle Änderungen richtig erkannt. Durch die operationsbasierte Methode bzw. den daraus resultierenden Match konnte auch erkannt werden, welche der beiden Referenzen umbenannt wurde (*privateAddress*) und welche erzeugt wurde (*workingAddress*). Dies wäre bei einem imperfekten Match durch Heuristiken nicht möglich.

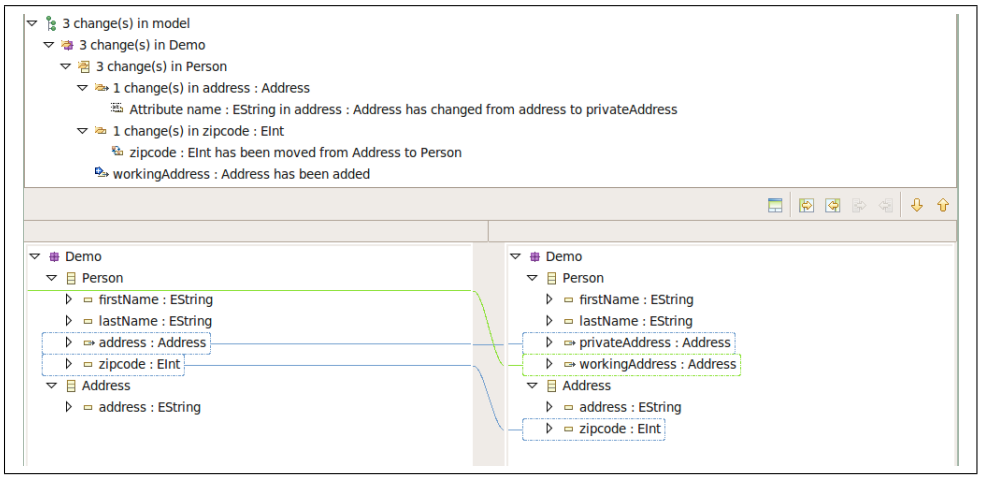

Abbildung 7.4: Demonstration der Implementierung: Differenzen.

Die in diesem Abschnitt vorgestellte Implementierung hat die Möglichkeit aufgezeigt, ein editor- und sprachunabhängiges Tracking von Modelloperationen in Eclipse zu realisieren. Dennoch ist die Eigenschaft der Editorunabhängigkeit nicht vollständig gegeben. Schließlich muss der Editor erstens Eclipse-basiert sein und zusätzlich noch in der selben Eclipse-Instanz verwendet werden, wo auch das Tracking stattfindet. Sonst kann nicht auf die konkrete Instanz der Resource zugegriffen und diese abgehört werden, auf der der Editor seine Änderungen durchführt. Das wäre nur möglich, wenn vollständig statusbasiert vorgegangen wird. Aus diesem und all den in Kapitel 5 genannten Gründen, die die Verwendung des statusbasierten Technik motivieren, wird im folgenden Abschnitt eine mögliche Realisierung des statusbasierten Modellvergleichs innerhalb der Eclipse Plattform vorgestellt.

7.2 Statusbasierter Vergleich

In diesem Abschnitt folgt die Beschreibung der Implementierung des statusbasierten Vergleichs auf Basis von EMF und EMF Compare. Die Vorzüge des statusbasierten Vergleichs im Gegensatz zum operationsbasierten Vergleich wurden bereits in Kapitel 5 ausführlich diskutiert. Im vorigen Abschnitt wurde gezeigt, dass der operationsbasierte Vergleich zwar in gewisserweise editorunabhängig implementiert werden kann, dass aber dennoch eine starke Abhängigkeit an die Plattform des Clients unumgänglich ist. Trotz dieser Abhängigkeit können zusammengesetzte Operationen nicht direkt erkannt werden und müssten daher, wie beim statusbasierten Vergleich, im Anschluss aus dem Deltadokument abgeleitet werden. Wenn daher der operationsbasierte Vergleich rein generisch und, soweit möglich, editorunabhängig realisiert ist, kann ein statusbasierter Vergleich, sofern ein perfekter Match durchführbar ist, ein Differenzmodell der selben Qualität wie der operationsbasierte Vergleich erzielen. Natürlich kann durch einen statusbasierten Vergleich nicht der konkrete Weg mit den exakten Operationen abgeleitet werden. Es kann beispielsweise nicht erkannt werden, ob der Name einer Klasse innerhalb einer Session mehrmals geändert wird. Für die Versionierung ist das jedoch auch nicht relevant. Es zählt letztlich nur das Resultat aller Änderungen. Ebenso ist die Chronologie der Änderungen im Nachhinein nicht sichtbar, es sei denn der Editor annotiert die Reihenfolge oder den Zeitpunkt der Änderungen. Die Chronologie ist jedoch nach bisherigen Überlegungen ebenfalls kein entscheidender Grund, auf die überwiegenden Vorzüge des statusbasierten Vergleichs zu verzichten.

Wie gesagt ist die Voraussetzung einer zuverlässigen Implementierung des statusbasierten Vergleichs die Durchführung eines perfekten Matchs. Wie bereits im Kapitel 5 besprochen, ist die vom Editor selbst vorgenommene Zuordnung von IDs zu den Elementen keine wünschenswerte Lösung. Das Problem dieser Technik ist in erster Linie, dass das Versionierungssystem von der Vergabe der IDs durch den Editor in hohem Maße abhängig ist. Wenn dieser keine IDs vergibt, kann das System keinen zuverlässigen Match durchführen. Die zuverlässige Funktionsweise des Versionierungssystems kann also nicht gewährleistet werden, wenn ein Editor verwendet wird, der diese Anforderung nicht erfüllt. Ein weiteres Problem ist auch, dass, sofern der Editor überhaupt IDs vergibt, diese genau an der Stelle gespeichert werden müsste, wo dies vom Versionierungssystem erwartet werden würde. Da dies vermutlich nicht bei allen Editoren der Fall ist, müsste das Versionierungssystem stets für alle möglichen Orte, wo Editoren ihre IDs platzieren, konfiguriert werden. Es müsste so aber im Vorhinein bekannt sein, welche Editoren zum Einsatz kommen. Um dieses Probleme dauerhaft zu lösen, muss die Verantwortung für die ID-Vergabe bei jener Komponente liegen, die die IDs auch tatsächlich nutzt – und das ist in diesem Fall das Versionierungssystem selbst. Nachdem der einzige Zeitpunkt innerhalb einer Änderungssession, in dem die Arbeits- und die Repositorykopie exakt identisch sind, jener des Checkouts ist, kann das Versionierungssystem zu diesem Zeitpunkt noch selbstständig den vollständigen 1:1-Match durchführen. Die daraus erfolgten Äquivalenzen können über *transiente* IDs in der Arbeitskopie auch über die Änderungen des Users hinweg beibehalten. Diese IDs sind deshalb *transient*, da sie nur während einer Änderungssession, also dem Zeitraum zwischen dem Checkout und dem Commit, gültig sein müssen. Die einzige Anforderung an den Editoren ist nunmehr, dass diese IDs nicht entfernt oder geändert werden dürfen. Neuen Elementen muss keine ID vergeben werden, da diese Elemente ohnehin nicht gematcht werden müssen.

Aufgrund der Einfachheit der Realisierung des ID-basierten Matchs wird dieser hier nicht näher erläutert. In diesem Zusammenhang ist eher interessant, wie diese IDs den Elementen zugeordnet und während der Änderungssession beibehalten werden können. Dies kann beispielsweise, wenn XMI[1] als Modellaustauschformat verwendet wird, über das Attribut `id` innerhalb des XMI-Namensraums (z.B. `http://www.omg.org/XMI`) bewerkstelligt werden. Natürlich kann sein, dass ein Editor dieses Attribut verwendet. Aus diesem Grund kann auch, wenn diese Eventualität berücksichtigt werden soll, ein eigens definiertes Attribut in einem eigenen Namensraum verwendet werden.

Wenn der Match auf Basis dieser IDs durchgeführt werden kann, ist die Erzeugung des EMF Compare Match-Modell eine einfache Angelegenheit. Das Match-Model besteht aus sogenannten `Match2Elements`-Objekten, die jeweils zwei äquivaliente Elemente miteinander verknüpfen. Zur Veranschaulichung ist eine in XMI serialisierte Instanz des EMF Compare Match-Meta-Modells in Listing 7.1 dargestellt.

Listing 7.1: Auszug eines EMF Compare Match-Model in XMI.

```
...
<matchedElements xsi:type="match:Match2Elements" similarity="
    1.0">
  <subMatchElements xsi:type="match:Match2Elements" similarity
      ="1.0">
    <subMatchElements xsi:type="match:Match2Elements"
        similarity="1.0">
      <leftElement
        href="platform:/resource/Origin.ecore#//Testclass/
            testOp"/>
      <rightElement
        href="platform:/resource/Working.ecore#//Testclass/
            testOp"/>
    </subMatchElements>
  </subMatchElements>
  ...
</matchedElements>
...
```

Da dieses Match-Model mithilfe von IDs erzeugt wurde, ist die Gleichheit (Attribut `similarity`) immer `1` und damit vollständig. Die gematchten Elemente werden über das Attribut `href` referenziert. In diesem Fall werden Plattform-URIs verwendet. Es ist aber jede von EMF unterstützte Referenz möglich. Beispielsweise können referenzierte Elemente auch in verteilten Modellen (z.B. im Repository) liegen. Soll nun auf Basis dieses perfekten Matchs ein Vergleich durchgeführt werden, kann dieses Match-Model an die EMF Compare API übergeben und ein Diff-Model errechnet werden. Der EMF Compare Editor benötigt zur Darstellung von Differenzen sowohl das Diff-Model, als auch das Match-Model. Diese beiden Modelle werden in der Klasse *ModelInputSnapshot* (`org.eclipse.emf.compare.diff.metamodel.ModelInputSnapshot`) zusammengefasst. Wie eine Instanz dieses *ModelInputSnapshot*s auf Basis eines eigens erstellten Matchs erzeugt wird, ist im Listing 7.2 skizziert. Statt der

[1] XMI – XML Metadata Interchange

`GenericDiffEngine` kann natürlich auf Wunsch auch eine selbst implementierte Ableitung der *DiffEngine* verwendet werden. Der erzeugte *ModelInputSnapshot* kann nun natürlich wie ein gewöhnliches EMF-Modell in XMI exportiert und im EMF Compare Editor dargestellt werden, um eine grafische Visualisierung, ähnlich der in Abbildung 7.4, zu erzielen. Die Durchführung der Serialisierung des zuvor generierten *ModelInputSnapshot* zeigt das Listing 7.3.

Listing 7.2: Erzeugung eines Diff-Models mit EMF Compare.

```
// die eigens erstellte Match-Instanz
matchModel = ...

// der ModelInputSnapshot
modelInputSnapshot = DiffFactory.eINSTANCE
                                .createModelInputSnapshot();

// zu verwendende DiffEngine
IDiffEngine diffEngine = new GenericDiffEngine();
modelInputSnapshot.setDiff(diffEngine.doDiff(matchModel));
modelInputSnapshot.setMatch(matchModel);
```

Listing 7.3: XMI-Serialisierung eines EMF Compare Diffs.

```
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet
      .getResourceFactoryRegistry()
      .getExtensionToFactoryMap()
      .put(".emfdiff",
        new org.eclipse.emf.ecore.xmi.impl.
            EcoreResourceFactoryImpl());
Resource emf_diffresource = resourceSet
      .createResource(URI.createPlatformResourceURI(
        "My.emfdiff", true));
emf_diffresource.getContents().add(modelInputSnapshot);
emf_diffresource.save(null);
```

Im Vergleich zu der operationsbasierten Methode des vorigen Abschnitts kann die gesamte Ausführung des Matchs, des Vergleichs und der weiteren Verarbeitung des Vergleichs nun zentral, beispielsweise auf dem Repositoryserver, durchgeführt und über WebService-Schnittstellen gesteuert werden. Diese Zentralisierung der Logik bringt erhebliche Erleichterungen bezüglich der Wartung, Konfiguration und sprachspezifischen Erweiterung des Gesamtsystems mit sich. Eine detaillierte Diskussion dieses Vorzuges und der restlichen Vorteile der statusbasierten Technik, in Gegenüberstellung zum operationsbasierten Vergleich, ist in Kapitel 5 gegeben.

8 Erkennung zusammengesetzter Operationen

In Kapitel 5 wurde die Notwendigkeit für die Erkennung sprachspezifischer, zusammengesetzter Operationen im Rahmen des Modellvergleichs bereits erläutert. Diese Notwendigkeit begründet sich vor allem durch die Vereinfachung der Komplexität des Differenzmodells für den User durch die Zusammenfassung atomarer, abstrakter Operationen. Außerdem steht hierbei die Erkennung von Refactorings im Vordergrund, die es erlaubt, unnötige Konfliktmeldungen (siehe Abschnitt 3.4) zu vermeiden oder zumindest intelligente Resolutionsvorschläge auf Basis dieser zusätzlichen Information zu ermöglichen. Da zusammengesetzte Operationen sprachspezifisch sind, müssen diese vom User durch die Bereitstellung von deskriptiven Definitionen in das System eingebunden werden. In diesem Kapitel wird daher ein benutzerfreundlicher Ansatz vorgestellt, der eine einfache beispielgetriebene Definition von zusammengesetzten Operationen und Refactorings erlaubt.

Die Anforderungen an eine Operationsdefinitionsmöglichkeit wurde in Kapitel 5 bereits angeschnitten. Dort wurde vor allem betont, dass die Definition ausschließlich durch deskriptive Artefakte, also ohne jeglichen Programmieraufwand, möglich sein soll. Es sollten daher vom User auch nur Technologien eingesetzt werden müssen, die der Zielgruppe üblicherweise geläufig sind. Da es sich bei der Zielgruppe in diesem Fall um Experten im Modellierungsbereich handelt, sind diese Technologien einerseits die Modellierung selbst und zumeist auch OCL[1] [35]. Neben der Voraussetzung bezüglich der Technologien können auch noch weitere wünschenswerte Eigenschaften dieses Features identifiziert werden. Dies ist zu Beginn vor allem die Vollständigkeit. Die Definitionsmethode darf, soweit möglich, den User nicht in seinen Definitionsmöglichkeiten einschränken. Das bedeutet, dass alle möglichen Operationszusammensetzungen und Refactorings abbildbar sein sollten. Außerdem ist es wichtig, dass, nachdem eine zusammengesetzte Operation erkannt wurde, atomare Operationen, die durch die definierte Operation zusammengefasst werden, erhalten bleiben. Nur so kann weiterhin eine funktionierende, generische Konflikterkennung und -resolution, die ja ausschließlich auf den generischen, zuvor erzeugten Differenzen arbeitet, gewährleistet werden. Ansonsten müsste, sobald eine neue Operation konfiguriert wurde, alle mit den Differenzen arbeitende Algorithmen ebenfalls erweitert und dahingehend angepasst werden. Fasst eine konfigurierte Operation die generischen, atomaren Operationen nur zusammen und stellt diese weiterhin zur Verarbeitung bereit, können die anschließenden Mechanismen ohne Anpassung, wie zuvor, damit arbeiten.

Wie bereits erwähnt, ist es wichtig, dass kein programmatischer Eingriff oder eine komplexe Konfiguration zur Definition zusammengesetzter Operationen verlangt werden darf. Es sollen nur Technologien verwendet werden, die einer Expertin oder einem Experten im Bereich der Modellierung geläufig sind. Daher verwendet der im folgenden

[1] OCL – Object Constraint Language

vorgestellte Ansatz eine beispielgetriebenen Methode, die dem User erlaubt, die Operation beispielhaft in der eigentlichen Modellierungssprache darzustellen und sie mit OCL-Constraints zu verfeinern. Der hier verfolgte Ansatz ist daher dem Bereich *Programming by example* [15] zuzuordnen.

8.1 Verwendung der Benutzeroberfläche

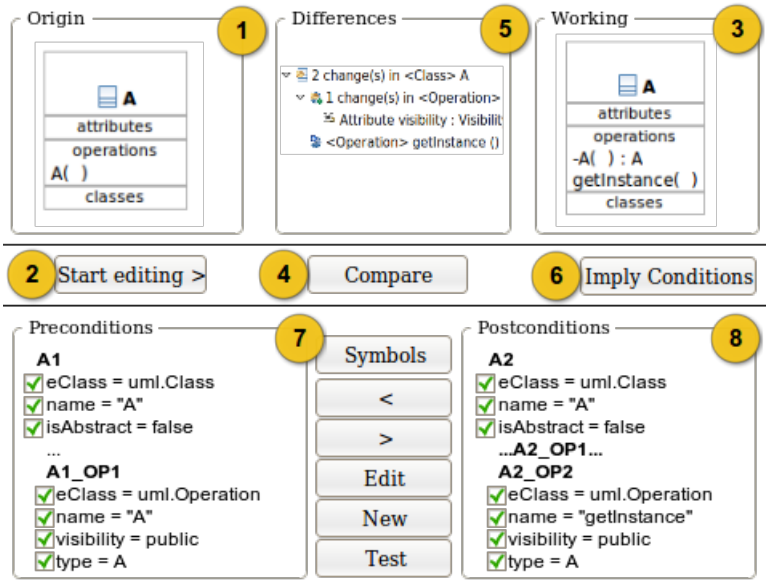

Abbildung 8.1: User Interface 1 der beispielgetriebenen Operationsdefinition.

Der einfachste Weg, diese Definitionsmethode zu erläutern, ist es, die Funktionsweise anhand eines Beispiels zu präsentieren. Abbildung 8.1 stellt den prototypischen Aufbau der graphischen Benutzeroberfläche dar, dessen Verwendung nun anhand der in der Abbildung dargestellten Nummern Schritt für Schritt erklärt wird. Für das folgende Beispiel wird die Operation *Convert to Singleton* verwendet. Diese Operation ändert eine Klasse, sodass nur noch eine Instanz dieser Klasse pro Laufzeitumgebung existieren kann. Es wird hierfür der Konstruktor der Klasse von der Verwendung anderer Klassen ausgeschlossen und daher auf eine private Sichtbarkeit geändert. Um eine Instanz dieser Klasse zu bekommen, müssen die *Clients* dieser Klasse die statische Methode *getInstance* verwenden, die im Zuge dieser Operation hinzugefügt wird. Diese Methode gibt die einzig existierende Instanz dieser Klasse zurück. Sie hat damit den Rückgabetyp der Klasse selbst.

Schritt 1: Ausgangssituation beispielhaft modellieren.

Im ersten Schritt muss die Ausgangssituation der zusammengesetzten Operation beispielhaft modelliert werden. Dieses Modell muss alle Aspekte enthalten, die für die zu definierende Operation von Bedeutung sind. Dies geschieht im Bereich *Origin* der Benutzerschnittstelle. Hierbei kann der selbe Editor eingesetzt werden, der auch in der täglichen Arbeit des Users genutzt wird.

Schritt 2: Button *Start editing* drücken.

Nachdem alle nötigen Aspekte in der Ausgangssituation dargestellt sind, muss nun der Button *Start editing* gedrückt werden. Diese Aktion erzeugt eine Kopie des Ausgangsmodells und stellt diese im Bereich *Working* der Benutzerschnittstelle dar. Zu diesem Zeitpunkt ist das Modell im Bereich *Origin* und im Bereich *Working* exakt identisch. Somit kann ein 1:1-Match dieser beiden Modelle durchgeführt werden. Dieser ist demzufolge vollständig und perfekt. Der Match, der alle äquivalenten Modellelemente referenziert, kann nun im Speicher gehalten werden, um die Modellelemente auch nach starker Veränderung wiedererkennen zu können.

Schritt 3: Zusammengesetzte Operation auf Arbeitskopie durchführen.

Nun kann der User alle Änderungen im Bereich *Working* vornehmen, die von der zusammengesetzten Operation zusammengefasst werden sollen. In unserem Beispiel sind das die weiter oben bereits beschriebenen Operationen.

Schritt 4: Button *Compare* drücken.

Wurden alle Änderungen der zusammengesetzten Operation durchgeführt, müssen diese vom User durch Betätigung des Buttons *Compare* bestätigt werden. Dies löst einen generischen, statusbasierten Vergleich des Modells in *Origin* mit jenem in *Working* auf Basis des zuvor erzeugten, perfekten Matchs (Schritt 2) aus.

Schritt 5: Durchgeführte Operationen werden dargestellt.

Die im letzten Schritt erkannten, atomaren und generischen Operationen werden im Bereich *Differences* dargestellt. In Abbildung 8.2 sind diese Differenzen zusätzlich in einer lesbaren Form abgebildet. In unserem Beispiel besteht der Unterschied zwischen den beiden Modellen aus insgesamt zwei Änderungen. Die erste Änderung bezieht sich auf die Sichtbarkeit des Konstruktors der Klasse *A*. Diese wurde im Sinne der zusammengesetzten Operation auf *private* geändert. In der zweiten Änderung wurde die Methode *getInstance* hinzugefügt.

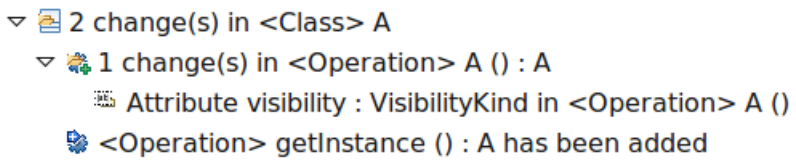

Abbildung 8.2: Ermittelte Differenzen der zusammengesetzten Operation.

Schritt 6: Button *Imply Conditions* drücken.

Nachdem nun die Operationen durchgeführt und ermittelt wurden, können aus dem Ausgangsmodell und dem veränderten Modell alle Vor- und Nachbedingungen für die zu definierende, zusammengesetzte Operation abgeleitet werden. Zuvor muss allerdings ein Kontextelement für die Formulierung der Bedingungen ermittelt oder ausgewählt werden. Sofern sich nur ein Modellelement als Wurzelelement in der Ausgangssituation befindet, wie dies in unserem Beispiel der Fall ist, kann dieses Element automatisch als Kontextelement selektiert werden. Falls jedoch mehrere Elemente auf der ersten Ebene des Modell existieren, muss eines vom User ausgewählt werden. Alle Bedingungen werden nun relativ zu diesem Element formuliert. Zuerst wird noch für jedes Modellelement ein Symbol oder,

anders ausgedrückt, eine Variable angelegt. Dieses Symbol dient in der momentanen Phase der Bedingungsformulierung als Kapselung eines Elements. Bei der Erkennung der definierten Operation wird es außerdem verwendet, um ein Element, das die Bedingung dieses Symbols erfüllt, der Rolle zuzuordnen, die das Beispielelement im Operationsszenario einnimmt. Eine Rolle in unserem Szenario wäre beispielsweise ein Konstruktor, dessen Sichtbarkeit auf *private* geändert wurde. Die Erzeugung der Symbole kann über die OCL *def*-Funktion vorgenommen werden. Auf die Realisierung wird weiter unten genauer eingegangen. Sind schließlich alle Modellelemente durch Symbole identifiziert worden, kann für jedes Symbol (oder Element) und jede generische Eigenschaft des Elements eine Bedingung erzeugt werden. All diese Bedingungen treffen zu diesem Zeitpunkt exakt und vermutlich ausschließlich nur auf das aktuelle Ausgangsmodell bzw. das veränderte Modell zu. Diese abgeleiteten Bedingungen sind in Abbildung 8.1 im Bereich *Preconditions* bzw. *Postconditions* dargestellt. Dort werden die Symbole baumartig, nach ihrer Zugehörigkeit zu anderen Symbolen oder Elementen, dargestellt. Beispielsweise repräsentiert das Symbol *A1* die Klasse *A* im Ausgangsmodell und *A1_OP1* den Konstruktor von *A1*. Die zugehörigen Bedingungen sind unter den jeweiligen Symbolen aufgeführt. Zum Beispiel ist dort vermerkt, dass der Konstruktor *A1_OP1* bei der Eigenschaft *eClass* den Wert *uml.Operation* aufweisen muss.

Schritt 7: Vorbedingungen (*Preconditions*) bearbeiten.

In den meisten Fällen werden die im vorigen Schritt implizierten Vorbedingungen zu straff sein, als dass sie sich bereits als Vorbedingung für die Operation eignen. Dennoch wird meist ein Großteil dieser Bedingungen für die Operationsdefinition zutreffend sein. Der User muss demnach in diesem Schritt die Vorbedingungen soweit auflockern bzw. anpassen, dass diese die Vorbedingung zur Ausführung der definierten Operation exakt abbilden. In unserem Beispiel sind an den abgeleiteten Bedingungen insgesamt drei Änderungen vorzunehmen. Die erste notwendige Änderung ist die Deaktivierung der Bedingung, die den Namen der Klasse A definiert ($A1.name = \,'A'$). Schließlich ist der Name der Klasse nicht Voraussetzung für die Anwendung dieser Operation. Aus diesem Grund muss einfach die Checkbox vor dieser Bedingung deaktiviert werden. Außerdem ist die Bedingung ($A1\_OP1.name = \,'A'$) keine gültige Vorbedingung. Der Konstruktor muss nicht immer den Namen $'A'$ haben, sondern den Namen der beinhaltenden Klasse aufweisen. Daher muss diese Bedingung in $A1\_OP1.name = A1.name$ abgeändert werden. Die letzte Änderung dreht sich um den Rückgabewert des Konstruktors. Dieser muss nicht den Typ A aufweisen, sondern ebenfalls den Typ der beinhaltenden Klasse. Daher muss auch diese Bedingung auf $A1\_OP1 = A1$ berichtigt werden. Alle veränderten Bedingungen sind in Abbildung 8.3 in kursiver Schrift dargestellt.

Schritt 8: Nachbedingungen (*Postconditions*) bearbeiten.

Der nächste und letzte Schritt einer Operationsdefinition ähnelt dem vorherigen Schritt, nur dass nicht die Vorbedingungen, sondern die Nachbedingungen angepasst werden. Dies geschieht auf die selbe Art und Weise wie bei den Vorbedingungen. In unserem Beispiel betrifft das vier Eigenschaften. Aus Platzgründen sind jedoch nur zwei dieser Änderungen in Abbildung 8.3 abgebildet. Konkret wurde aus dem selben Grund wie bei den Vorbedingungen die Eigenschaft $A2.name = \,'A'$ verallgemeinert. Außerdem müssen die selben Eigenschaften, die bei $A1\_OP1$ verändert wurden (*name* und *type*) auch bei dem gegenüber-

liegenden Äquivalent $A2\_OP1$ angepasst werden. Die Eigenschaft *visibility* hat bei diesem Konstruktor jedoch den Wert *private*. Dies konnte jedoch bereits aus dem *Working*-Modell abgeleitet werden. Bei den Bedingungen der hinzugefügten *getInstance*-Methode muss schließlich der Rückgabewert verallgemeinert werden. Der Constraint, der den Namen mit *getInstance* festlegt, muss klarerweise beibehalten werden, da in unserem Beispiel nur dieser Name als Singleton-Methode verwendet wird.

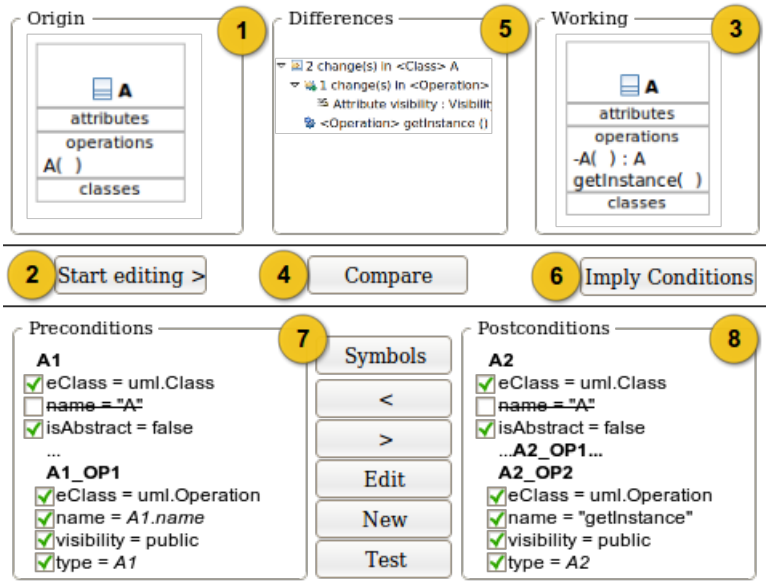

Abbildung 8.3: User Interface 2 der beispielgetriebenen Operationsdefinition.

In Abbildung 8.1 bzw. 8.3 sind zwischen dem Bereich der Vor- und Nachbedingungen einige Buttons abgebildet, deren Funktion in der folgenden Liste erläutert wird.

Symbols: Der *Symbols*-Button öffnet ein Fenster, in dem der User die Symbolnamen nach den eigenen Vorstellungen umbenennen und somit der Rolle, die ein Symbol innerhalb dieses Operationsszenarios repräsentiert, einen aussagekräftigeren Namen geben kann. Außerdem ermöglicht dieses Fenster die Definition weiterer benutzerdefnierter Symbole.

< bzw. >: Diese beiden Buttons können verwendet werden, um eine veränderte Regel der Vor- bzw. Nachbedingungen in die jeweilige andere Bedingungsliste zu übertragen. Da alle Modellelemente und damit auch die Symbole mit ihrem jeweiligen gegenüber gematcht wurden, können auch die Regeln einer Seite der gegenüberliegenden Regel (Vor- oder Nachbedingung) zugeordnet werden. Wenn daher einige Regeln deaktiviert wurden (z.B. die Bedingung des Namens $A1.name = 'A'$) kann diese Deaktivierung mit einem Klick auf den jeweilig anderen Bereich übertragen werden. Diese Buttons dienen daher ausschließlich dem Benutzerkomfort und haben keinen direkten implementierungstechnischen Einfluss.

Edit: Der *Edit*-Button dient zur Editierung der aktuell ausgewählten Regel. Es kann hierfür ein Editor geöffnet werden, der Code-Highlighting und -Completion für die Bedingungssprache, in unserem Fall OCL, anbietet. Hilfreich wäre an dieser Stelle auch die Möglichkeit, die aktuelle Regel gegen ein Szenario zu testen.

New: Mithilfe des *New*-Buttons können weitere, benutzerdefinierte Regeln hinzugefügt werden, die nicht automatisch abgeleitet werden konnten. Das ist für einige Fälle, insbesondere bei der Definition komplexerer Refactorings, von großer Bedeutung. Für die Erstellung benutzerdefinierter Regeln gilt das selbe, wie für das Editieren bestehender. Sie müssen im Kontext des Kontextelements aller anderen Regeln formuliert werden. Dies kann, wie beim *Edit*-Button, durch einen Editor mit Code-Highlighting und -Completion erleichtert werden.

Test: Mit diesem Button können alle Regeln im Bereich der Vor- bzw. Nachbedingungen gegen das aktuelle Modell im *Origin*- bzw. *Working*-Bereich getestet werden. Das bedeutet, dass geprüft werden kann, ob die Regeln im Bezug auf das jeweilige Modell erfüllt sind. Ist dies nicht der Fall, liegt offensichtlich ein Fehler in der Regeldefinition vor.

8.2 Realisierung

Nachdem die Anwendung dieses beispielgetriebenen Ansatzes erläutert wurde, soll nun auf die Realisierung etwas detaillierter eingegangen werden. Die Implementierung findet auf Basis des Eclipse Modeling Frameworks (EMF) und EMF Compare statt. In den Bereichen *Origin* und *Working* der Benutzeroberfläche kann daher jeder EMF-basierte Editor für jedes Ecore-basierte Modell zur Anwendung kommen. So wird gewährleistet, dass der User das Beispiel innerhalb seiner gewohnten Umgebung erarbeiten kann. Für den Vergleich dieser beiden Modelle, der nach der Betätigung des Buttons *Compare* vorgenommen wird, kann der statusbasierten 2-Wegvergleich von EMF Compare verwendet werden. Damit die so ermittelten Operationen zuverlässig und exakt sind, muss jedoch ein perfekter Match im Gegensatz zum heuristischen Match aus EMF Compare eingesetzt werden. Hierfür kann, wie oben bereits erwähnt, im *Schritt 2*, wo beide Modelle exakt identisch sind, ein 1:1-Match durchgeführt werden. Dieser Match kann dann während dem gesamten Vorgang der Operationsdefinition im Speicher gehalten werden, sodass zu jedem Zeitpunkt, auch nach einer starken Veränderung der Elemente, diese einander zuordenbar bleiben.

In Bezug auf die Umsetzung dieses Ansatzes ist weiterhin die Erzeugung und Anwendung der Symbole und Bedingungen ein zentraler Aspekt. Diese werden in *Schritt 6* aus den beiden Modellversionen abgeleitet. Zur Abbildung der Bedingungen kann OCL verwendet werden. OCL ist im Bereich der Modellierung ein weit verbreitetes Mittel um Einschränkungen auszudrücken und eignet sich daher auch hervorragend für die Implementierung dieses Ansatzes. Zu Beginn werden im *Schritt 6* die Symbole für jedes Modellelement der beiden Modelle erzeugt. Zur Definition dieser Symbole kann die OCL-Funktion *def* angewendet werden. Das Beispiel in Listing 8.1 soll dies verdeutlichen.

Listing 8.1: Definition der Symbole in OCL.

```
1 context A
2 def: a   : NamedElement = self
3 def: op1 : NamedElement = self.eContents()->at(1)
4 def: op2 : NamedElement = self.eContents()->at(2)
```

Um die Symbole, aber auch die anderen Bedingungen, abzubilden, muss zu allererst ein Kontextelement definiert werden. Dies kann, sofern nur ein Element die Wurzel des *Origin*-Modells bildet, automatisch ausgewählt werden – auch wenn im *Working*-Modell andere Modellelemente auf erster Ebene hinzugekommen sind. In diesem Fall kann das Element im *Working*-Modell als Kontextelement dienen, das dem eindeutig auswählbaren Kontextelement im *Origin*-Modell gegenüberliegt (1:1-Match). Im Beispiel der OCL-Symboldefinitionen wird die Auswahl des Kontextelements in der ersten Zeile vorgenommen und diesem in der nächsten Zeile der Name *a* degeben. In Zeile 3 wird das erste Modellelement, das von *a* beinhaltet wird, als *op*1 festgelegt usw. Sind alle Symbole in dieser Form abgebildet, können nun für jedes Symbol die Eigenschaften als Bedingungen erzeugt werden. Es müssen dazu alle Eigenschaften und deren Werte des jeweiligen Objekts über die generischen Schnittstellen von EMF durchlaufen werden und als Bedingung formuliert werden.

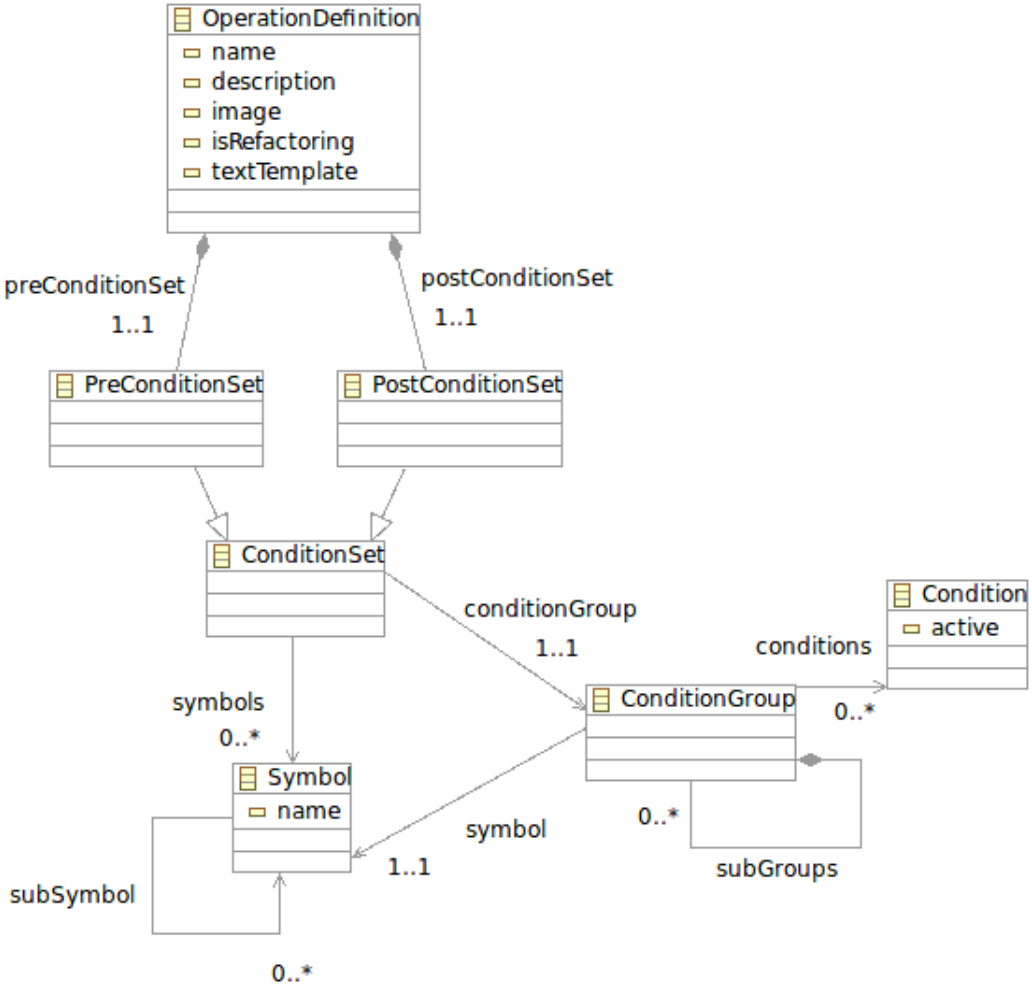

Abbildung 8.4: Metamodell für die Operationsdefinition.

Wenn eine Definition vom User durchgeführt wurde, muss diese in irgendeiner Form gespeichert werden. Das geschieht vorzugsweise modellbasiert. In Abbildung 8.4 ist daher das hierfür entworfene Metamodell zur Abbildung einer Operationsdefinition dargestellt. Die zentrale Klasse dieses Metamodells ist *OperationDefinition*. Diese enthält neben dem Namen einer textuellen Beschreibung und anderen Attributen auch ein

textTemplate. Das wird verwendet, um eine gefundene Instanz der definierten Operation im Differenzmodell in menschenlesbarer Form anzuzeigen. Beispielsweise könnte das Texttemplate für die Singleton-Definition wie in Listing 8.2 aussehen. Ausdrücke in Form von #{...} werden vor der Anzeige einer Operation mit den konkreten Instanzen ersetzt.

Listing 8.2: Texttemplate zur Beschreibung der Differenz.

```
#{A1.name} was converted into a Singleton
```

Eine *OperationDefinition* enthält außerdem noch eine Referenz auf das *MatchModel*, das *DiffModel* und jeweils das Modell aus dem Bereich *Origin* und *Working*. Diese Referenzen auf externe Klassen sind in der Abbildung 8.4 nicht enthalten. Eingezeichnet sind jedoch die Referenzen zu dem *Pre-* und *PostConditionSet*. Diese beiden Klassen sind von der Klasse *ConditionSet* abgeleitet und enthalten daher eine mehrwertige Referenz auf *Symbol*, die die zuvor beschriebenen und über OCL *def* abgebildeten Symbole repräsentiert. Diese direkt referenzierten Symbole sind allerdings nur die vom User zusätzlich definierten Symbole. Die automatisch abgeleiteten Symbole und deren Bedingungen sind in hierarchischer Form (Referenz *subGroups*) über *ConditionGroup*s und *Condition*s realisiert.

8.3 Erkennung der Operationen anhand erstellter Definitionen

Instanzen des zuvor eingeführten Metamodells beschreiben also eine zusammengesetzte Operation. Nun ist interessant, wie Vorkommen dieser Operation in generischen Differenzmodellen erkannt werden können. Eine Operationsdefinition besteht aus einem Differenzmuster (*Differences*), Vorbedingungen (*Preconditions*) und Nachbedingungen (*Postconditions*). Will man das Vorkommen der zusammengesetzten Operation prüfen, ist der erste Schritt die Erkennung des Differenzmusters. Wenn dieses im generischen Differenzmodell gefunden werden kann, muss geprüft werden, ob es eine Belegung der Symbole im Ursprungsmodell gibt, die die Vorbedingungen der Operationsdefinition erfüllen. Das selbe muss nun für die Nachbedingungen geschehen. Konnte auf diese Weise das Vorkommen des Differenzmusters und eine jeweilige Belegung der Symbole, die den Vor- und Nachbedingungen genügt, nachgewiesen werden, kann somit auch das Auftreten der Operation im geprüften Differenzmodell erkannt werden. Nun muss eine Instanz der eben erkannten Operation, die die gematchten Differenzen des geprüften Differenzmodells zusammenfasst, erzeugt und anschließend in das Differenzmodell eingefügt werden. Damit ist das Differenzmodell um die erkannte Operation erweitert worden und somit der Erkennungsprozess abgeschlossen.

8.4 Einschränkungen

Der eben vorgestellte Ansatz weist jedoch noch einige Einschränkungen auf. Komplexe Refactorings mit unvorhersehbarer Anzahl an Operationen können noch nicht eindeutig

definiert werden. Das folgende Beispiel in Abbildung 8.5 stellt das *Origin*- und *Working*-Modell der Operation *Extract Super-Class* dar und dient zur Demonstration dieser Einschränkung.

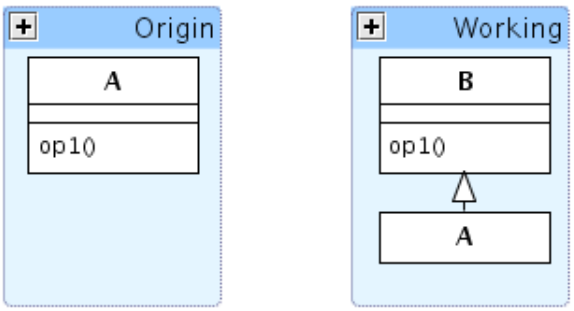

Abbildung 8.5: Beispielmodelle zur Demonstration der Einschränkung.

Bei der zusammengesetzten Operation *Extract Super-Class* wird eine neue Klasse, die Superklasse, hinzugefügt und ein oder mehrere Operationen von einer oder mehreren Klassen in diese Superklasse gezogen und pro Klasse eine Vererbungsbeziehung zur erstellten Superklasse erzeugt. Diese Beschreibung der Operation lässt schon vermuten, dass es sich nicht um eine vorhersehbare Anzahl an Operationen, wie im Fall der Singleton-Operation, handelt. In der Abbildung 8.5 sind alle für diese zusammengesetzte Operation notwendigen Änderungen dargestellt. Diese sind das Hinzufügen einer neuen Klasse (B), das Erzeugen der Vererbungsbeziehung (A zu B) sowie das Verschieben der Operation *op1* von A nach B.

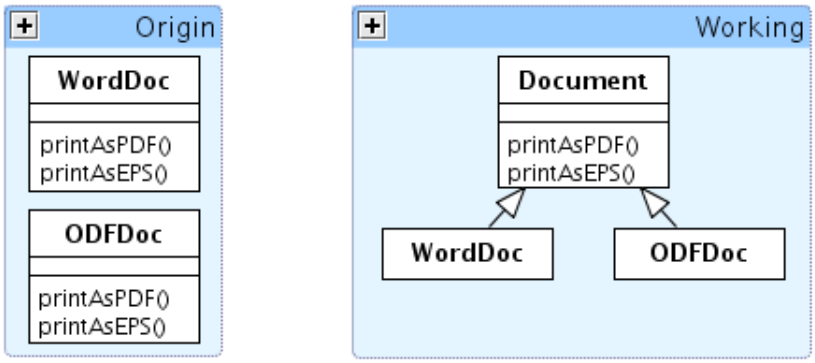

Abbildung 8.6: Anwendungsfall zur Demonstration der Einschränkung.

Abbildung 8.6 zeigt einen Anwendungsfall, bei der die Operation *Extract Super-Class* durchgeführt wurde. Die Operationsdefinition mit den in Abbildung 8.5 dargestellten Beispiel-Modellen würden hier jedoch je nach Definition der OCL-Bedingungen entweder zweimal oder sogar viermal matchen, obwohl die Operation nur einmal ausgeführt wurde. Das liegt daran, dass die Operation *Extract Super-Class* auch auf mehrere Klassen und/oder Methoden gleichzeitig und nicht immer nur auf eine Klasse mit einer Methode angewendet werden kann.

Um auch derartige Fälle abzudecken, müsste eine Möglichkeit geschaffen werden, die dem User erlaubt, Operationen im Bereich *Differences* zu gruppieren und für solche

Gruppe Wiederholungsbedingungen zu spezifizieren. Der User könnte so im obigen Fall einerseits definieren, dass für mehrere Klassen eine Vererbungsbeziehung hinzukommen und mehrere Operationen all dieser Klassen in die Superklasse gezogen werden können, um diese Situation *einmal* als *Extract Super-Class* zu matchen. Jede Operationsgruppe müsste daher einerseits mit einem Selektor ausgestattet werden, der die Modellelemente, für die sich diese Operationsgruppe wiederholen könnte, auswählt, und andererseits mit einem Quantor ausgezeichnet werden, der angibt, ob diese Gruppe *für alle*, *für einige* oder *genau für eines* der selektierten Modellelemente ausgeführt werden kann.

Außerdem ist es nach dem derzeitigen Ansatz nicht möglich, mehrere Arten der Ausführung einer zusammengesetzten Operation anzugeben. Oft sind mehrere Wege möglich, ein und die selbe Operation durchzuführen. Beispielsweise könnte eine Methode im obigen Beispiel nicht verschoben werden, sondern gelöscht und neu angelegt worden sein. Demnach müssten Operationsgruppen, die ja standardmäßig mit einem logischen *Und* verknüpft sind – es wird nach dem gesamten Differenzmuster, also Differenz 1 *und* Differenz 2 *und* Differenz 3 etc., gesucht – auch wahlweise mit einem *Oder* verknüpfbar sein.

8.5 Anwendung der Operationsdefinitionen und gefundener Operationen

Da die Operationsdefinition bei diesem Ansatz explizit und nicht in der Implementierung eines Editors versteckt vorliegt, wie dies beim direkten Tracking innerhalb der Editoren der Fall wäre, entstehen weitere Nutzungsmöglichkeiten der Operationsdefinitionen.

Eine dieser Möglichkeiten ist die Ausführung der definierten, zusammengesetzten Operationen. Mithilfe des EMF Compare Merge-Services können alle atomaren Differenzelemente auf ein Modell angewendet werden, um eine Version in eine andere überzuführen. Es wäre also eine Implementierung denkbar, die den User auffordert, für jedes Symbol einer Operationsdefinition ein Modellelement auszuwählen. Diese Auswahl muss selbstverständlich den Vorbedingungen, die durch die OCL-Constraints jedes Symbols explizit vorliegen, genügen. Hinter jedem Symbol stecken nun ein oder mehrere Modellelemente, die eine gültige Ausgangssituation für die zusammengesetzte Operation sind. Nun können alle generischen, atomaren Operationen, die das *DiffModel* (Bereich *Differences*) enthält, auf die ausgewählten Modellelemente mithilfe des EMF Compare Merge-Services ausgeführt werden. Auf diese Weise können beispielsweise Editoren das Ausführen definierter, zusammengesetzter Operationen auf Modelle anbieten.

Außerdem können die expliziten Operationsdefinitionen genutzt werden, um das in ihnen steckende Wissen zu dokumentieren. Durch eine *Model-to-Model*-Transformation wäre es möglich, ähnlich wie JavaDoc eine Dokumentation dieser Operation zu generieren.

Vor allem wird diese Operationsdefinition jedoch für das Erkennen von Anwendungen einer komplexeren Operation oder eines Refactorings innerhalb eines generischen Differenzmodells verwendet, unter der Voraussetzung, dass die sprachspezifische Schicht, also die Operationsdefinitionen selbst, leichtgewichtig, deskriptiv und von der generischen Schicht, also der Vergleichsalgorithmus, getrennt ist. Die Erkennung und Auszeichnung

sprachspezifischer Operationen und Refactorings ist im Kontext der sprachspezifischen Konflikterkennung ein wichtiger Aspekt. Zwei Änderungen können sich in Bezug auf ihre Intention widersprechen, also in Konflikt zu einander stehen und dennoch aus generischer Sicht völlig problemlos vereinbar sein. Im Kontext des UML Klassendiagramms wäre dies beispielsweise die Extraktion einer Superklasse auf der einen Seite und die Extraktion einer Schnittstelle auf der anderen (siehe Abbildung 9.3). Betrachtet man nur das zusammengeführte (offensichtlich valide) Ergebnis, ist es nahezu unmöglich, diesen Widerspruch der Intentionen zu erkennen. Auch bei der Betrachtung der atomaren, generischen Operationen fällt die Erkennung schwer und verlangt eine komplexe, unübersichtliche Definition eines derartigen Konflikts. Hat man jedoch die zusammengesetzten Operationen mittels einer Definition erkannt und im Differenzmodell ausgezeichnet, lässt sich der im vorigen Beispiel genannte Konflikt leichter abbilden und in weiterer Folge aufdecken.

Die Erkennung von Refactorings bringt zusätzlich noch weitere, entscheidende Vorteile, da hinter einer vermeintlich großen Anzahl an kleinen Änderungen die tatsächliche Intention abgeleitet werden kann. Es können generische Konflikte, die durch die kleinen Änderungen verursacht wurden, verhindert werden, indem das Refactoring erst nach der Durchführung aller sonstigen Änderungen erneut ausgeführt wird (vgl. Ausführung von zusammengesetzten Operationen weiter oben). Ein Beispiel, wie die Erkennung von Refactorings die Konfliktmeldungen minimieren kann, wurde bereits in Kapitel 3.4 besprochen.

9 Erkennung von Konflikten

In den letzten Kapiteln wurde die theoretische Grundlage des Modellvergleichs und der Deltarepräsentation (Kapitel 3) erläutert, eine Implementierung auf Basis der Eclipse-Plattform (Kapitel 7) beschrieben, sowie ein Ansatz zur Definition und Erkennung sprachspezifischer, zusammengesetzter Operationen und Refactorings auf Basis generischer Deltas (Kapitel 8) vorgestellt. Es ist nun nach Anwendung aller vorgestellten Ansätze möglich, die innerhalb einer Änderungssession eines Users durchgeführten Operationen, angereichert mit weiteren Informationen, wie beispielsweise Refactorings, zu erkennen und abzubilden. Wenn mehrere Personen parallel an Modellen innerhalb eines Projekts arbeiten, kann es klarerweise vorkommen, dass zwei Personen das selbe Modell verändern. In diesem Fall müssen die vorgenommenen Änderungen dieser beiden Personen analysiert werden, um festzustellen, ob ein Konflikt vorliegt oder ob die Änderungen beider Personen problemlos kombiniert (Durchführung eines Merges) werden können. Bevor nun unterschiedliche Möglichkeiten dieser Analyse im Detail besprochen werden, folgt eine kurze Wiederholung der Arten und Einteilungsmöglichkeiten von Konflikten, die bei gleichzeitiger Veränderung von Modellen auftreten können. Dem folgt dann die Erläuterung der Möglichkeiten, die jeweiligen Konflikte zu erkennen. In Abbildung 9.1 ist zur besseren Übersicht das Bild aus Kapitel 3 noch einmal dargestellt. Diese Kategorisierung basiert in erster Linie auf den technischen Anforderungen, um Konflikte zu erkennen, und nicht auf der Ausprägung der Konflikte selbst.

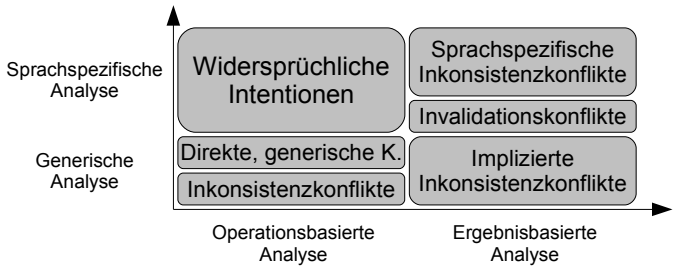

Abbildung 9.1: Konflikte und deren Erkennung.

Abbildung 9.1 zeigt insgesamt sechs verschiedene Konfliktarten, die jeweils auf den zwei Dimensionen der Erkennbarkeit aufgetragen sind. Die vertikale Dimension gibt an, ob sprachspezifisches Wissen notwendig ist, um Konflikte entdecken zu können, während die horizontal aufgetragene Dimension die Notwendigkeit ausdrückt, auch das Ergebnis einer Zusammenführung für eine erfolgreiche Erkennung berücksichtigen zu müssen. Diese Konfliktarten werden in der folgenden Auflistung näher spezifiziert.

Inkonsistenzkonflikte: Diese Konfliktgruppe umfasst Konflikte, die auftreten, wenn die Operationsfolgen zweier Personen nicht gesamtheitlich vereinbar sind, da mindestens eine Operation durch eine andere ungültig oder undurchführbar gemacht

wurde. Dies ist beispielsweise der Fall, wenn ein Element gelöscht wurde, das von einer anderen Person geändert wurde (→ *DeleteUpdate*-Konflikt).

Direkte, generische Konflikte: Generische Konflikte im Allgemeinen sind jene Konflikte, die durch eine sprachunabhängige Analyse erkannt werden können. Wenn diese zudem noch direkt, also durch die reine Betrachtung der Operationsfolge ersichtlich werden, handelt es sich um Operationen, die sich direkt widersprechen und lokal nicht kommutativ sind. Das bedeutet, dass deren Reihenfolge der Ausführung einen Einfluss auf das Endergebnis hat. Beispielsweise besteht ein solcher Konflikt, wenn beide Personen das selbe Element verändern. Im diesem Zusammenhang spielt die *Unit of Consistency* eine Rolle. Diese gibt an, ab welcher Elementgranularität bei gleichzeitiger Änderung ein Konflikt gemeldet werden soll. Im Kontext des UML Klassendiagramms kann beispielsweise diese *Unit of Consistency* auf Klassenebene gesetzt werden, um jede gleichzeitige Änderung einer Klasse, also z.B. die Änderung des Klassennamens auf der einen Seite und die Änderung der Eigenschaft `isAbstract` auf der anderen, als Konflikt anzusehen. Ist die Granularität feiner eingestellt, widersprechen sich diese Änderungen nicht und können vereint werden, sodass nur die Änderung der selben Eigenschaft als Konflikt gemeldet wird (→ *ChangeChange*-Konflikt).

Implizierte Inkonsistenzkonflikte: Konflikte dieser Kategorie ähneln jenen der zuvor besprochenen Inkonsistenzkonflikten. Nur können diese nicht direkt durch die Analyse der Operationsfolge erkannt werden, sondern werden nur ersichtlich, wenn auch deren Ausführung auf den Ursprung und das resultierende Ergebnis berücksichtigt wird. Diese Probleme treten daher nicht direkt auf, sondern werden nur durch Operationen impliziert. Als Beispiel könnte ein Fall genannt werden, indem eine Person eine Klasse löscht und eine andere Person ein Attribut dieser Klasse geändert hat. Auf den ersten Blick kann man sagen, dass dies eigentlich ein Inkonsistenzkonflikt ist. Schließlich wird ein Element geändert, das zuvor gelöscht wurde. Jedoch wird bei genauerer Betrachtung klar, dass das geänderte Element nicht direkt gelöscht wurde und daher bei reiner Betrachtung der Operationen diese Inkonsistenz nicht festgestellt werden kann. Die Feststellbarkeit hängt natürlich auch von der Granularität der Deltas ab. Werden alle implizit gelöschten Elemente ebenfalls innerhalb der Deltarepräsentation erfasst, kann ein Teil dieser Konfliktart auch bei reiner Betrachtung der Operationen erkannt werden. Im Sinne der Minimalität der Deltas kann jedoch nicht davon ausgegangen werden, dass implizierte Änderungen im Deltadokument explizit aufscheinen.

Invalidationskonflikte: Invalidationskonflikte, also Operationsfolgen, deren Kombination ein invalides Resultat zur Folge haben, können nicht durch die alleinige Analyse der Operationsfolgen erkannt werden. Hierfür ist die Betrachtung und Interpretation der Status bzw. die des Resultats sowie sprachspezifisches Wissen notwendig.

Sprachspezifische Inkonsistenzkonflikte: Diese Konflikte treten auf, wenn Änderungen bezüglich der sprachspezifischen Semantik zu einander inkonsistent sind. Folgendes Beispiel im Kontext des UML Klassendiagramms (siehe Abbildung 9.5) soll derartige Konflikte näher erläutern: Wenn eine Person eine neue Klasse hinzufügt und diese als *Containment* einer anderen Klasse definiert, indem er oder sie eine *Containment*-Referenz erstellt, dann ist dieses *Containment* nur durch die Semantik der Beziehung definiert und nicht durch ein physisches *Containment* auf generischer Elementebene gegeben. Wenn nun die andere Person die bestehende Klasse entfernt, die die neu erstellte Klasse durch die *Containment*-

Referenz beinhaltet, steht die zuvor beinhaltete Klasse alleine da. Das war aber vermutlich nicht die Intention der ModelliererInnen. Da das *Containment* einzig durch die Semantik der *Containment*-Beziehung definiert ist, kann dieser Konflikt nur durch sprachspezifisches Wissen und eine Analyse des Ergebnisses erkannt werden.

Widersprüchliche Intentionen: Bei der ausschließlichen Betrachtung der Operationsfolgen zweier Personen in Verbindung mit sprachspezifischem Wissen können Konflikte erkannt werden, die entstehen, wenn Operationen in Hinsicht auf deren Absicht nicht vereinbar sind. Das ist im Kontext des UML Klassendiagramms beispielsweise der Fall, wenn eine Person ein *Interface* aus einer Klasse und eine andere Person eine abstrakte Klasse extrahiert (siehe Abbildung 9.3). Spezifisches Wissen ist zur Erkennung solcher Konflikte zwar nötig, die Analyse des Mergeresultats ist allerdings nicht erforderlich. Es widersprechen sich lediglich die durchgeführten Operationen. Natürlich ist die Voraussetzung für die Erkennung, dass diese Refactoringoperationen dementsprechend erkannt wurden.

Da nun die unterschiedlichen Konfliktarten in Hinsicht auf deren Sprachabhängigkeit und Anforderungen, diese erkennbar zu machen, besprochen wurden, soll nun auf die Möglichkeiten eingegangen werden, diese Konflikte tatsächlich zu erkennen.

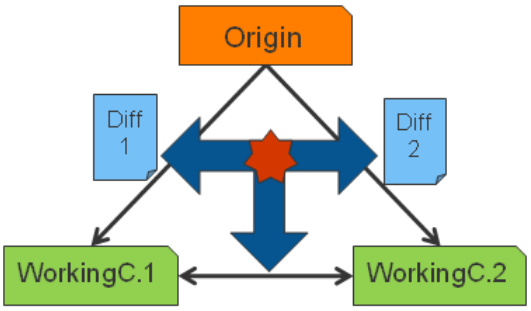

Abbildung 9.2: Konflikterkennung auf Basis zweier Deltadokumenten.

Die Ausgangssituation der im Folgenden beschriebenen Konflikterkennung ist in Abbildung 9.2 abgebildet und umfasst die Existenz zweier Deltadokumenten, *Diff1* und *Diff2*, die jeweils die Operationen von einem gemeinsamen Ursprung, *Origin*, zu zwei unterschiedlich veränderten Modellen, *WorkingC.1* und *WorkingC.2* beinhalten. Im weiterer Folge wird davon ausgegangen, dass zusammengesetzte, sprachspezifische Operationen erkannt und in die Deltadokumente übernommen wurden. Möglichkeiten derartiger Definition und Erkennung wurden in Kapitel 8 vorgestellt. In diesem Kontext ist das Ziel die Erkennung von Konflikten und Vereinigung (roter Stern in Abbildung 9.2) der nicht in Konflikt stehenden Operationen, sodass letzlich ein Deltadokument erzeugt wird, das alle Änderungen vereinigt und deren eventuelle Konflikte aufzeigt (*Diff1 union Diff2*). Da auch sprachspezifische Konflikte, also widersprüchliche Operationen, sprachspezifische Inkonsistenz- und Invalidationskonflikte, erkannt werden sollen, wird auch im Zuge dieses Abschnitts auf deren Definition eingegangen. Im Kapitel 5 wurde betont, sprachspezifische Erweiterungen ausschließlich durch benutzerfreundliche und deskriptive Artefakte definierbar zu machen. Das gilt folglich auch für die Definition sprachspezifischer Konflikte.

9.1 Operationsbasierte Konflikterkennung

Zu Beginn wird auf die Konflikterkennung auf Basis der reinen Operationsanalyse eingegangen. Diese sind entsprechend der Abbildung 9.1 generische Konflikte und Inkonsistenzkonflikte. Wird zusätzlich noch auf sprachspezifisches Wissen, in Form von benutzerdefinierten Konfliktbeschreibungen, eingegangen, können auch widersprüchliche Intentionen erkannt werden.

9.1.1 Erkennung von generischen Konflikten und Inkonsistenzen

Generische Konflikte (*ChangeChange*-Konflikte) sowie direkte Inkonsistenzen (*DeleteUpdate*-Konflikte) können durch die sprachunabhängige Betrachtung der Operationen ermittelt werden. Im Allgemeinen kommen hierbei die Regeln entsprechend Tabelle 3.1 aus Kapitel 3 zur Anwendung.

Nach [30] tritt ein Konflikt bei der Analyse von Operationsfolgen auf, wenn die Operationen nicht lokal kommutativ sind, also die Reihenfolge ihrer Ausführung einen Einfluss auf das Ergebnis hat. Dies ist immer der Fall, wenn zwei oder mehrere Operationen das selbe Element verändern. In [30] ist der Nachweis der Kommutativität von großer Bedeutung, da in seiner Arbeit auch Operationen, wie *Erhöhe Feld 'x' um 2*, berücksichtigt werden. Derartige Operationen können jedoch nur durch das direkte Mitschreiben der Operationen erkannt werden, da im Nachhinein nicht mit Sicherheit abgeleitet werden kann, ob beispielsweise das Feld einmal um 2 erhöht wurde, oder 2 mal um 1. Im Kontext der Modellversionierung ist es nicht vordergründig, wie ein Element verändert wurde, sondern nur welchen Wert dieses Feld schlussendlich einnimmt. Aus diesem Grund muss zur Erkennung generischer Konflikte nicht unbedingt die Kommutativität nachgewiesen werden, sondern lediglich geprüft werden, ob eine Operation das selbe Element des Ursprungs betrifft, das auch von einer Operation im gegenüberliegenden Deltadokuments verändert wurde.

Zur Erkennung von direkten Inkonsistenzen zwischen zwei Deltadokumenten ist die Vorgehensweise ähnlich wie jene der Erkennung generischer Konflikte. Es muss für jede Operation, die ein Element verändert, nach einer Operation im gegenüberliegenden Deltadokument gefunden werden, die dieses Element löscht. Letztlich ist sowohl das Ändern als auch das Löschen eines Elements eine generische Operation, die ein Element aus dem Ursprungselement referenziert und daher entweder zu einem *ChangeChange*- oder einem *DeleteUpdate*-Konflikt führen kann. Im Listing 9.1 ist der Algorithmus zur Erkennung generischer Konflikte und Inkonsistenzkonflikte skizziert.

Das Listing 9.1 soll die Vorgehensweise nur exemplarisch verdeutlichen. Die Laufzeit ist mit n^m – wobei n die Anzahl der Operationen der einen Seite und m die Operationen der anderen Seite ist – natürlich nicht optimal, sofern die Methode `findDiffElement()` eine lineare, uninformierte Suche implementiert. In einer produktiven Implementierung kann die Laufzeit durch die Verwendung von *Hash-Maps*, die das Element des Ursprungs als Schlüssel verwenden und auf das jeweilige *DiffElement* mappen, oder durch die jeweilige Extraktion der veränderten Ursprungselemente in eine Liste und Kontrolle, ob ein Element zwei mal verändert wurde, deutlich verbessert werden.

Listing 9.1: Generische Erkennung von Konflikten.

```
// diff1 (Differenzen von Origin zu WorkingC.1
DiffModel diffModel1 = ...

// diff2 (Differenzen von Origin zu WorkingC.2
DiffModel diffModel2 = ...

for (DiffElement diffElement : diffModel1.getDiffElements()) {
  DiffElement oppositeDiff =
    findDiffElement( // Findet DiffElemente
      diffModel2.getDiffElements(), // in dieser Diff-Liste
      diffElement.getLeftElement() // mit diesem LeftElement
      );
  if (oppositeDiff != null) {
    // es gibt einen Konflikt
    if (oppositeDiff.getType() == DELETE_TYPE ||
        deffElement.getType() == DELETE_TYPE
    ) {
      // DeleteUpdate-Konflikt gefunden
      ...
    } else {
      // ChangeChange-Konflikt gefunden
      ...
    }
  }
}
```

9.1.2 Erkennung widersprüchlicher Intentionen

Die Erkennung widersprüchlicher Intentionen ist komplizierter als jene der generischen Konflikte. Schließlich muss hierbei schon sprachspezifisches Wissen abgebildet und zur Erkennung verwendet werden. Bevor eine Möglichkeit vorgestellt wird, wie diese Konflikte definiert und gefunden werden können, soll zuvor noch die Erkennung solcher Konflikte anhand eines Beispiels motiviert werden.

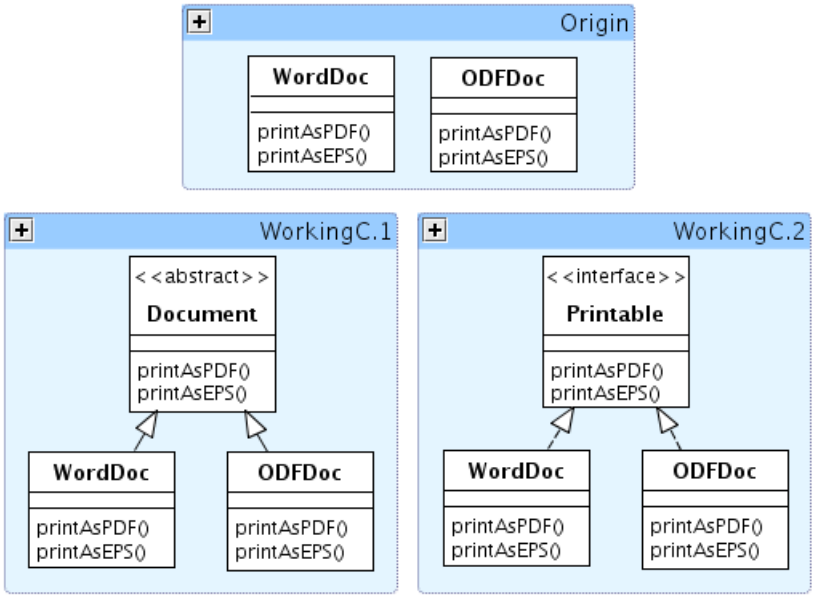

Abbildung 9.3: Widersprüchliche Intentionen.

In Abbildung 9.3 wurde von einer Person in *WorkingC.1* eine abstrakte Superklasse erstellt, die die gemeinsamen Methoden `printAsPDF()` und `printAsEPS()` abstrahiert. Die andere Person hat in *WorkingC.2* ein *Interface* extrahiert, das von den beiden bestehenden Klassen implementiert wird. Eine Zusammenführung beider Änderungen würde sowohl aus generischer Sicht als auch aus syntaktischer Sicht ein korrektes Ergebnis liefern. Dennoch sind die Operationen beider Personen in den meisten Fällen widersprüchlich und stellen daher einen Konflikt dar. Um diesen erkennen zu können, muss der User die Operationen *Extract super class* und *Extract interface* als widersprüchliche Operationen definieren. Diese Definition kann über eine ähnliche Methode, wie die Definition zusammengesetzter Operationen in Kapitel 8, durchgeführt werden. Die zu modellierenden Aspekte finden doch in genau umgekehrter Weise statt. Letztlich muss der User festlegen können, dass ein Differenzmuster widersprüchlich zu einem anderen Differenzmuster ist. Vom Konzept her ähnelt diese Vorgehensweise auch dem Ansatz der *Conflict Sets* in [19].

In Abbildung 9.4 ist eine Skizze der Benutzeroberfläche zur Definition widersprüchlicher Operationen dargestellt. Die Anwendung bzw. Funktionsweise einer Definition widersprüchlicher Operationsmuster wird in der folgenden Aufzählung Schritt für Schritt erläutert.

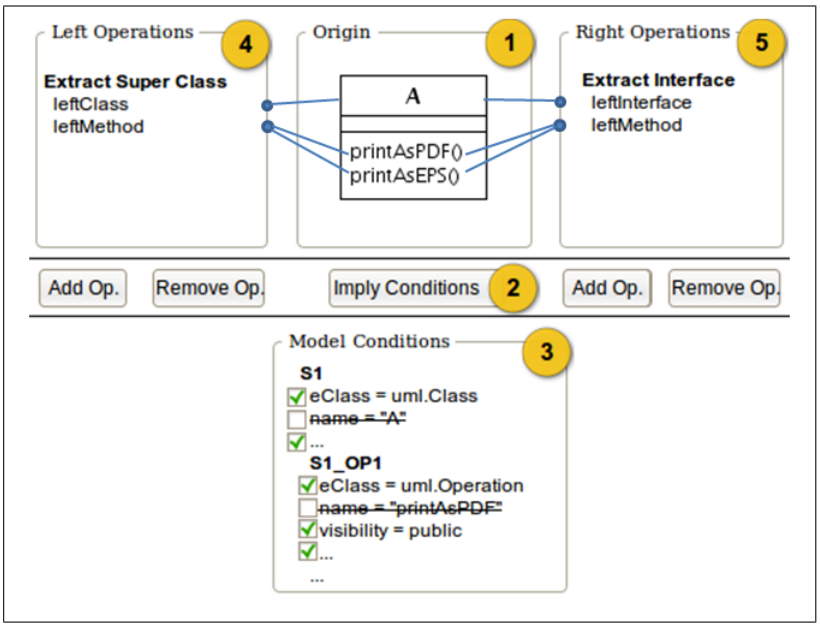

Abbildung 9.4: Benutzeroberfläche zur Definition widersprüchlicher Operationen.

Schritt 1: Ausgangssituation beispielhaft modellieren.

Im Bereich *Origin* muss zuerst die Ausgangssituation für beide später als widersprüchliche Operationen definierte Differenzmuster modelliert werden. Wichtig ist hierbei nicht, dass die Ausgangssituation vollständig abgebildet wird, sondern nur, dass all jene Modellelemente angeführt werden, die von beiden Differenzmustern gleichzeitig verändert werden. Der Bereich *Origin* dient später als Mapping der von beiden Differenzmustern verwendeten Elemente.

Schritt 2: Modellbedingungen ableiten.

Sind alle von beiden Differenzmuster verwendeten Elemente modelliert, muss der Button *Imply Conditions* gedrückt werden. Dieser löst das Ableiten, wie jener in Kapitel 8, der OCL-Bedingungen für das *Origin*-Modell aus.

Schritt 3: Modellbedingungen anpassen.

Wie auch bei der Operationsdefinition in Kapitel 8 können nun die Bedingungen, denen Instanzen des Ursprungsmodells genügen müssen, aufgelockert werden. In unserem Beispiel sind das die jeweiligen Namen der Modellelemente. Außerdem müssten auch die abgeleiteten Bedingungen der Rückgabetypen der beiden Operationen *printAsPDF()* und *printAsEPS()* verallgemeinert werden, da es bei beiden Operationen keine Vorbedingung ist. Eventuell können diese Deaktivierungen bzw. Editierungen der Modellbedingungen aus den Vorbedingungen der Operationsdefinition übernommen werden.

Schritt 4: Erstes Differenzmuster erstellen.

Über den Button *Add Op.* kann nun das Differenzmuster der einen Seite abgebildet werden. Zur Auswahl stehen hier generische, atomare Operationen, sowie zuvor definierte, zusammengesetzte Operationen oder Refactorings. Hier sollte es zudem möglich sein, Differenzgruppen mit logischen Operatoren, wie beispielsweise *oder*, zu verknüpfen. Jede ins Differenzmuster übernommene Operation hat Eigenschaften, die ein Element im Ursprungsmodell referenziert. Diese Referenzen geben an, welche Modellelemente des Ursprungs dieser Operation unterzogen

wurden. Im Zuge der Erstellung des Differenzmusters müssen nun diesen Eigenschaften Modellelemente in unserem exemplarischen Ursprungsmodell zugeordnet werden. Dies geschieht durch die in blau dargestellten Linien.

Schritt 5: Zweites Differenzmuster erstellen.

Wie im Schritt 4 müssen nun die der gegenüberliegenden Seite widersprüchlichen Operationen über den Button *Add Op.* hinzugefügt und die jeweiligen Eigenschaften dieser Operationen mit dem exemplarischen Ursprungsmodell verknüpft werden. Diese Verknüpfungen geben nun an, dass die eben definierten Operationen nur dann widersprüchlich sind, wenn sie auf die selben Elemente angewendet werden wie das linke Differenzmuster. Das Modell im Bereich *Origin* dient daher gewissermaßen als Mapping zu den von beiden Operationen verwendeten Modellelementen. Diese Modellelemente müssen zudem noch den in Schritt 3 angepassten Bedingungen genügen und stellen daher Schablonen dar, die bei der Erkennung ausgefüllt werden müssen.

Sind widersprüchliche Intentionen auf die eben beschriebene Art und Weise abgebildet, indem ihre Differenzmuster, angewendet auf ein Muster im Ursprungsmodell, als widersprüchlich definiert wurden, sind diese relativ einfach in den beiden Deltadokumenten erkennbar. Es muss nur nach dem Vorkommen des linken bzw. rechten Differenzmusters gesucht werden und anschließend eine Belegung der Modellschablone, passend zu den Modellbedingungen, gefunden werden. Wurde eine Belegung gefunden, müssen die Differenzmuster natürlich dementsprechend auf diese gefundene Belegung angewendet worden sein, um die widersprüchliche Intention als Konflikt melden zu können. Konnte keine Belegung gefunden werden, wurden zwar die Operationen durchgeführt, jedoch nicht auf die selben Elemente bzw. nicht auf die definierte Art, sodass kein Widerspruch vorliegt.

9.2 Ergebnisbasierte Konflikterkennung

Im vorigen Abschnitt wurde die Konflikterkennung auf Basis ermittelter Operationen zweier EntwicklerInnen erläutert. Nun werden jene Konfliktsituationen betrachtet, die nur durch eine genauere Analyse des Ergebnisses dieser Operationen erkannt werden können. Wie bereits erwähnt, handelt es sich in diesem Zusammenhang um implizierte Inkonsistenzen, die auch ohne sprachspezifisches Wissen ermittelt werden können, sowie um Invalidationskonflikte und sprachspezifische Inkonsistenzen, für deren Erkennung neben der Berücksichtigung des Ergebnisses auch sprachspezifische Information nötig ist.

9.2.1 Erkennung von implizierten und sprachspezifischen Inkonsistenzen

Die Erkennung von implizierten Inkonsistenzkonflikten ist relativ einfach. Sie entstehen, wenn durch eine Entfernung eines Elements implizit auch deren physische Containments entfernt werden, die von einer anderen Person geändert wurden. Auch wenn die dominante Auflösungsstrategie derartiger Konflikte die Durchführung der Entfernung wäre, sollte eine derartige Situation auf Wunsch des Benutzers dennoch erkannt

und als potenzieller Konflikt ausgezeichnet werden. Beispielsweise trifft dies im UML Klassendiagramm auf Attribute zu, wenn ihre beinhaltende Klasse entfernt wurde. Zur Erkennung implizierter Inkonsistenzkonflikte muss bei jedem entfernten Element überprüft werden, ob auf der gegenüberliegenden Seite ein Containment dieses Elements verändert wurde. Diese Vorgehensweise wurde in Listing 9.2 mit EMF skizziert. Dieser Algorithmus überprüft für jedes Containment eines `EObject`s über die Methode `eContents()`, ob eine Änderung auf der gegenüberliegenden Seite vorgenommen wurde. Sofern eine Änderung im gegenüberliegenden *DiffModel* gefunden werden kann, liegt ein implizierter Inkonsistenzkonflikt vor.

Auf die eben beschriebene Weise können allerdings nur implizierte Inkonsistenzen erkannt werden, die durch physische Containments entstanden sind. Oft liegt die Containmentbeziehung jedoch nicht physisch vor, sondern wird durch die Semantik eines gewissen Elements ausgedrückt. Beispielsweise wurde eine kompositionelle Beziehung innerhalb des in EMF realisierten UML 2 Klassendiagramms in Version 3.0 über eine *Property* realisiert, die eine *AggregationKind* mit dem Wert *COMPOSITE* besitzt und das beinhaltete Element referenziert. Da es sich hier rein um die Semantik einer Sprache handelt, muss diese vom User für konkrete Sprachen bzw. Metamodelle spezifiziert werden. Zur Spezifikation derartiger semantischer Containment-Beziehungen bietet sich wiederum die Verwendung von OCL an. Diese OCL-Bedingung müsste, ausgehend von einem Elementtyp, alle semantisch beinhalteten Elemente auswählen. Im zuvor genannten Beispiel wären dies alle referenzierten Elemente einer *Property*, deren Eigenschaft *Aggregation* ein *AggregationKind:COMPOSITE* innehat. Im oben skizzierten Algorithmus (Listing 9.2) müssten dann nicht nur alle Elemente in `eContents()` auf Änderung überprüft werden, sondern außerdem alle Elemente, die aus der Ausführung der OCL-Bedingung ausgehend vom aktuellen Element (`originOfRemovedObject`) zutreffen. Außerdem bedarf es der Kontrolle, ob auf der gegenüberliegenden Seite ein Element hinzugefügt wurde, dass auf diese semantische Containment-Bedingung zutrifft. So kann auch der oben bereits beschriebene Fall abgedeckt werden, bei dem eine Person eine Klasse löscht und die andere Person eine neue Klasse hinzufügt und diese als Containment der gelöschten Klasse definiert. Dieser Fall ist zur Verdeutlichung auch in Abbildung 9.5 dargestellt. Hier wurde auf der rechten Seite eine neue Klasse *Engine* hinzugefügt, die als Containment der Klasse *Car* definiert wurde. In der linken Seite wurde jedoch die Klasse *Car* entfernt, sodass die Beziehung von *Engine* zu *Car* im Ergebnis nicht mehr realisiert werden kann und *Engine* ohne jegliche Referenz alleine im zusammengeführten Modell verbliebe.

Direkte, indirekte und sprachspezifische Inkonsistenzkonflikte sind allesamt spezielle Formen von *DeleteUpdate*-Konflikten und stellen damit keine *harten* Konflikte dar, sondern können eher als potenzielle Fehlerquellen interpretiert werden. Daher ist zumindest ein Hinweis auf derartige Fälle notwendig. Es ist wegen der Ähnlichkeit dieser Konflikte darauf zu achten, dass ein Konflikt nicht auf die eben besprochenen, unterschiedlichen Erkennungsarten mehrmals gemeldet wird. Beispielsweise könnte der in Abbildung 9.5 dargestellte Fall bereits durch die sprachunabhängige Inkonsistenzerkennung aufgedeckt werden, da der gelöschten Klasse auf der gegenüberliegenden Seite eine *Property* hinzugefügt wurde, die die Containmentbeziehung auf die Klasse *Car* referenziert. Somit wurde auch eine Eigenschaft (*Properties*) der gelöschten Klasse geändert, was zu einer generischen Erkennung eines *DeleteUpdate*-Konflikts führt. Dies ist aber abhängig von der Realisierung der Containmentbeziehung des jeweiligen Metamodells. Daher ist dennoch die Möglichkeit der Definition semantischer Abhängigkeiten, wie zuvor beschrieben, nötig, um in jedem Fall derartige Konflikte erkennen zu können.

Listing 9.2: Erkennung implizierter Inkonsistenzkonflikte.

```
DiffModel diff1 = ... // V -> V'
DiffModel diff2 = ... // V -> V''
diff1Union2.addAll(getImplicitDeleteUpdates(diff1, diff2));
diff1Union2.addAll(getImplicitDeleteUpdates(diff2, diff1));
...

private List<ImplicitDeleteUpdate> getImplicitDeleteUpdates(
    DiffModel diff1, DiffModel diff2) {
  for (EObject removedObject : getRemovedObjects(diff1)) {
    originOfRemovedObject = getOrigin(removedObject);
    for (EObject containedObject : originOfRemovedObject.
        eContents()) {
      if (isChanged(containedObject, diff2)) {
        list.add(createImplicitDeleteupdate(containedObject));
      }
    }
  }
}

/**
 * Retrieves all removed objects within the specified
 * <code>DiffModel</code>.
 */
private EList<EObject> getRemovedObjects(DiffModel diffModel)
    {
  ...
}

/**
 * Retrieves original object of changed object,
 * using the MatchModel.
 */
private EObject getOrigin(EObject eObject) {
  ...
}

/**
 * Checks for changes of specified <code>eObject</code> in
 * specifed <code>diffModel</code>.
 */
private EObject isChanged(EObject eObject, DiffModel diffModel
    ) {
  ...
}
```

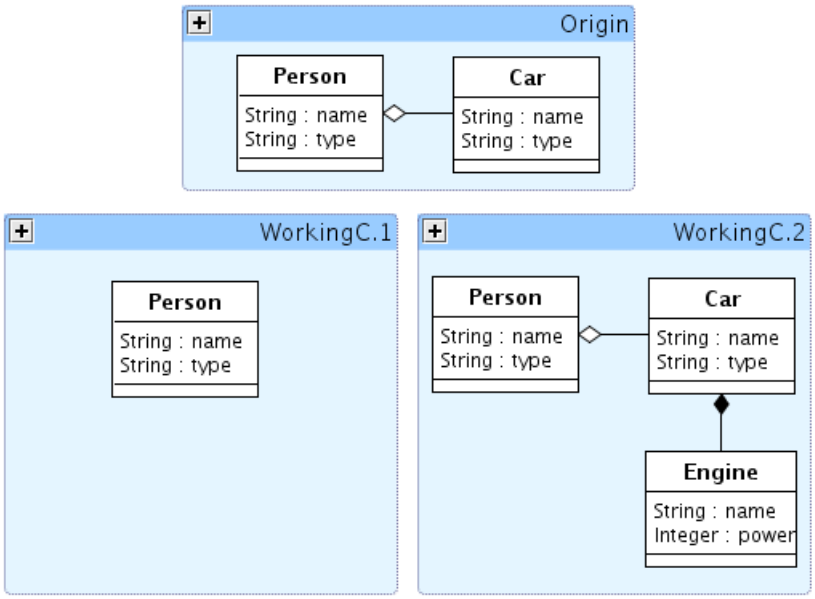

Abbildung 9.5: Inkonsistenzkonflikt durch sprachspezifische Containmentbeziehung.

9.2.2 Erkennung von Invalidationskonflikten

Die nächste und letzte Art der hier besprochenen Konflikttypen, ist jene der Invalidationskonflikte. Diese treten auf, falls die Zusammenführung aller Änderungen beider Seiten ein invalides Modell hervorruft. Die Invalidität ist nur auf Ebene des Metamodells definierbar. Die Erkennung derartiger Konflikte bedarf daher sowohl der Betrachtung des Ergebnisses, als auch sprachspezifisches Wissen. Dieses sprachspezifische Wissen muss in Form von Validationsregeln, die ein Modell auf seine Konformität zum Metamodell prüft, vorliegen. Diese Überprüfung kann damit mit der statischen Kontrolle der Syntax einer Sprache in der Quellcode-Domäne verglichen werden. In Abbildung 3.5 im Kapitel 3.4 ist ein Beispiel eines Invalidationskonfliktes dargestellt. Dort wurde eine Situation im Kontext des UML Klassendiagramms beschrieben, wo bei der Zusammenführung aller Änderungen ein Vererbungskreis und damit ein invalides Modell entstünde. Im Sinne der Erkennung derartiger Konflikte muss zuerst eine Simulation der Zusammenführung der Änderungen beider Seiten durchgeführt werden und das Ergebnis mittels Validationsregeln auf Gültigkeit geprüft werden. Abschließend sind im Sinne der Benutzerfreundlichkeit eventuell auftretende Verstöße gegen die Validationsregeln auf die auslösenden Operationen zurückzuführen.

Die Validationsregeln eines Metamodells sind üblicherweise bereits von dessen Herstellern vorgegeben. Oft liegen diese sogar schon in einer formalen, maschinell überprüfbaren Form vor. Es ist daher wünschenswert, diese Spezifikationen wiederverwenden zu können, da die nachträgliche Definition dieser Regeln für mehrere Sprachen ein zeit- und wissensintensiver Prozess ist. EMF bietet mit dem EMF Validation Framework[1] bereits ein hervorragendes, generisches und erweiterbares Framework, um Validationsregeln für EMF Metamodelle zu definieren und Modellinstanzen auf diese hin zu prüfen. Dieses

[1] EMF Validation Framework: http://www.eclipse.org/modeling/emf/?project=validation

Framework wird bereits für den Großteil der EMF-basierten Metamodelle verwendet. Aus diesem Grund empfiehlt sich auch im Sinne der Wiederverwendung die Nutzung dieser Validierungsumgebung im Rahmen der Erkennung von Invalidationskonflikten.

Das EMF Validation Framework erweitert die Basisvalidationsfunktion des EMF-Kerns (*EValidator*), der ausschließlich über Codegenerierung arbeitet und nicht, wie das EMF Validation Framework, dynamisch erweiterbar ist. Validationsregeln können innerhalb des EMF Validation Frameworks in erster Linie auf zwei Arten definiert werden. Einerseits kann die Implementierung der Regeln in Java geschehen oder andererseits über die Verfassung von OCL-Constraints. Um eine Implementierung der Validationsregel in Java bereit zu stellen, muss die Klasse `org.eclipse.emf.validation.AbstractModelConstraint` abgeleitet und die Methode `IStatus : validate(IValidationContext ctx)` implementiert werden. Der `IValidateContext` enthält das zu validierende `EObjekt`. Nachdem die Validierung innerhalb der Java-Implementierung vorgenommen wurde, muss eine `IStatus`-Instanz zurückgegeben werden, die das Ergebnis des Validierungsvorganges beschreibt. Um auch die Verwendung eigener, benutzerdefinierter Constraint-Sprachen zu ermöglichen, kann ein Parser vom User für diese eigenen Sprachen registriert werden. Dies geschieht über den Extension-Point `org.eclipse.emf.validation.constraintParsers` innerhalb der *plugin.xml*. Unabhängig davon, in welcher Sprache die Regeln definiert wurden, müssen diese über den Extension-Point `org.eclipse.emf.validation.constraintProviders` innerhalb der *plugin.xml* eingebunden werden. Ein Beispiel für eine derartige Einbindung wird in Listing 9.3 aus der Eclipse-Online-Hilfe[2] demonstriert.

Wie aus diesem Listing ersichtlich ist, können Constraints durch Kategorien hierarchisch gruppiert werden. In Zeile 3 und 6 werden zwei Kategorien eingeführt, die in Zeile 12 bei der Constraint-Definition referenziert werden. In Zeile 13 und 25 wird je eine Validationsregel beschrieben. Die erste ist in Java verfasst, deren Implementierung in der Klasse `ThingReferences` vorgenommen wurde. Die Einführung der in OCL definierten Bedingung beginnt in Zeile 25, wobei der OCL-Constraint selbst im CDATA-Bereich (Zeile 40) direkt unter dem `constraint`-Element positioniert werden muss. Wichtig für die spätere Identifikation der Regeln ist vor allem der Status-Code (Zeile 14 bzw. 26) und die ID (Zeile 18 bzw. 31). Über das Attribut `severity` kann angegeben werden, ob es sich bei der Verletzung dieses Constraints um einen Fehler (`ERROR`) oder eine Warnung (`WARNING`) usw. handelt. Im Attribut `lang` wird festgelegt, in welcher Sprache die Bedingung implementiert wurde. Mögliche Werte sind, wie bereits erwähnt, `OCL` (Zeile 16) und `Java` (Zeile 28). Der Wert in `mode` (Zeile 29) beschreibt, ob ein Constraint im *Batch* oder *Live*-Modus arbeitet. Ist, wie in der ersten Constraint-Definition dieses Attribut nicht angeführt, wird der Standardwert *Batch* verwendet. Der *Live*-Modus überprüft die Änderungen innerhalb einer Transaktion direkt nach deren Durchführung während einer Änderungssession und soll damit ein sofortiges Feedback über die Validität ermöglichen. In Zeile 35 wird daher auch ein Event konfiguriert, nach dessen Auftreten der Constraint überprüft werden soll. Der *Batch*-Modus wird üblicherweise zur statischen Validierung nach einer gesamten Änderungssession verwendet und ist daher im Kontext der Konflikterkennung am Bedeutendsten. Im `target`-Element innerhalb eines Constraints im *Batch*-Mode kann nur eine Einschränkung auf einen Typ definiert werden, deren Instanzen mit der Bedingung geprüft werden sollen. Ein

[2] Eclipse-Online-Hilfe: http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.emf.validation.doc/references/extension-points/org_eclipse_emf_validation_constraintProviders.html

Listing 9.3: Definition eines Constraint-Providers.

```
<extension
      point="org.eclipse.emf.validation.constraintProviders">
   <category name="Example Constraints"
         id="com.example.validation">
      Description of the example constraints category.
      <category name="Foo" id="foo">
         Constraints for the EMF implementation of the "Foo"
            metamodel.
      </category>
   </category>
   <constraintProvider>
      <package namespaceUri="http:///com/example/foo.ecore" />
      <constraints categories="com.example.validation/foo">
         <constraint class="com.example.validation.constraints.
            ThingReferences"
             statusCode="11"
             severity="WARNING"
             lang="Java"
             name="Things References"
             id="thingReferences">
           <description>
             Things must not reference things that do not exist.
           </description>
           <message>"{0}" references non-existing things "{1}."</
              message>
           <target class="Thing" />
         </constraint>
         <constraint
             statusCode="12"
             severity="ERROR"
             lang="OCL"
             mode="Live"
             name="Things Must be Named"
             id="thingName">
           <description>Things must have names.</description>
           <message>Thing has no name.</message>
           <target class="Thing">
             <event name="Set">
               <feature name="name" />
             </event>
           </target>
           <!-- The OCL constraint expression. -->
           <![CDATA[
              name->notEmpty()
             ]]>
         </constraint>
      </constraints>
   </constraintProvider>
</extension>
```

auslösendes Event muss hier nicht angegeben werden.

Bevor Constraints verwendet werden können, müssen diese zuvor noch an einen *Client Context* gebunden werden. Der *Client Context* gibt an, welche Modelle oder Modellelemente mit welchen Bedingungen validiert werden sollen. So kann beispielsweise für einen Dateityp oder für einen ganzen Namensraum die Überprüfung einzelner Constraints oder ganzer Constraint-Kategorien konfiguriert werden. Diese Einbindung geschieht wiederum über einen Extension-Point namens `org.eclipse.emf.validation.constraintBindings` innerhalb der *plugin.xml*.

Da nun sowohl Konfiguration bezüglich der Ausführung der Constraints und die Constraints selbst gezeigt wurden, können diese auf die jeweiligen Modelle angewendet werden. Üblicherweise wird die Definition und Konfiguration dieser Constraints, wie schon erwähnt, bereits von den Herstellern der Metamodelle und Editoren selbst vorgenommen und zur Verfügung gestellt, um Instanzen der Metamodelle auf Validität prüfen zu können. Die Konflikterkennungskomponente muss daher nur noch, sofern die Validitätsregeln in der oben beschriebenen Form vorliegen, alle Änderungen beider Personen testweise durchführen und das Ergebnis auf diese Regeln hin überprüfen. Zur Durchführung der Änderungen in Form von EMF Compare Diffs kann das Merge Service von EMF Compare genutzt werden. Die Validierung des Ergebnisses selbst kann anschließend mittels dem Validation Service des EMF Validation Frameworks vorgenommen werden. Wie diese Validierung vorgenommen wird, ist im Listing 9.4 skizziert.

Listing 9.4: Validierung des Ergebnisses aller Änderungen.

```
// Das Resultat aller Aenderungen in resultObjects
List resultObjects = ...

// Erstelle und konfiguriere BatchValidator
IBatchValidator validator = (IBatchValidator)
    ModelValidationService.getInstance().newValidator(
    EvaluationMode.BATCH);

// Teste auch Live-Constraints
validator.setIncludeLiveConstraints(true);

// Fuehre die Validierung durch
IStatus results = validator.validate(resultObjects);

if (!results.isOK()) {
    // Es liegen ein oder mehrere Probleme vor...
    // Fuehre Probleme auf die verursachenden Operationen
    // zurueck und annotiere diese im Diff-Modell.
}
```

Bisher wurden die ersten beiden Aufgaben, also die testweise Zusammenführung aller Änderungen mittels EMF Compare Merge Service und die Prüfung der sprachspezifischen Validationsregeln unter der Verwendung des EMF Validation Frameworks, zur Erkennung von Invalidationskonflikten, besprochen. Dem User können somit bereits alle Probleme präsentiert werden, die bei der Zusammenführung aller Änderungen entste-

hen. Derartige Konflikte bleiben also nicht unerkannt und können bereits gemeldet werden. Zur Wiedererkennung der Invalidationskonflikte kann die ID bzw. der Statuscode und die Beschreibung des Constraints verwendet werden. Diese stellt in gewisser Weise die sprachspezifische Semantik der Regel dar. Wenn jedoch im Anschluss der Konflikterkennung eine intelligente Konfliktresolution durchgeführt werden soll, die dem User bereits adäquate Resolutionsstrategien vorschlägt, ist die alleinige Erkennung und Beschreibung der Probleme eventuell zu wenig. Auch im Sinne der Benutzerfreundlichkeit wäre es sinnvoll, die auftretenden Invalidationskonflikte den verursachenden Operationen innerhalb der Deltadokumente zuzuordnen.

Handelt es sich um Constraints im *Live*-Modus ist diese Zuordnung einfacher als bei Constraints im *Batch*-Modus. Schließlich ist das auslösende Event direkt innerhalb des Constraints mit dem `target`-Element definiert. Es muss dementsprechend jene Operation identifiziert werden, die auf das fehlerhafte Modellelement angewendet wurde und dem spezifizierten Event entspricht. Die Identifikation der verursachenden Operationen bei *Batch*-Modus-Constraints ist etwas komplexer und zum Teil ohne weitere Konfiguration sogar unmöglich. Das ermittelte *IStatus*-Objekt wird ja entweder durch einen OCL-Constraint oder durch eine Java-Implementierung erzeugt. Besonders im Fall der Java-Implementierung ist daher das Aussehen dieses *IStatus*-Objekts stark von der Implementierung abhängig. Es ist nicht vorhersehbar ob beispielsweise das Objekt als fehlerhaftes Objekt zurückgegeben wird, auf dem die verursachenden Operationen ausgeführt wurden. Im Beispiel des Vererbungskreises (siehe Abbildung 3.5) könnte beispielsweise je nach Implementierung eine Klasse als fehlerhaft ausgegeben werden, die gar nicht verändert wurde, sondern nur als neuer Supertyp einer anderen Klasse hinzugefügt wurde. Eine stets funktionierende Vorgehensweise zur Identifikation der verursachenden Operationen ist also zum Teil nicht möglich. Außerdem wird speziell bei komplexen Validationsregeln oft eine gesamte Situation als fehlerhaft beschrieben. Bei generischer Betrachtung dieser *IStatus*-Meldung können damit nicht immer eindeutig die tatsächlich verursachenden Operationen ermittelt werden, sondern nur alle Operationen, die die betreffenden, fehlerhaften Modellelemente verändert haben.

Wenn die exakte Ermittlung der verursachenden Operationen tatsächlich benötigt wird, ist es also nötig, für einige, sprachspezifische Validationsregeln zusätzlich Algorithmen zu implementieren, die aus einem Deltadokument die jeweiligen, verursachenden Operationen ableiten können. Eine vernünftige Lösung dieses Problems ist offensichtlich nicht trivial und konnte daher nicht im Zuge dieser Arbeit vollständig berücksichtigt werden. Zum aktuellen Zeitpunkt der Überlegungen scheint eine Integration dieser Aufgabe innerhalb der Definition der Validationsregeln sinnvoll zu sein. Somit würde die Auswahl der verursachenden Operationen dort abgebildet werden, wo auch die Validation selbst implementiert ist. Dies wäre daher eine Konzentration der Logik an jenem Ort, wo ohnehin das sprachspezifische Wissen zur Erkennung der Validationsprobleme nötig ist. An dieser Stelle wird auch entschieden, welche Objekte als fehlerhaft zu melden sind. Allerdings würde so von den Metamodell- und Editor-Herstellern verlangt werden, das verwendete Diff-Metamodell anzuwenden. Schließlich muss in Instanzen dieses Metamodells nach der verursachenden Operation gesucht werden. Sofern dieses Diff-Metamodell quelloffen und innerhalb der Community akzeptiert ist, sollte dies zumindest kein Ausschlusskriterium für die Lösung sein. Allerdings ist in jedem Fall die Qualität der Konfliktresolution von jener der von dritten Parteien implementierten Ermittlung der verursachenden Operationen abhängig. Eine Lösung für dieses Problem bedarf weiterer Überlegungen.

10 Conclusio

In der vorliegenden Arbeit wurden bestehende Arbeiten und neu entwickelte Konzepte und Implementierungen für den Modellvergleich, die Repräsentation der Unterschiede zwischen zwei Modellversionen und die Konflikterkennung zwischen zwei parallel durchgeführten Änderungsfolgen auf eine Modellversion vorgestellt. Hierbei wurde stets versucht die Grundfunktionalität unabhängig von Sprache und Editor zu halten und diese durch einfach zu erstellende, deskriptive Artefakte um sprachspezifische Aspekte vom User erweiterbar zu machen.

Es wurde mehrmals die große Bedeutung des exakten und vollständigen Modellvergleichs im Rahmen der Konflikterkennung und in weiterer Folge eines Versionierungssystems für Modelle unterstrichen und gezeigt, wie dieser in die Realität umgesetzt werden kann. Diese Arbeit argumentierte, dass für einen breiten und damit sprach- und editorunabhängigen Einsatz eines derartigen Systems der statusbasierte Vergleich viele Vorteile gegenüber dem editorbasierten Tracking von Operationen, also dem operationsbasierten Vergleich, aufweist. Es wurden Implementierungen und Konzepte vorgestellt, die den statusbasierten Ansatzes nutzen und erweitern, um einen exakten Modellvergleich und eine qualitätiv hochwertige Konflikterkennung zu ermöglichen, die den Anforderungen für eine anschließende, intelligente Konfliktresolution genügen. Hierfür können noch einmal die folgenden, vorgestellten Techniken oder Konzepte hervorgehoben werden:

- **Transiente IDs** – Um einen zuverlässigen und exakten Match von Modellelementen zu ihrem Ursprung zu erlauben, muss auf die Verwendung heuristischer Methoden verzichtet werden. Zur Gewährleistung der vollständigen Editorunabhängigkeit können auch den Editoren nicht die Verantwortung über die Vergabe von IDs, die für den späteren Match verwendet werden, übertragen werden. Es ist die Aufgabe des Versionierungssystem selbst, IDs zu diesem Zweck zu vergeben. Durch die Annotation der Elemente mit eindeutigen Attributen zum Zeitpunkt des *Checkouts*, wo die Modellversionen exakt identisch sind und damit ein Match einfach durchführbar ist, kann eine Wiedererkennung der ursprünglichen Elemente auch nach starker Veränderung gesichert werden, ohne sich in eine Abhängigkeit von den eingesetzten Editoren zu begeben.
- **Erkennung zusammengesetzter Operationen** – Wenn der Vergleich zweier Modellversionen auf generischer Basis durchgeführt wird, sind auch die erkannten Operationen generischer Natur. Diese sind oftmals komplex und abstrakt, sodass *eine* Änderung des Users als Folge *mehrerer* atomarer Änderungen auf einer Metaebene dargestellt sind. Die Operationen entsprechen daher nicht immer der Sprache des Users, was deren weitere Verarbeitung für den User erschwert. Dieser Aspekt, sowie die Minimierung der späteren Konfliktmeldungen, sind Gründe für die Notwendigkeit, zusammengesetzte Operationen und Refactorings innerhalb einer generischen Operationsfolge vom System erkennbar zu machen. Da diese

zusammengesetzten Operationen und Refactorings abhängig von der eingesetzten Modellierungssprache sind, müssen diese vom User selbst abgebildet werden können. Das Kapitel 8 stellt eine beispielgetriebene Methode vor, derartige Operationen und Refactorings zu definieren und in das System einzubinden, um es in weiterer Folge zu befähigen, diese Operationen innerhalb eines Deltadokuments zu erkennen und anzuzeigen.

- **Erkennung sprachspezifischer Konflikte** – Neben der Erkennung generischer Konflikte, die auch ohne sprachspezifisches Wissen erkennbar sind, wurden in Kapitel 9 Möglichkeiten besprochen, auch sprachspezifische Konflikte, wie beispielsweise widersprüchliche Intentionen und Invalidationskonflikte, aufzudecken. Die spezifischen Artefakte werden wiederum vom User in Form von Definitionen widersprüchlicher Operationsfolgen bzw. Abbildungen metamodellspezifischer Validationsregeln in das System eingebunden. Diese Artefakte befähigen das System Konflikte erkenntlich zu machen, die durch die metamodellunabhängige Analyse nicht ersichtlich gewesen wären.

Im Zuge dieser Arbeit konnten allerdings nicht alle Aspekte und Herausforderungen adressiert werden, die die Konflikterkennung bei der Versionierung von Modellen aufzeigt. Beispielsweise müssen im Bereich der beispielgetriebenen Definition zusammengesetzter Operationen noch weitere Überlegungen angestellt werden, um tatsächlich alle Refactorings in benutzerfreundlicher Weise abbildbar zu machen. Eine genaue Beschreibung bestehender Einschränkungen und erste Lösungsvorschläge wurden im Zuge des Kapitels 8 besprochen. Ebenso sollte der vorgestellte Ansatz zur Abbildung widersprüchlicher Operationen aus Kapitel 9 nur als grundlegende Idee, deren detaillierte Ausarbeitung und Evaluierung noch ausständig ist, betrachtet werden. Für die Erkennung der verursachenden Operationen bei Invalidationskonflikten konnte ebenfalls im Zuge dieser Arbeit keine allgemein einsetzbare Methode gefunden werden.

Einige Aspekte blieben im Rahmen dieser Arbeit gänzlich ungenannt. Beispielsweise konnte nicht auf die Möglichkeit einer Echtzeit-Konflikterkennung eingegangen werden, die den User zu dem Zeitpunkt vor einem Konflikt warnt, in dem er eine Änderung vornimmt, wie dies zum Beispiel beim kollaborativen Editieren von Textdokumenten oft gemacht wird. Außerdem wurde nicht weiter auf die Möglichkeit eingegangen, sprachspezifische Definitionen über einen Communityserver, ähnlich einem Wiki, wartbar zu machen und allen AnwenderInnen des Versionierungssystems und der jeweiligen Modellierungssprache zur Verfügung zu stellen. Auf diese Weise könnte der Aufwand, sprachspezifische Aspekte abzubilden und einzubinden, durch Open-Source- und Community-Effekte minimiert werden.

Dennoch konnten im Zuge dieser Arbeit einige Probleme im Rahmen der Konflikterkennung bei der Versionierung von Modellen gelöst und Argumente für oder gegen gewisse Methoden zusammengetragen werden. Die vorliegende Arbeit sollte daher eine umfassende Grundlage bilden, um davon ausgehend weitere Forschungen und Entwicklungen in diesem Bereich anzustellen und um letztlich die Implementierung eines in breiten Umfeld einsetzbaren Konflikterkennungssystems zu bewerkstelligen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] *Concurrent Versions System Product Description*. Website. http://www.nongnu.org/cvs/; abgerufen am 29.08.2008.

[2] *Eclipse Platform*. Website. http://www.eclipse.org; abgerufen am 29.12.2008.

[3] *Subversion Produktbeschreibung*. Website. http://subversion.tigris.org; abgerufen am 29.08.2008.

[4] Alanen, Marcus und Ivan Porres: *Difference and Union of Models*. In: *UML 2003 – The Unified Modeling Language*, Band 2863 der Reihe *Lecture Notes in Computer Science*, Seiten 2–17. Springer-Verlag, Oct 2003.

[5] Altmanninger, Kerstin, Alexander Bergmayr, Wieland Schwinger und Gabriele Kotsis: *Semantically Enhanced Conflict Detection between Model Versions in SMoVer by Example*. In: *Proceedings of the 1st International Workshop on Semantic-Based Software Development (SBSD) in conjunction with OOPSLA Conference*, 2007.

[6] Blanc, Xavier, Isabelle Mounier, Alix Mougenot und Tom Mens: *Detecting Model Inconsistency through Operation-based Model Construction*. In: *Proceedings of the 30th International Conference on Software engineering*, Seiten 511–520. ACM, 2008.

[7] Brun, Cédric und Laurent Goubet: *EMF Compare Dokumentation*, 2008.

[8] Brun, Cédric und Alfonso Pierantonio: *Model Differences in the Eclipse Modeling Framework*. UPGRADE, IX(2):29–34, 2008.

[9] Budinsky, Frank, David Steinberg, Ed Merks, Raymond Ellersich und Timothy J. Grose: *Eclipse Modeling Framework – A developers guide*. The Eclipse Series. Pearson Education, Inc., 2003.

[10] Buffenbarger, Jim: *Syntactic Software Merging*. In: *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*, Seiten 153–172. Springer-Verlag, 1995.

[11] Cicchetti, Antonio, Davide Di Ruscio und Alfonso Pierantonio: *Composition of Model Differences*. In: *Proceedings of European Workshop on Composition of Model Transformations (CMT), colocated with the European Conference on Model-Driven Architecture (ECMDA)*, 2006.

[12] Cicchetti, Antonio, Davide Di Ruscio und Alfonso Pierantonio: *A Metamodel Independent Approach to Difference Representation*. Journal of Object Technology, TOOLS EUROPE 2007(10):165–185, 2007.

[13] Conradi, Reidar und Bernhard Westfechtel: *Towards a Uniform Version Model for Software Configuration Management*. In: *Proceedings of the SCM-7 Workshop on System Configuration Management*, Seiten 1–17. Springer-Verlag, 1997.

[14] Conradi, Reidar und Bernhard Westfechtel: *Version Models for Software Configuration Management.* ACM Computing Surveys, 30(2):232, 1998.

[15] Cypher, Allen, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers und Alan Turransky (Herausgeber): *Watch What I Do: Programming by Demonstration.* MIT Press, 1993.

[16] Dart, Susan: *Spectrum of Functionality in Configuration Management Systems.* Technischer Bericht, 1990. CMU/SEI-90-TR11 ESD-90-TR-212. http://www.sei.cmu.edu/legacy/scm/abstracts/abscm_spectrum_of_func_TR11_90.html.

[17] Dig, Danny, Can Comertoglu, Darko Marinov und Ralph Johnson: *Automated Detection of Refactorings in Evolving Components.* In: *Proceedings of the 20th ECOOP, Lecture Notes in Computer Science*, Band 4067, Seiten 404–428. Springer-Verlag, 2006.

[18] Dig, Danny, Kashif Manzoor, Ralph Johnson und Tien N. Nguyen: *Refactoring-Aware Configuration Management for Object-Oriented Programs.* In: *Proceedings of the 29th International Conference on Software Engineering*, Seite 427. IEEE Computer Society, 2007.

[19] Edwards, W. Keith: *Flexible Conflict Detection and Management in Collaborative Applications.* In: *Proceedings of the 10th Annual ACM Symposium on User interface Software and Technology*, Seiten 139–148. ACM, 1997.

[20] Estublier, Jacky, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy und Darcy Wiborg-Weber: *Impact of Software Engineering Research on the Practice of Software Configuration Management.* ACM Transactions on Software Engineering and Methodology, 14(4):383–430, 2005.

[21] Fowler, Martin, Kent Beck, John Brant, William Opdyke und Don Roberts: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, June 1999.

[22] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Professional, 1995.

[23] Hunt, J. W. und M. D. McIlroy: *An Algorithm for Differential File Comparison.* Technischer Bericht CSTR 41, 1976.

[24] Jouault, Frédéric, Jean Bézivin und Atlas Team: *KM3: A DSL for Metamodel Specification.* In: *Proceedings of 8th FMOODS, Lecture Notes in Computer Science*, Seiten 171–185. Springer-Verlag, 2006.

[25] Jouault, Frederic und Ivan Kurtev: *Transforming Models with ATL.* In: *Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science*, Seiten 128–138. Springer-Verlag, 2006.

[26] Kögel, Maximilian: *Towards Software Configuration Management for Unified Models.* In: *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models – CVSM 08*, Seite 19. ACM, 2008.

[27] Lédeczi, Ákos, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle und Gábor Karsai: *Composing Domain-Specific Design Environments.* Computer, 34(11):44–51, 2001.

[28] Levenshtein, V. I.: *Binary Codes Capable of Correcting Deletions, Insertions, and Reversals.* Cybernetics and Control Theory, 10(8):707–710, 1966.

[29] Lin, Yuehua, Jeff Gray und Frédéric Jouault: *DSMDiff: A Differentiation Tool for Domain-Specific Models.* European Journal of Information Systems, 16(4):349–361, 2007.

[30] Lippe, Ernst und Norbert van Oosterom: *Operation-based Merging.* In: *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, Band 17, Seiten 78–87. ACM, 1992.

[31] Mens, Tom: *A State-of-the-Art Survey on Software Merging.* IEEE Transactions on Software Engineering, 28(5):449–462, 2002.

[32] Mens, Tom, Gabriele Taentzer und Olga Runge: *Detecting Structural Refactoring Conflicts Using Critical Pair Analysis.* Electronic Notes in Theoretical Computer Science, 127(3):113–128, 2005.

[33] Ohst, Dirk, Michael Welle und Udo Kelter: *Differences Between Versions of UML Diagrams.* In: *ESEC / SIGSOFT FSE*, Seiten 227–236. ACM, 2003.

[34] OMG: *Model Object Facility (MOF) Core specification.* Specification Version 2.0, Object Management Group, January 2006.

[35] OMG: *Object Constraint Language.* Specification Version 2.0, March 2006.

[36] Parnas, D. L.: *On the Criteria to be used in Decomposing Systems into Modules.* Communications of the ACM, 15(12):1053–1058, 1972.

[37] Störrle, Harald: *A Formal Approach to the Cross-language Version Management of Models.* In: *Proceedings of 5th Nordic Workshop on Model Driven Engineering*, Seiten 83–97, 8 2007.

[38] Wang, Yuan, David J. DeWitt und Jin yi Cai: *X-Diff: An Effective Change Detection Algorithm for XML Documents.* In: *Proceedings of the 19th International Conference on Data Engineering*, Seiten 519–530. IEEE Computer Society, 2003.

[39] Wolf, Timo: *Rationale-based Unified Software Engineering Model.* Doktorarbeit, Technische Universität München, 2007.

[40] Yuehua Lin, Jing Zhang, Jeff Gray: *Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development.* In: *Proceedings of the Object Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM, 2004.