FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Sprachspezifische Modellversionierung für SysML

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Stefan Schindler, Bsc
Matrikelnummer 0020882

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel

Wien, 1. August 2018

Stefan Schindler	Gerti Kappel

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Language specific model versioning for SysML

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Stefan Schindler, Bsc
Registration Number 0020882

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel

Vienna, 1st August, 2018

Stefan Schindler	Gerti Kappel

Erklärung zur Verfassung der Arbeit

Stefan Schindler, Bsc
Fürstenallee 8a, 5020 Salzburg

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, 1. August 2018

Stefan Schindler

Danksagung

Mein innigster und herzlichster Dank gilt meiner Freundin und Lebensgefährtin Frau Dr. Sophie Gensluckner. Nur durch ihre liebevolle Unterstützung und ihre schier endlose Geduld in allen Phasen meines Studiums, nicht zuletzt aber auch ihre finanzielle Unterstützung, wurde diese Arbeit überhaupt erst möglich.

Auch möchte ich mich recht herzlich bedanken bei Frau Prof. Dr. Gerti Kappel für die Betreuung dieser Arbeit sowie bei Dr. Philip Langer für seine wertvolle Begleitung, Unterstützung und Hilfe, aber auch seine Geduld während der Erstellung dieser Arbeit.

Weiters möchte ich mich bedanken bei meiner Familie sowie meinen Freunden und Studienkollegen, für die Unterstützung und den Zuspruch aber auch die notwendige Zerstreuung während meines gesamten Studiums. Ein besonderer Dank gilt meinem Freund Mag. Florian Wachter für das Korrekturlesen dieser Arbeit.

Kurzfassung

Die Modellversionierung wird durch die wachsende Popularität und Verbreitung von domänenspezifischen Modellierungssprachen ein immer wichtigerer Bestandteil in der Softwareentwicklung, wenn dafür modellbasierte Methoden verwendet werden. Im selben Kontext kann die modellbasierte Systementwicklung (MBSE) genannt werden. Modelle und die verteilte, kollaborative Arbeit damit, gewinnen immer mehr an Stellenwert. SysML ist eine domänenspezifische Modellierungssprache zur Systementwicklung.

Die quelloffene Entwicklungsplattform Eclipse bietet mit dem Eclipse Modeling Framework eine hervorragende Plattform für die Softwareentwicklung mit modelgetriebenen Methoden. Die Initiative „Collaborative Modeling with Eclipse“ bietet auf Basis der genannten Technologien und dem Modellierungswerkzeug Papyrus eine Modellierungsplattform an, die es ermöglichen soll, in Teams komfortabel an und mit Modellen zu arbeiten.

Diese Arbeit versucht die Ergebnisse, die bei Vergleichen von SysML Diagrammen in der genannten Modellierungsplattform erzielt wurden, zu verbessern. Dazu wurde eine Reihe von Modellvergleichen durchgeführt und analysiert, die einige Schwachpunkte in der Differenzerkennung und Visualisierung aufgedeckt haben. Durch die Entwicklung geeigneter Erweiterungen für das EMF Compare Plugin ist es gelungen diese Schwachpunkte zu beheben.

Abstract

With the increasing popularity and usage of domain specific modeling languages, model versioning is be getting more and more important in software development, especially when model-based development methods come into play. This is also an important aspect for model-based systems engineering. Models and the distributed, collaborative development of models are constantly gaining significance. SysML is a domain-specific modeling language for systems engineering. The open source development platform Eclipse and its outstanding Eclipse Modeling Framework are an excellent base for model-driven software development. The initiative „Collaborative Modeling with Eclipse“ provides a modeling platform, founded on the named technologies and the modeling tool Papyrus, that tries to enable the comfortable working with models in teams. This work tries to improve the results, retrieved from the modeling platform when comparing SysML models. From an analysis of carried out model comparisons, several weak spots in differencing and visualization of those differences were found. Through the development of appropriate extensions, the identified weak spots could be addressed.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Motivation

Versionierung oder besser Versionsverwaltung ist aus der Softwaretechnik nicht mehr wegzudenken. Sobald die Entwicklung einer Software einen gewissen Grad an Aufwand und Komplexität übersteigt, muss im Team gearbeitet werden. Eine Entwicklungsabteilung kann aus zwei oder mehreren Entwicklern im selben Büro, oder aber auch aus mehreren Teams bestehen, die über die ganze Welt verteilt arbeiten. Die gemeinsame Arbeit an einem Projekt bedarf jedenfalls, im Kleinen wie im Großen, einer geplanten, strukturierten und systematischen Vorgehensweise.

Die Arbeit an Softwareprojekten wird hierbei durch das Softwarekonfigurationsmanagement (SCM[1]), unterstützt. Ein wichtiger Teil von SCM ist die Versionierung mittels eines Versionsverwaltungssytems (VCS[2]). Hauptaufgaben von VCS sind unter anderem die Protokollierung und Archivierung des Änderungsverlaufs im Projekt, zumeist von mehreren parallelen Zweigen, sowie die Zusammenführung von Änderungen und Behebung von Konflikten, die bei konkurrierenden Änderungen auftreten.

Egal, ob ein dokumentenlastiger, schwergewichtiger Softwareprozess oder eine agile, leichtgewichtige Softwareentwicklungsmethode eingesetzt wird, Modelle und Diagramme dienen oft nur der Veranschaulichung und Dokumentation [39, 158, 198]. Dieser Umstand ändert sich seit der Entwicklung und Etablierung modellbasierter Methoden wie MDE[3]. Hier werden Modelle zum integralen Bestandteil des gesamten Softwareentwicklungsprozesses. Sie stellen als Systemabbildung die funktionelle Basis dar, die von Transformatoren und Generatoren in ausführbare Software übersetzt wird.

[1] Software Configuration Management
[2] Version Control System
[3] Model Driven Engineering

Als Standard für Softwaremodellierung hat sich über Jahre UML[4] etabliert. Die Sprache wurde von der OMG[5], einem Industriekonsortium dem viele namhafte IT-Firmen angehören, standardisiert und wird laufend erweitert und weiterentwickelt. Da UML zwar sehr umfangreich und vielseitig einsetzbar ist, aber dennoch niemals allen Anforderungen oder Problemstellungen genügen kann, besteht die Möglichkeit Erweiterungen für UML in Form von UML-Profilen zu erstellen.

Eine solche Erweiterung stellt SysML[6] dar. SysML dient der Systementwicklung, hier vor allem dem modelbasierten Ansatz der Systementwicklung (MBSE[7]), und wurde ebenfalls von der OMG standardisiert [195]. Es erweitert eine Teilemenge von UML um Spezifika der Systementwicklung und fügt einige Konzepte der Systementwicklung neu hinzu [74], z.B. Anforderungsmodellierung. Durch die interdisziplinären Anforderungen bei der Systementwicklung ist ein koordiniertes, kooperatives Arbeiten unumgänglich, eine Versionspflege und Verwaltung ist unbedingte Pflicht.

Alle führenden VCS, wie z.B. Subversion oder Git, arbeiten zeilenorientiert auf Textbasis und funktionieren daher bei Quellcode ausgezeichnet. Auch UML oder SysML Modelle werden meist als Text gespeichert. Hierfür bietet die OMG ausdrücklich das standardisierte Format XMI[8] an, um z.B. den Datenaustausch zwischen Anwendungen möglichst reibungslos zu gestalten. Es zeigte sich allerdings, dass für die Modellversionierung eine Beschränkung auf zeilenorientierte Textvergleiche nicht ausreicht [15, 102].

Ein Grund dafür ist über die Datenstruktur von Modellen gegeben. Diese werden als Graph abgebildet und serialisiert gespeichert, z.B. im eben erwähnten Format XMI. Bei zeilenweisen Textvergleichen gehen dann aber die von der graphbasierten Modellstruktur transportierten, syntaktischen sowie semantischen Informationen verloren [17, 33, 34]. Beispielsweise würden bei zeilenweisem Textvergleich zwei identische Elemente in unterschiedlicher Reihenfolge fälschlicherweise als Änderung erkannt werden [17]. Aufgrund der graphbasierten Datenstruktur können einzelne Änderungen im Modell auch zu mehreren Änderungen in der serialisierten Darstellung führen [33].

Ein weiteres Problem könnte sinngemäß als Bruch in der Darstellung oder als Darstellungslücke bezeichnet werden [17, 33]. Wenn es bei der Versionierung von Quellcode zu Konflikten kommt, dann werden die entsprechenden konkurrierenden Versionen, also Quelltextzeilen, zur Bereingung angezeigt. Entwicklung und Konfliktbehebung bedienen sich derselben Darstellung. Bei der Modellversionierung sieht das oft anders aus. Zur Modellierung werden üblicherweise grafische Editoren oder zumindest Editoren mit Baumdarstellung eingesetzt [33, 34, 199]. In beiden Fällen wird mit konkreter Syntax gearbeitet. Zur Versionierung gelangt allerdings die graphbasierte Repräsentation der abstrakten Syntax des Modells [34]. Genau hier kommt es zum Bruch oder zur Lücke, wenn der Anwender bei der Konfliktbehebung mit einer anderen, ungewohnten Darstellung des

[4] Unified Modeling Language
[5] Object Management Group
[6] Systems Modeling Language
[7] Model Based Systems Engineering
[8] XML Metadata Interchange

Modells konfrontiert wird. Die direkte Bearbeitung durch den Anwender ist bei Formaten wie XMI auch gar nicht vorgesehen [17, 199].

Darüber hinaus besteht ein gravierender Unterschied zwischen einem Modell und der grafischen Repräsentation als Diagramm. Ein Diagramm enthält andere Informationen als das dargestellte Modell. Dazu zählen etwa Layoutinformationen über Position und Aussehen von Elementen [34]. Ist der Vergleich von Modellen und Diagrammen bereits aufgrund der graphbasierten Datenstrukturen aufwändiger als reine Textvergleiche, so verkompliziert sich die Lage durch die zusätzlichen Informationen die zur Diagrammdarstellung notwendig sind noch mehr. Um bei der Modellversionierung zufriedenstellende Ergebnisse erzielen zu können, müssen die Informationen von Modell und Diagramm gemeinsam einer detaillierten Betrachtung unterzogen werden [34].

Als Ansatz zur Lösung der Probleme des Modellvergleichs und der Modellzusammenführung im Umfeld der Entwicklungsplattform Eclipse, wurde EMF Compare [187, 38] entwickelt. EMF Compare ist ein erweiterbares Plugin für die Eclipse IDE, aufbauend auf dem Eclipse Modeling Framework (EMF [176]). Es stellt einerseits ein Framework für den Modellvergleich zur Verfügung und ist andererseits ein Werkzeug um Diagramme zu vergleichen und zusammenzuführen. EMF Compare wurde mit Bedacht auf die beschriebene Problematik entworfen und erstellt Vergleiche auf Modellebene und nicht auf Zeilenebene. Durch Spezialisierung des prinzipiell generischen Modellvergleichs erkennt EMF Compare z.B. Datenkonstrukte, die nur der Diagrammdarstellung dienen. Um hier nachvollziehbare und intuitive Ergebnisse liefern zu können, also z.B. eine grafische Darstellung der durchgeführten Änderungen im Diagramm, werden Änderungen im Modell bzw. generell der Datenstruktur gruppiert und „maskiert“. Letztlich ist EMF Compare aber lediglich ein Hilfsmittel zur Versionierung von Modellen und Diagrammen. Die eigentliche Versionierung erfolgt neuerdings häufig mit EGit, einem Eclipse Plugin, das Unterstützung für das verteilte Versionsverwaltungssystem Git bietet. Um einfach und schnell Modelle und Diagramme in unterschiedlichen Sprachen, etwa UML oder SysML, zu erstellen, bietet Papyrus, ebenfalls als erweiterbares Plugin, eine grafische Modellierungsumgebung für Eclipse.

Die Initiative „Collaborative Modeling with Eclipse“ [181] vereint nun Eclipse und Papyrus mit speziell angepassten Versionen von EMF Compare und EGit um eine gemeinsame Basis für effizientes kollaboratives Arbeiten mit Modellen und Diagrammen zu ermöglichen. Diese speziellen Versionen von EMF Compare und EGit sind notwendig um bei der Modellversionierung untereinander angemessen kooperieren zu können. Die Unterstützung von UML Diagrammen wird über entsprechende Plugins bereitgestellt. SysML stellt eine Erweiterung von UML dar und deshalb kommen diese Plugins auch hier zum Einsatz. Da SysML aber einige neue Diagrammarten und Konzepte der Systementwicklung mitbringt, funktioniert die Erkennung von Elementen bzw. Differenzen noch nicht komplett, bzw. nicht in zufriedenstellendem Ausmaß. Die Problemstellung der Diplomarbeit ist das Erarbeiten einer verbesserten SysML Unterstützung zur Modellversionierung mit Papyrus, EMF Compare und EGit. Der Fokus liegt hier auf einer korrekten bzw. einer für den Benutzer einfach nachvollziehbaren Darstellung von Differenzen im Versionierungsprozess.

Als thematische Fundamentierung dieser Arbeit dienen die Kapitel zu den Grundlagen und dem Stand der Wissenschaft, worin die Ergebnisse der umfassenden Literaturrecherche dargelegt wurden. Mit dieser Arbeit soll nichts Neues erfunden werden, sondern bestehendes Wissen und bekannte Techniken angewendet werden. Zu diesem Zweck soll die Funktionsweise bestehender Erweiterungen analysiert und mit dem gewonnenen Wissen der Analyse eine geeignete Vorgehensweise reproduziert werden. Das Ergebnis der Arbeit sollen prototypische Umsetzungen zur Erfüllung der funktionellen Anforderungen sein, die sich aus der Analysephase ergeben.

1.2 Methodisches Vorgehen

Die in Anlehnung an die Design Science Research (DSR) Richtlinien von Hevner et al. [83] entwickelte Methode, Design Science Research Methodology (DSRM) von Peffers et.al. [149], schreibt die folgenden Phasen für eine wissenschaftliche Arbeit vor:

1. Problemidentifikation und Motivation
2. Definition der Ziele für eine Lösung
3. Design und Entwicklung
4. Demonstration
5. Evaluation
6. Kommunikation

Da diese Phasen eine passendere Zuordnung zu der vorliegenden Arbeit ermöglichen, als die DSR Richtlinien und DSRM auf DSR basiert, also der kontextuelle Rahmen nicht verlassen wird, wurde diese Methode als Vorlage zur Ausarbeitung des Themas gewählt.

1.3 Aufbau der Arbeit

Die Grundlagen der Themengebiete, die mit dieser Arbeit zusammenhängen, werden in Kapitel 2 diskutiert. In Kapitel 3 wird der Stand der Wissenschaft erläutert. Das Kapitel 4 beschreibt die Analyse zur Problemerhebung beim Vergleichen von SysML-Diagrammen. In Kapitel 5 werden die Zielvorstellungen definiert, der Lösungsweg beschrieben, sowie die Ergebnisse vorgestellt. Und das abschließende Kapitel 6 ist der Reflexion und Diskussion gewidmet. Der Anhang A enthält das EMF Compare Differenzmodell. In Anhang B finden sich Erläuterungen zur durchgeführten Analyse und der Anhang C enthält das Analyseprotokoll in Tabellenform.

KAPITEL 2

Grundlagen

Das Wissen um die Grundlagen von Methoden, Praktiken und Technologien ermöglicht und unterstützt das Verständnis und ist hilfreich bei der Anwendung und Umsetzung. In diesem einführenden Kapitel sollen daher die vielfältigen Themenbereiche, die das Fundament bilden, auf dem diese Arbeit aufbaut, in Form einer umfassenden Literaturrecherche eingehend betrachtet und besprochen werden. Den Anfang macht der überaus vielschichtige Begriff des Modells, worauf eine Betrachtung des Metamodells folgt. Danach werden Modellierungssprachen und modellbasierte Entwicklungsmethoden besprochen und im Anschluss der Bogen von der Versionierung zur Modellversionierung gespannt. Das Kapitel schließt mit der Vorstellung der verschiedenen Softwareprodukte, die bei der Umsetzung dieser Arbeit Verwendung fanden.

2.1 Modellbegriffe

Die Bedeutung des Begriffs „Modell“ und die Tätigkeit des Modellierens sind Inhalt zahlreicher Veröffentlichungen. Darunter fallen einerseits ganz allgemein theoretische Überlegungen [25, 110, 86, 172, 196, 156, 133, 132, 120], andererseits aber auch sehr spezifische, auf einzelne Wissenschaftszweige zugeschnittene Betrachtungen [99, 150, 118, 121, 162, 182, 82, 41]. Auf die Wichtigkeit einer Definition des Modellbegriffs wird in der Literatur immer wieder hingewiesen. Mindestens so oft wird auch darauf hingewiesen, wie vielfältig der Begriff ist und wie unvereinbar gewisse Vorstellungen und Positionen dazu sind. Im Folgenden soll ein kurzer Blick auf verschiedene Aspekte von Modellen, sowie unterschiedliche Sichtweisen, Meinungen und Theorien geworfen werden.

2.1.1 Was ist ein Modell?

Abhängig von Erfahrung und Bildung kann das Wort „Modell“ unterschiedlichste Assoziationen hervorrufen. Im Alltag umgeben uns Modelle äußerst unterschiedlicher Ausprägungen, unser Handeln ist geprägt von Modellen, auch benutzen wir ständig Modelle.

Modell...	...von etwas	...für etwas
Perspektive	Herstellungsperspektive	Anwendungsperspektive
Arbeitsmittel	Medium	Werkzeug
Blickrichtung	retrospektiv	prospektiv
Zweck	beschreibend, erklärend	voraussagen
Einsatz	Fachwissensvermittlung	Erkenntnisgewinnung

Tab. 2.1: Bedeutungsdimensionen von Modellen [99]

Jedoch geschieht dies wohl meist, ohne dass wir es registrieren bzw. die Modelle als solche identifizieren [134, 41, 156, 118]. Das macht es schon im Ansatz schwierig, nur von dem was wir als Modell ansehen oder verstehen, einen Modellbegriff abzuleiten.

Das Wort „Modell“ stammt vom lateinischen Wort „modulus“ ab und bedeutet Maß, Maßstab[1]. Schlägt man „Modell“ in der Onlineausgabe des Duden [3] nach, so werden mehr als zehn Bedeutungserklärungen und etwas mehr als fünfzig Synonyme angezeigt. Mahr [121] schreibt, es wird vermutet, das Wort „Modell“ sei ein Homonym[2] und dass es nur zufällig in vielen verschiedenen Zusammenhängen benutzt werde. Das Wort „Modell“ ist also durch die Vielzahl an Bedeutungen zu einem sehr schwammigen, bedeutungselastischen Begriff geworden. Mögliche Bedeutungsdimensionen wurden von Koch et al. [99] anschaulich zusammengefasst, siehe Tab. 2.1.

Umgangssprachlich könnten als beschreibende Merkmale für Modelle z.B. vereinfachend, repräsentierend, von verkleinertem Maßstab, zeichenhaft oder abstrahierend genannt werden [121]. Diese Merkmale sind zwar für sehr viele Modelle zutreffend, jedoch können damit niemals alle Modelle zufriedenstellend charakterisiert werden. Als einfaches Gegenbeispiel könnte ein Atommodell herangezogen werden, dabei handelt es sich um keine Verkleinerung, sondern im Gegenteil immer um extreme Vergrößerung.

Über das Beispiel des Atommodells wird der nächste, nicht unwichtige Aspekt aufgezeigt, nämlich die Frage ob ein Modell korrekt ist. Als anschauliches Beispiel für die möglichen Auswirkungen eines falschen Modells wird gern der Aderlass herangezogen [156, 41]. Diese mittelalterliche Praktik basierte auf einem falschen Modell der Blutzirkulation und führte nicht selten zu Tod. Aus schlechten oder falschen Modellen können also sehr schnell schlechte oder falsche Ergebnisse resultieren. Eine allgemeine Festlegung dessen, was ein Modell ist könnte dabei helfen, Modelle zu erkennen, zu differenzieren, zu bewerten, zu entwickeln und anzuwenden. Die Frage, ob Modelle richtig oder falsch sind, wird von Mittelstraß [132] übrigens als nicht zutreffend abgelehnt. Er differenziert Modelle als, entweder „anwendungsstark“ oder „anwendungsschwach“, je nachdem ob sie „leisten, wozu sie entwickelt wurden“ oder eben nicht.

[1] Eine sehr ausführliche Auseinandersetzung mit der Herkunft und Geschichte des Modellbegriffs bietet Müller [133].

[2] laut Duden Online [1]: gleichlautend, aber von unterschiedlicher Bedeutung

Der Mensch benutzt Modelle schon sehr lange, vielleicht seit dem Beginn der Entwicklungsgeschichte [133]. Ludewig [119] sagt, dass Modelle nicht erfunden wurden, sondern existieren seit der Mensch existiert. Er geht dann noch einen Schritt weiter, indem er das Modellieren als eine dem Menschen angeborene Fähigkeit beschreibt, durch Reduktion zu abstrahieren [118]. Der Mensch, so Ludewig weiter, benutzt Modelle um die Realität überschaubar zu machen und zu lernen. Dabei bilden die Fähigkeiten der Modellierung, Abstraktion und Konkretisierung für Ludewig einen zentralen Teil der menschlichen Grundausrüstung und sind mit der charakteristisch-menschlichen Fähigkeit der Reflexion unmittelbar verbunden. Modelle haben wohl vielmehr mit unserer angeborenen Denkweise zu tun, obwohl wir das in den meisten Fällen gar nicht wahrnehmen. Stachowiak bringt es auf den Punkt indem er sagt, „[..]daß alle Erkenntnis Erkenntnis in Modellen und/oder durch Modelle ist[..]“ [173].

Tatsächlich ist die Bedeutung und die Verwendung des Modellbegriffs weder in der Umgangssprache, noch im wissenschaftlichen Umfeld einheitlich [182, 25, 163]. In sehr vielen wissenschaftlichen Fachdisziplinen werden eigene Modelle bzw. eigene Modellbegriffe oder Modelltheorien gepflegt. Ludewig [118] differenziert die meist unbewusste, allgemein menschliche Modellbildung von der wissenschaftlichen Modellbildung, wenn er sagt, der eigentliche Zweck der Forschung sei die Entwicklung von Modellen, da Theorien als die Resultate der Forschung in jedem Fall Modelle sind.

Eine allgemeingültige Definition des Begriffs „Modell“ dürfte nicht einfach zu formulieren sein, vermutlich ist es überhaupt nicht möglich [121]. Kreisel [110] ist gar der Ansicht, dass, obwohl praktisch jeder Wissenschaftszweig Modelle braucht, eine genau Untersuchung des Begriffs nur selten nötig ist. Ähnlicher Meinung sind Wedekind et al. [193], wenn sie sagen, „daß man die Modelle einer Theorie oder Disziplin durchaus virtuos handhaben kann, ohne einen allgemein zufriedenstellenden Begriff von ‚Modell‘zu besitzen“. Doch, so fahren Wedekind et al. fort, will man „[...] über Modellbildung und ihre methodischen wie technischen Grundlagen sprechen [...] dann wird schnell deutlich, daß die sorgfältige Einführung von Metabegriffen unverzichtbar ist“. Auch Rothenberg [156] betont, dass es entscheidend ist klar zu definieren, was ein Modell wirklich ist, um zu vermeiden, dass Modelle falsch verwendet werden[3]. Den Bedarf an einer Modelltheorie sieht Mahr [121] darin begründet, dass „Modelle zwar zu den wichtigsten Mitteln unserer Wissens- und Werkproduktion gehören, wissenschaftstheoretisch aber weitgehend ignoriert sind“. Ludewig [118] führt aus, dass erst durch die Betrachtung des Modellbegriffs ein rationaler Umgang mit Modellen möglich wird. Er sieht darin ein Mittel, um das Erkennen, Analysieren und Erstellen von Modellen, sowie das klare Unterscheiden von Modell und Original zu verbessern.

Die „Allgemeine Modelltheorie“ von Stachowiak [172] ist im deutschen Sprachraum wohl die einflussreichste und am häufigsten zitierter Arbeit zu dem Thema. Stachowiak versucht „[..] den allgemeinen Begriff des Modells in seinen Hauptcharakteristika intuitiv-umgangssprachlich zu untersuchen [..]“. Dazu definiert er drei Hauptmerkmale:

[3] sinngemäß aus dem Enlischen übersetzt

- Abbildungsmerkmal:
  „Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.“
- Verkürzungsmerkmal:
  „Modelle erfassen im allgemeinen nicht alle Attribute des durch sie repräsentierten Originals, sondern nur solche, die den jeweiligen Modellerschaffern und/oder Modellbenutzern relevant scheinen.“
- Pragmatisches Merkmal:
  „Modelle sind ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion a) für bestimmte – erkennende und/oder handelnde, modellbenutzende – Subjekte, b) innerhalb bestimmter Zeitintervalle und c) unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen.“

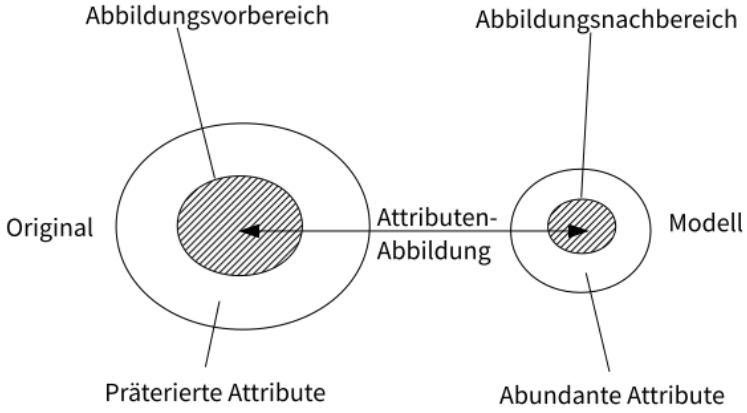


Abb. 2.1: Original-Modell-Abbildung nach Stachowiak [172]

Dem Abbildungsmerkmal legt Stachowiak den „mathematischen (mengentheoretischen, algebraischen) Abbildungsbegriff zugrunde“. Nach dem Verkürzungsmerkmal haben Originale meist Attribute die bei der Modellierung weggelassen werden, also nicht Teil des Modells sind. Stachowiak hat für diese Attribute die Bezeichnung „präterierte (= übergangene, ausgelassene) Attribute“ eingeführt. Attribute die zwar im Modell vorkommen, im Original aber keine Entsprechung finden, also dem Modell hinzugefügt wurden, nennt Stachowiak „abundant (= überfließend, überschüssig)“. Siehe dazu Abb. 2.1.

Im Laufe der Zeit wurde immer wieder Kritik an der Theorie geübt. Wendler [196] wie auch Mahr [121] weisen darauf hin, dass der Begriff des Originals in der *Allgemeinen Modelltheorie* von Stachowiak ein Problem darstellt, da es Modelle gibt, für die kein Original existiert, somit ist das Abbildungsmerkmal nicht erfüllt. Interessanterweise wird von anderer Seite genau die Verwendung des Begriffs Original als geradezu passend erachtet, da ein Original nicht zwingend physisch existieren muss bzw. empirisch untersuchbar sein muss [118, 82]. Schütte [163] hingegen kritisiert, dass der allgemeine Modellbegriff

von Stachowiak dazu verleiten könnte, Modelle als Abbilder der Realität zu verstehen, obwohl Stachowiak Modelle als Konstruktion versteht. Ähnlich argumentiert Thomas [182], der in Stachowiaks Theorie einen Widerspruch zwischen Abbildungsmerkmal und Konstruktionsleistung erkennt. Weiters kritisiert Thomas indirekt die Einbeziehung der Zeit als Teil des pragmatischen Merkmals. Er sieht die Bedeutung der Zeit als evident und daher nicht als modellspezifisch an.

Hier treffen unterschiedliche Positionen verschiedener Fachdisziplinen aufeinander. Wissenschaftstheoretische und philosophisch-erkenntnistheoretische Ansätze führen zu teils sehr unterschiedlichen Auffassungen. Es wird vor Augen geführt, was etwas weiter oben im Text bereits angedeutet wurde. Alle Fachdisziplinen in einer Definition zu vereinen gleicht der Suche nach dem Stein der Weisen. Ludewig [119] sagt dazu „[..] niemand kann einfach definieren was ein Modell ist, und erwarten das andere diese Definition akzeptieren [..]“[4]. Mahr legt die Vermutung nahe, die seiner Aussage nach auch in der Literatur immer wieder durchschimmert, wonach ein allgemeiner Modellbegriff, der alle Aspekte zu erfassen versuche, tatsächlich nichts erfassen würde und deshalb unbrauchbar wäre [121]. Er folgt damit der Meinung von Mittelstraß [132], der sagt, dass innerhalb disziplinärer Kontexte meist begriffliche Klarheit vorherrscht, aber dass es beim Verlassen des Kontexts, oder zwischen unterschiedlichen disziplinären Kontexten, diese begriffliche Klarheit in der Regel nicht gibt.

Es scheint, als würde der Fokussierung des Modellbegriffs auf Spezifika disziplinärer Domänen, einer allgemeinen Definition gegenüber der Vorzug gegeben. Doch selbst innerhalb mehr oder weniger klar abgesteckter Fachdisziplinen ist der Modellbegriff immer wieder Inhalt von lebhaften Diskussionen, siehe z.B. [86, 163, 87, 164] oder [97, 81, 96]. Hier soll im Weiteren der Ansicht von Mittelstraß gefolgt werden. Der nächste Abschnitt ist daher dem Modellbegriff in der Informatik gewidmet.

2.1.2 Modelle in der Informatik

Der vorigen Abschnitt hat gezeigt, dass sich die Definition eines allgemeinen Modellbegriffs, als überaus schwierig herausgestellt hat. Es wurde auch die Vermutung laut, eine allgemeine Definition würde keinen Sinn machen und die Klärung des Modellbegriffs wäre in den Fachdisziplinen besser aufgehoben. Ob sich diese Ansicht mit Blick auf die Informatik bestätigen lässt, soll im Anschluss kurz beleuchtet werden. Nach Ansicht mancher Autoren kommt der Informatik bei der Definition eines Modellbegriffs ein besonderer Stellenwert zu, da sie sich beinahe ausschließlich aus der Erstellung und Bearbeitung von Modellen, sowie der Arbeit mit Modellen zusammensetzt.

Modelle sind, unter anderem, Skizzen als Gedankenstütze, aber auch Dokumentation, sind Grundlage und Beschreibung der Arbeitsweise, haben und sind Methode, sind Theorie und Praxis. Modelle werden genauso schnell entworfen wie verworfen. Modelle sind gleichzeitig Werkbank, Werkzeug und Werkstoff.

[4] sinngemäß aus dem Englischen übersetzt

Im Vergleich zu anderen Wissenschaften ist die Informatik noch sehr jung und hat sich - möglicherweise verursacht durch die rasanten Entwicklungen und durch die relative Kurzlebigkeit der Technologien - gar nicht erst mit der Definition eines fundamentalen Modellbegriffs aufgehalten. Thomas [182] spricht in diesem Zusammenhang von einer Lücke zwischen Forschung und Praxis.

Der Stellenwert von Modellen in der Informatik wird in der Literatur vielfach angesprochen und unterstrichen. Zumindest darüber ist also auf breiter Basis eine Einigkeit gegeben. So beschreibt etwa Jean Bézivin die Informatik sinngemäß als die Wissenschaft von der Erzeugung von Softwaremodellen [41]. Hesse und Mayr [82] schreiben „Die Informatik kann man als die Disziplin der Modellbildung schlechthin bezeichnen, denn jede Form der Verarbeitung von Wissen und Information erfolgt über Modelle."

Reisig [151] beschreibt die ersten zwanzig Jahre der Informatik (1955-1975) als „Modellieren mit Programmen". Die Entwicklungen der letzten zwei bis drei Dekaden umschreibt Reisig dann als „Programmieren mit Modellen". Das ER-Diagramm[5], besser eigentlich die ER-Modellierung, wurde Mitte der 1970er Jahre vorgestellt [48] und seither immer wieder adaptiert. Das objektorientierte Programmierparadigma wurde sogar schon Mitte der 1960er Jahre erfunden[6], benötigte dann aber einige Zeit um sich am Markt zu etablieren. Ab den 1990er Jahren erfuhr die Objektorientierung mehr und mehr Aufmerksamkeit in der Fachwelt. Mit der UML[7] wurde dann 1997 der erste Versuch unternommen, die diversen Modellierungstechniken, die sich rund um die objektorientierte Programmierung etablierten, in einer Sprache zu vereinen und zum Standard zu erheben. Im Umfeld der Bemühungen zur Weiterentwicklung des MDE[8] Ansatzes wurde aus dem von Alan Kay im Kontext der Objektorientierung formulierten Paradigma „Everything is an object" das Paradigma „Everything is a model" [41]. Der gedankliche Übergang vom ersten zum zweiten Paradigma birgt implizit die Bestätigung für eine Modellierungsleistung beim Einsatz von objektorientierten Techniken. Diese Beispiele führen vor Augen, dass die Modellierung nicht erst mit UML und MDE eingeführt wurde, sondern immer schon da war, als integraler Bestandteil dessen, was die Informatik als Wissenschaft und Technik ausmacht. Es soll auch unterstrichen werden, dass Programmierung immer auch Modellierung bedeutet, und das nicht erst seit Einführung der Objektorientierung.

Wenn also das Programmieren als eine Form des Modellierens aufgefasst werden kann, wie oben beschrieben, und das Produkt des Programmierens die Software ist, dann ist also Software auch in gewisser Weise modelliert. Aber ist jede Software ein Modell?

Wedekind et al. [193] beschreiben das Übersetzen von Modellen der verschiedenen Fachwissenschaften in ausführbare Programme als grundlegende Aufgabe der Informatik und sehen in der Simulation einen wesentlichen Einsatzbereich dieser übersetzten Modelle. Sehr ähnlich klingt Ludewig [118], wenn er die Frage stellt, ob Software als „[..] Modell

[5] Entity-Relationship Diagram

[6] Als erste objektorientierte Programmiersprache wird allgemein Simula 67 angesehen, eine Weiterentwicklung von Simula I, entwickelt von Dahl und Nygaard [59].

[7] Unified Modelling Language

[8] Model-Driven Engineering

eines Weltausschnitts [..]“ bezeichnet werden kann und weiters Rechnersysteme, wohl eher scherzhaft, mit Puppenhäusern vergleicht, in denen die Welt vor- oder nachgespielt wird.

Hier wird mit dem Realitätsanspruch ein wichtiger Aspekt der Diskussion um den Modellbegriff im Allgemeinen eingebracht, und die obige Frage erweitert. Ist Software ein Modell der Realität? Die Verarbeitung von Informationen anhand eines Modells das ein Abbild eines Realitätsausschnitts darstellt, könnte durchaus als Simulation aufgefasst werden. Allerdings hängt diese Einschätzung auch wesentlich von dem jeweiligen Einsatzbereich ab. Bei meteorologischen Anwendungen liegt der Bezug zur Simulation noch spürbar nahe. Es könnte auch z.B. ein Zeichenprogramm bzw. ein Textverarbeitungsprogramm als Simulation angesehen werden, nämlich als Simulation von Papier und Stift bzw. einer Schreibmaschine. Bei einer Bankanwendung zur Kontoführung, die ja auch einen Realitätsausschnitt darstellt, ist die Simulation schon nicht mehr so spürbar. Schnell stellt sich die Frage, was ein Browser simulieren soll oder welche simulierte Realität hinter einem Betriebssystems steht?

Wedekind et al. schwächen diesbezüglich auch umgehend ab, indem sie den Begriff der modellierten Realität stark einschränken und den Begriff der Simulation an diverse formale Bedingungen knüpfen. Und auch Ludewig stellt den Realitätsaspekt stark in Frage indem er sinngemäß sagt, dass die Realität immer richtig ist und das Modell immer falsch [119]. Meyer [131] sieht ebenfalls ein Problem in der Vorstellung von Realität als Vorlage von Software Systemen und sagt unter anderem, dass Realität im Auge des Betrachters liegt und das ein Softwaresystem kein Modell der Realität ist, sondern höchstens ein Modell eines Modells von einem gewissen Teil einer Realität. Meyer sagt damit explizit, dass Software keine Realität widerspiegelt und implizit, dass Software ein Modell darstellt[9].

Es kann also sicher nicht jede Software als Simulation bezeichnet werden und nicht jeder Software kann ein Realitäts- oder Weltausschnitt zugeordnet werden. Im vorigen Abschnitt wurde dieser Umstand im Kontext des Abbildungsmerkmals der Allgemeinen Modelltheorie bereits kurz angesprochen. Mahr [120] sagt dazu, dass oft erst die Anwendung eines technischen Systems Realität erzeugt, die Entwicklung des Systems auf Basis dieser Realität also nicht möglich ist. Schütte [163] bringt ein, dass neue oder neuartige Systeme oft nicht auf Basis eines empirischen Originals entwickelt werden. Thomas [182] weist ebenfalls auf diesen Umstand hin, wenn er sagt, dass Anforderungen an Anwendungen oft erst durch den tatsächlichen Gebrauch ermittelt werden können.

Anhand der Anforderungen, die an eine Software gestellt werden, kann ein weiterer, von Hesse [80] sehr treffend als Janusköpfigkeit beschriebener Aspekt, dargestellt werden. Die im Requirements Engineering erhobenen Anforderungen an eine Software stellen als Modell einerseits die Vorstellungen aus Kundensicht dar, z.B. Funktionalität, Design, Performance usw. Andererseits wird die Software diesen Anforderungen entsprechend

[9] Eine lebhafte Diskussion zum Thema „Modell und Realität“ führten Kaschek und Schütte [86, 163, 87, 164].

entwickelt. Das Modell der Anforderungen stellt also ein Nachbild der Kundenvorstellungen dar und fungiert gleichzeitig als ein Vorbild für die Entwicklung. Ludewig [118] spricht von einem deskriptiven Modell der Kundenvorstellungen und einem präskriptiven Modell für die Realisierung. Diese Zwei- oder oft auch Mehrdeutigkeit ist aber nicht auf Modelle im Informatik- bzw. Softwarekontext beschränkt, auf diesen Umstand wurde bereits im ersten Abschnitt kurz hingewiesen, siehe Tab. 2.1. Hier kommen wir wieder auf Ludewig [118, 119] zurück, der davor warnt ein Modell für das Original zu halten.

Die Wirtschaftsinformatik ist bemüht eine Brücke zwischen der Welt der Wirtschaftstreibenden als Auftraggeber und der Welt der Informatik als Auftragnehmer zu schlagen. Hier sind geeignete Modelle als Sprache einer erfolgreichen Kommunikation besonders wichtig. Auf die große Bedeutung von Modellen in der Wirtschaftsinformatik wird in der Literatur auch entsprechend hingewiesen [182, 121]. Eine besondere Position nimmt hierbei die Referenzmodellierung ein. Allerdings herrscht, wie könnte es anders sein, auch darüber was ein Referenzmodell ist, kein einheitliches Verständnis [183, 58]. In der Diskussion um die Referenzmodellierung tauchen immer wieder zwei Modellbegriffe auf, die hier bisher noch keine Erwähnung fanden. Und zwar der abbildungsorientierte und der konstruktionsorientierte Modellbegriff. Schütte [162] verortet die Ursprünge dieser Modellbegriffe in der Betriebswirtschaftslehre, daraus erklärt sich dann auch der Bezug zur Wirtschaftsinformatik.

Der abbildungsorientierte Modellbegriff weist Ähnlichkeiten mit dem Abbildungsmerkmal von Stachowiaks Modelltheorie auf, ein Modell wird aber nicht von einem Original, sondern von einem realen Problem abgeleitet, es wird ein Realitätsbezug unterstellt [182]. Wenig überraschend ist dieser Realitätsbezug dann auch der Hauptkritikpunkt am abbildungsorientierten Modellbegriff [182, 162, 31].

Der konstruktionsorientierte Modellbegriff geht im Gegensatz dazu bei der Modellbildung von einer subjektiven Konstruktionsleistung aus und klammert somit die implizit geforderte Objektivität des abbildungsorientierten Modellbegriffs aus [182, 31, 162]. Thomas [182], wie auch Vom Brocke [31] weisen darauf hin, dass dem konstruktionsorientierten Modellbegriff mittlerweile in der Literatur mehr Beachtung geschenkt wird als dem abbildungsorientierten Modellbegriff.

Auf der Grundlage der allgemeinen Modelltheorie von Stachowiak, die konstruktionsorientiert interpretiert wird, expliziert Thomas seinen Modellbegriff: „Ein Modell ist eine durch Konstruktionsprozess gestaltete, zweckrelevante Repräsentation eines Objekts“ [182]. Ein Modell ist also nach Thomas nicht an Begriffe wie „Original“ oder „Abbildung“ gebunden und auch die Zeit als relativierender Faktor wird ausgelassen. Mit dieser Definition folgt Thomas den Standpunkten der Kritiken an der Stachowiakschen Modelltheorie und am abbildungsorientierten Modellbegriff und umgeht sozusagen die aufgezeigten Schwachpunkte. Erwähnenswert sind auch die von Thomas jeweils in Anhang seiner Arbeit zum Modellverständnis [182] bzw. zum Referenzmodellverständnis [183] gesammelten Begriffsdefinitionen. Es finden sich dort 35 Definitionen des Modellbegriffs bzw. 39 Definitionen des Referenzmodellbegriffs. Weit weniger ausführlich ist die Sammlung von Modelldefinitionen die Muller [134] präsentiert. Rein pragmatisch fällt die Modellde-

finition von Weilkiens [195] aus: „Das Modell (engl. model) beschreibt ein System für einen bestimmten Zweck."

Modelle, Modellierung bzw. Modellbildung und Modellanwendung sind also in der Informatik, und eben gerade in der Softwareentwicklung von zentraler Bedeutung. Und das nicht erst seit der Entwicklung modellbasierter bzw. modellgetriebener Ansätze. Aber obwohl Modelle in der Informatik einen dermaßen hohen Stellenwert einnehmen, scheint es auch hier keine einheitliche Antwort auf die Frage nach dem Modellbegriff zu geben.

2.1.3 Modell versus Diagramm

Die ersten beiden Abschnitte über Modelle haben eine grundlegende Problematik aufgezeigt, nämlich bei der Eingrenzung des Modellbegriffs, sowohl im allgemeinen Gebrauch, als auch speziell in der Informatikdomäne. Die Differenzierung zwischen Modell und Diagramm fällt weniger schwierig aus, bleibt aber trotzdem meist unbeachtet. Im Folgenden soll ein kurzer Blick auf den Unterschied zwischen Modellen und Diagrammen geworfen werden.

Auch in Arbeiten mit Bezug zur Modellierung wird oft von Modellen gesprochen, obwohl eigentlich Diagramme gemeint sind. Ein Diagramm ist aber eine grafische Repräsentation von Modelldaten, eine Visualisierung von Zusammenhängen. Ein Benutzer arbeitet bei der Beschäftigung mit Modellen üblicherweise an Diagrammen. Am Beispiel der Datenmodellierung wird der Zusammenhang besser fassbar. Werden im Kontext der Datenbankerstellung die Zusammenhänge einer Domäne abgebildet, bietet sich dafür eine Diagrammdarstellung an, z.B: ein ER-Diagramm. Zur Darstellung konkreter Datensätze sind Tabellen die bessere Wahl. Das Modell im eigentlichen Sinn stellt nun die Struktur der zueinander in Beziehungen stehenden Tabellen dar.

Laut Stachowiak [172] sind Diagramme „[. . .] graphische Darstellungen entweder von empirischen Zahlenwerten und/oder Funktionsverläufen (Diagramme im engeren Sinne, z.B. Visualisationen statistischer Verteilungen oder physikalischer Weg-Zeit-Gesetze u.dgl.) oder von topischen und/oder strukturellen Originalbeschaffenheiten (Schaubilder und empirische Graphen, etwa die Aufzeichnung einer choreographischen Figur, die Graphendarstellung eines Staatensystems) [...]" [ohne Fußnoten]. Weilkiens [195] sagt, dass ein Modell die vollständige Beschreibung eines Systems beinhaltet, ein Diagramm dagegen nur einen bestimmten Aspekt visualisiert. Noch expliziter wird Delligatti [54], wenn er den folgenden Satz zum Mantra erhebt: „Ein Diagramm eines Modells ist nie das Modell selbst, es ist lediglich eine Sicht auf das Modell"[10]. Er vergleicht weiters einen Berg mit einem Modell und ein Foto des Berges mit einem Diagramm. Wiederum stellt das Diagramm nur eine Sicht auf das Modell dar.

Das Beispiel von Delligatti ist sehr bildhaft und spontan nachvollziehbar, hat im Bezug auf die Modellierung allerdings einen kleinen Schwachpunkt. In dem Beispiel existiert das Modell, bevor davon Ansichten, also Diagramme erzeugt werden. In der praktischen

[10] sinngemäß aus dem Englischen übersetzt

Modellierung jedoch entsteht das Modell, also die Gesamtheit der Informationen, erst durch das Erstellen der Diagramme, also durch die Modellierung. Es wird jedoch sehr anschaulich die Teile-Ganzes Beziehung zwischen Diagrammen und zugrundeliegenden Modellen verdeutlicht. Ein weiterer, sehr gravierender Aspekt tritt bei der Versionierung von Modellen zu Tage, siehe dazu Abschnitt 2.4.3.

2.1.4 Das Metamodell

In den bisherigen Abschnitten waren Modelle im Fokus, also die direkten Produkte der Modellierung. Um die Begriffsgrundlage der Modellbildung zu komplettieren, müssen aber auch die Begriffe Metamodell und Metamodellierung besprochen werden. Dem Metamodellbegriff wurde und wird in der Fachliteratur bisher deutlich weniger Aufmerksamkeit geschenkt als der Diskussion über den Modellbegriff [179, 155]. Durch die zunehmende Popularität von Modellierungssprachen und modellgetriebener Entwicklungsmethoden, sowie durch den breiteren Einsatz von DSLs, rückte aber das Metaisierungsprinzip [180] und damit die Bedeutung von Metamodellen automatisch in den Fokus der Fachdisziplinen.

Der Präfix „meta“ kommt aus dem Griechischen und bedeutet im gegebenen Kontext laut Duden Online „dass sich etwas auf einer höheren Stufe, Ebene befindet, darüber eingeordnet ist oder hinter etwas steht“ [2]. Ein Metamodell stellt, kurz umrissen, das Regelwerk und die Elemente oder anders formuliert die Sprache für die Modellerstellung bereit. Der Sprachbegriff ist hier von einiger Bedeutung, weil damit ein Bezug zu formalen Sprachen hergestellt werden kann [43]

Eine Beschreibung für Metamodelle, die relativ häufig zu finden ist aber von vielen Vertretern der Fachwelt als problematisch bezeichnet oder überhaupt abgelehnt wird [82], lautet wie folgt: „Ein Metamodell ist [...] das Modell eines Modells“ [176] oder „Metamodell: ein Modell von Modellen“ [145]

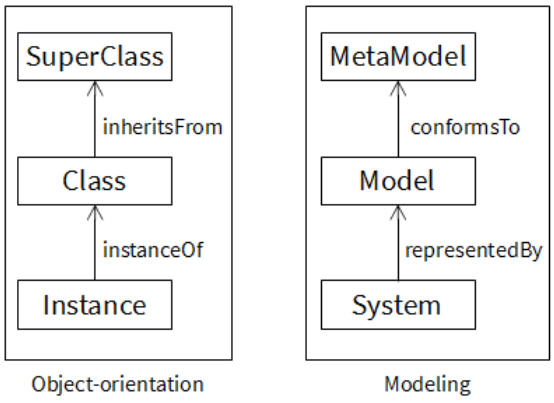


Abb. 2.2: Grundlegende Elemente und Beziehungen [40]

Ein anderer Aspekt, der mehrfach als unpassend oder unzutreffend beschrieben wurde, ist die Bezeichnung „Instanz von“ als Beschreibung der Beziehung zwischen Modell und Meta-

modell [40, 56]. Dieses Beziehungsverständnis stammt aus der Welt der Objektorientierung und sollte nicht einfach aus Bequemlichkeit übernommen werden. Stattdessen sollten die Domänen der Objektorientierung und der Modellierung anhand ihrer grundlegenden Elemente und deren Beziehungen differenziert werden [40], siehe Abb. 2.2.

Bézivin [41] beschreibt den Zusammenhang zwischen Modell und Metamodell sehr schön anhand einer Landkarte und der zugehörigen Legende. Die Landkarte stellt einen abstrahierten Ausschnitt der Realität dar, z.B. die Region eines Landes oder das Straßennetz einer Stadt. Zur Darstellung dieses Realitätsausschnitts werden die Elemente der Legende benutzt. Die Legende definiert und beschreibt diese Elemente. Durch die Verknüpfung mit einer Beschreibung erhalten die Elemente eine Bedeutung, sie werden zu Symbolen, z.B. das Symbol für eine Brücke oder das Symbol für einen Bahnhof. Ohne die Legende ist die Karte (theoretisch) nicht lesbar, da die Elemente alleine keine Bedeutung haben.

Durch die Legende der Landkarte wird eine Sprache definiert, siehe Abb. 2.3. Die grafischen Elemente der Legende stellen die Notation oder Syntax der Sprache dar, die Beschreibungen definieren die Bedeutung, also die Semantik. Die semantische Domäne ergibt sich durch den topografischen Kontext. Die semantische Zuordnung wird durch den Aufbau der Legende gegeben, eine Beschreibung gehört zu einem Element, und zwar zum Nebenstehenden. Die Karte ist demnach in der Sprache der Legende verfasst.

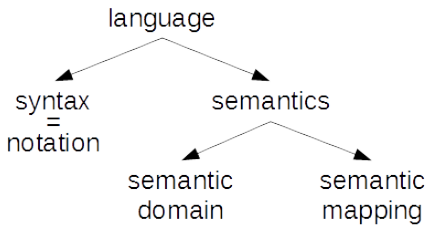


Abb. 2.3: Struktur einer Sprache [72]

Landkarten sind in aller Regel grafischer Natur, die Legende ist somit eine grafische Sprache. Die Beziehung zwischen Karte und Legende ist vergleichbar mit der Beziehung zwischen Modell und Metamodell. Die Karte ist dabei das Modell, die Sprache der Legende wird damit zur grafischen Modell- oder Modellierungssprache. Die Karte ist konform zu der Legende, jedem Element der Karte kann über die Legende eine Bedeutung zugeordnet werden. Die Legende kann, mit einigen Einschränkungen, als Metamodell der Landkarte verstanden werden.

Um zu zeigen, dass ein Metamodell nicht einfach das Modell eines Modells ist, bedient sich Bézivin abermals des Landkartenbeispiels [41]. Wird anhand einer sehr detaillierten Karte eine Übersichtskarte erstellt, entsteht dabei ein Modell eines Modells. Die Übersichtskarte ist dabei aber in keinem Fall ein Metamodell der Detailkarte. Bezugnehmend auf Abb. 2.2 stehen die Detailkarte und die Übersichtskarte nämlich in einer „repräsentiert durch“ Beziehung. Und beide Karten stehen zu ihrer Legende in einer „konform zu“ Beziehung. Es bleibt bei der Legende als Metamodell der Karten.

Wie am Beispiel der Landkarte beschrieben, wird ein Modell üblicherweise in einer Modellierungssprache erstellt. Eine Modellierungssprache ist wiederum nicht mehr als eine künstliche Sprache mit Fokus auf Modellierung, d.h. die Sprache wurde mit Bedacht auf die Modellierungsanforderungen einer Domäne (DSML[11]) oder aber domänenübergreifend, ohne spezielle Ausrichtung (GPML[12]), entwickelt. Künstliche Sprachen der Informatik, z.B. Programmiersprachen, werden formal definiert, nur so kann eine Maschine die Sprache interpretieren und bearbeiten. Formale Sprachdefinitionen stellen die Basis der Automatisierung dar [174]

In relativer Übereinstimmung [72, 20, 129] werden formale Sprachen definiert durch Syntax und Semantik, siehe Abb. 2.3. Die Syntax, oder Notation, zerfällt dabei in abstrakte und konkrete Syntax. Die abstrakte Syntax enthält die Konstrukte der Sprache und die Beschreibung des Aufbaus[13] d.h., wie die Konstrukte in Beziehung stehen können, aber nicht die Notation. Als konkrete Syntax werden Zeichen, Zeichenketten (textuelle Syntax) oder grafische Elemente (grafische Syntax) bezeichnet, die der Sprache als Ausdrücke dienen. Die abstrakte Syntax wird durch die Elemente der konkreten Syntax darstellbar, z.B: als Quellcode oder Diagramm. Zu einer abstrakten Syntax kann es mehrere konkrete Syntaxen geben [175, 109]. Die Semantik wird unterteilt in die semantische Domäne und die semantische Zuordnung. Die semantische Domäne gibt die Bedeutung der Sprachelemente wieder, die semantische Zuordnung verbindet die Semantik mit der Syntax auf eindeutige Weise.

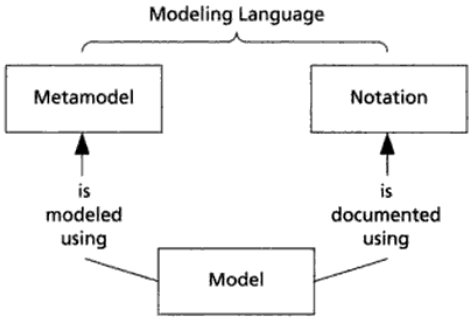


Abb. 2.4: Beziehung von Modell zu Sprachelementen [79]

Nach Abb. 2.4 stellt das Metamodell einen Teil einer Modellierungssprache dar. Manchmal werden Sprachen, speziell Modellierungssprachen aber mit Metamodellen gleichgesetzt [129, 77] oder das Metamodell wird als Definition der Sprache bezeichnet [44]. Anderswo werden Metamodell und abstrakte Syntax synonym verwendet [73, 42]. Wie passt das ins obige Beispiel der Landkarte bzw. zur Definition einer formalen Sprache?

[11] Domain specific modeling language
[12] General purpose modeling language
[13] Wile [200] bezeichnet die abstrakte Syntax als die Beschreibung der strukturellen Essenz einer Sprache.

Der beschriebenen Sprachdefinition folgend, entspricht das Metamodell der abstrakten Syntax. Gerade im Modellierungskontext wird die abstrakte Syntax meist über ein Metamodell ausgedrückt [109]. Mehr als die Definition der abstrakten Syntax ist mit dem Metamodell eigentlich nicht getan [94, 67, 191]. Da aber der wichtigste Aspekt einer Modellierungssprache im Metamodell steckt [138], nämlich das Domänenwissen und damit der formale Aufbau, die Syntax jedoch austauschbar ist, werden Metamodelle oft, nicht ganz korrekt, als Sprachen oder Modellierungssprachen bezeichnet [97]

Favre [57] kommt zu der Auffassung, dass ein Metamodell das Modell einer Modellierungssprache ist. Bézivin [40] schlägt als Beschreibung der Beziehung zwischen Modell und Metamodell die Bezeichnung „konform zu“ vor. Ein Metamodell modelliert damit einen Teil einer Modellierungssprache, wie weiter oben beschrieben. Ein Modell, das in dieser Modellierungssprache erstellt wird, ist zu dem Metamodell konform. Wie oder wodurch wird aber das Metamodell formal beschrieben? Hier kommt das anfangs erwähnte Metaisierungsprinzip zum Tragen. Strahringer [180] formuliert es folgendermaßen: „Das Metaisierungsprinzip beschreibt denjenigen Aspekt eines Modells, der in der übergeordneten Modellierungsstufe abgebildet wird.“

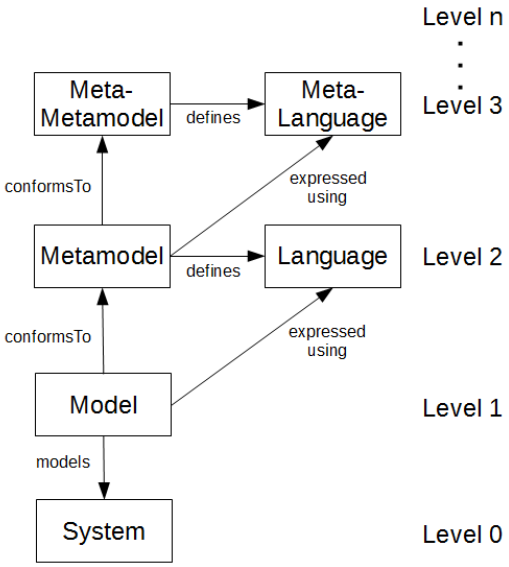


Abb. 2.5: Metaisierungsprinzip[14]

Diesem Prinzip folgend, lässt sich über jedes Modell ein Metamodell stellen und da ein Metamodell ebenfalls ein Modell ist, ergibt sich auf der nächsten Ebene ein Metametamodell usw., siehe Abb. 2.5. Auf Ebene 0 steht das zu modellierende Objekt, z.B. ein System oder eine Systemvorstellung. Ebene 1 enthält das Modell, ausgedrückt in der Sprache der Ebene 2, z.B. UML. Die Sprache der Ebene 2 wird definiert durch die Ebene

[14] erstellt aus Abb. von Kühne [97], Bézivin [40] und Strahringer [180]

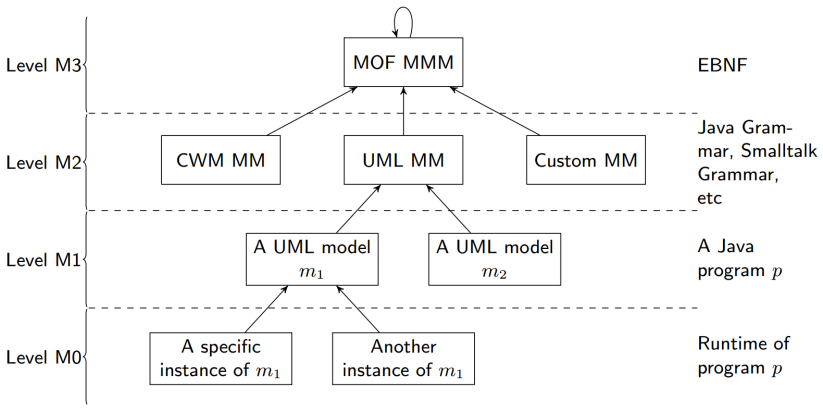


Abb. 2.6: 4-Ebenen Architektur [33]

3 usw. Da das Metaisierungsprinzip wiederholt anwendbar ist, können theoretisch beliebig viele Modellierungsstufen definiert werden. Stufen höher als Ebene 3 bieten aber aus technischer Sicht keinen Mehrwert, deshalb wird allgemein eine 4-Ebenen Architektur verwendet, siehe Abb. 2.6.

Anhand dieser Abbildung ist die Parallele zu Programmiersprachen gut nachvollziehbar, in diesem Fall Java. Die Ebene M3 ist reflexiv definiert, also durch sich selbst. Dadurch sind keine weiteren Ebenen notwendig. Zur Definition von Programmiersprachen kann auf Ebene M3 z.B. die erweiterte Backus-Naur Form verwendet (EBNF) werden. Dabei ist zu beachten, dass eine EBNF Grammatik sowohl abstrakte, als auch konkrete Syntax definiert.

Wofür werden Metamodelle nun praktisch eingesetzt? Metamodellierung wird von Stahl und Völter als einer der wichtigsten Aspekte der modellgetriebenen Softwareentwicklung bezeichnet, siehe dazu Abschnitt 2.3 über modellgestützte Entwicklungsmethoden. Als Anwendungsfälle werden die folgenden beschrieben [175]:

- Definition der abstrakten Syntax von Modellierungssprachen,
- Validierung von Modellen anhand des Metamodells,
- Model-zu-Model Transformationen,
- Generierung von Quellcode anhand des Metamodells,
- Unterstützung von Modellierungssprachen für Modellierungswerkzeuge.

Allgemein ist die Situation ähnlich zu bewerten, wie bereits beim Modellbegriff geschehen. Metamodelle, sowie die Metamodellierung, haben einen hohen Stellenwert in der Informatik, besonders in der Softwaretechnik. Strahringer stellte 1995 fest, dass es keine präzise Definition vom Begriff Metamodell gibt [180]. Auch nach aktuell etwas mehr als

zwanzig Jahren, behält diese Feststellung Gültigkeit. Es wurde zwar vieles publiziert und diskutiert und einiges standardisiert, aber letztlich bleibt die Frage, warum in der sehr formalen Softwarewelt bisher keine einheitliche, allgemeingültige, formale Begriffswelt geschaffen werden konnte [155].

2.2 Modellierungssprachen

In diesem Abschnitt sollen die Modellierungssprachen, die im Kontext dieser Arbeit von Bedeutung sind, kurz vorgestellt werden. Den Anfang macht die Unified Modeling Language, danach folgt die Meta Object Facility und zum Abschluss wird die relativ neue Sprache SysML besprochen.

2.2.1 Die Unified Modeling Language

Die Unified Modeling Language (UML)[15] ist, kurz gesagt, eine grafische Modellierungssprache. Meist wird die UML zur Modellierung von objektorientierter Software eingesetzt[16], kommt aber auch in anderen Domänen zum Einsatz und kann über einen Erweiterungsmechanismus für spezielle Anforderungen angepasst werden. Weiterentwickelt und standardisiert wird die UML von der Object Management Group (OMG). Die UML hat sich als weltweiter Standard etabliert, die Akzeptanz als ISO Standard[17] bekräftigt diesen Status.

Die UML blickt auf eine über zwanzigjährige Entwicklungsgeschichte zurück. Noch weiter zurück, und damit knapp 30 Jahre vor der ersten UML Version, liegt die Erfindung der Objektorientierung durch die Entwickler der Programmiersprache Simula im Jahre 1966 [52]. Dieses vollkommen neue Verständnis von Abstrahierung und Modellierung wurde dann über die Programmiersprache (und zugleich Entwicklungsumgebung) Simula zur Marktreife weiterentwickelt. Fast alle heute von objektorientierten Programmiersprache bekannten Grundkonzepte wurden damals mit Simula eingeführt [140].

In der beinah universellen Einsetzbarkeit/Anwendbarkeit, gepaart mit relativ einfacher Vermittelbarkeit und daher flacher Lernkurve[18], liegt die Mächtigkeit dieser Grundkonzepte. Die meisten der damals stark am Markt vertretenen Programmiersprachen wurden um Objektorientierung erweitert, z.B. Pascal, C (eigentlich C++), Ada. Beinah alle neuen imperativen Programmiersprachen, die seither entwickelt wurden, nutzten von vornherein Objektorientierung[19].

Der Einsatz einer objektorientierten Programmiersprache ist allein jedoch noch kein Garant für gute Software bzw. ganz generell für Projekterfolge. Die Sprache ist schließ-

[15] zu Deutsch sinngemäß etwa „vereinheitlichte Modellierungssprache“

[16] Objektorientierung ist allerdings keine zwingende Voraussetzung, `http://www.uml.org/what-is-uml.htm`

[17] ISO/IEC 19501 (UML 1.4.2) und ISO/IEC 19505 (UML 2.4.1)

[18] zumindest was die Grundkonzepte der Objektorientierung betrifft

[19] bekannte aber unrühmliche Ausnahmen sind PHP und Perl, beide wurden erst nachträglich um Objektorientierung erweitert

lich auch nur ein Werkzeug. Ähnlich hat das Brooks [32] in seinem berühmten Essay "No Silver Bullet" beschrieben. Was noch fehlte waren Methoden zur objektorientierten Analyse, zum Design und zur Visualisierung. Grafische Visualisierungen als Hilfsmittel der Softwareentwicklung waren ebenfalls bekannt und etabliert, etwa das ER-Diagramm oder das Nassi-Shneidermann-Diagramm, der Objektorientierung wurden diese Visualisierungsarten allerdings nicht gerecht. Ab Anfang der 1990er Jahre kam es dann zu der Periode, die in der Literatur immer wieder ein wenig euphemistisch als die Methodenblüte bezeichnet wird [195], die eine Vielzahl von Methoden und Notationen hervorbrachte. Booch, Rumbaugh und Jacobson[20] vereinheitlichten ihre Notationen und geboren war die UML.

Mit der Object Management Group (OMG[21]) existiert seit 1989 ein internationales Non-Profit Industriekonsortium, das sich der Entwicklung und Pflege von Technologiestandards verschrieben hat. Im Jahr 1997 wurde die UML 1.1 der OMG zur Standardisierung vorgelegt und noch im selben Jahr als OMG Standard angenommen. Seither kümmert sich die OMG um die Pflege und Weiterentwicklung der UML. Im Jahr 2005 wurde die stark überarbeitete Version 2.0 veröffentlicht, die momentan aktuelle Version 2.5.1 wurde im Dezember 2017 veröffentlicht.

Was also ist die UML heute? Zuallererst ist die UML keine Entwicklungsmethode und enthält auch keine. Die ursprünglich enthaltene Methode wurde von den drei Amigos separat weiterentwickelt und Anfang 1999 in dem Buch „The Unified Software Development Process“ veröffentlicht, auch bekannt als „Unified Process“.

„Die UML ist in erster Linie die Beschreibung einer einheitlichen Notation und Semantik sowie Definition eines Metamodells“, schreibt Bernd Oestereich [140] und etwas später in derselben Publikation „Die Unified Modeling Language (UML) ist eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme“. Außerdem sagt er: „Modeling Language ist hauptsächlich eine vornehme Umschreibung für Notation“.

Die UML ist eine formale Sprache. Die Spezifikation definiert mit dem UML Metamodell eine abstrakte Syntax. Die konkrete Syntax wird anhand der Notation zur grafischen Modellierung spezifiziert. Die Definition der Semantik liegt anhand der Elementbeschreibungen vor. Die UML Spezifikation wurde mit Version 2.5 etwas einfacher und lesbarer gestaltet, sowie auf ein Dokument reduziert[22]. Die Spezifikation ist aber weiterhin von einigen normativen Referenzen abhängig und zwar von den OMG Spezifikationen für:

- Meta Object Facility (MOF)[23],
- Object Constraint Language (OCL)[24],

[20] auch bekannt als „Die drei Amigos“

[21] `http://www.omg.org`

[22] Die UML-Spezifikation war ab der Version 2.0 in zwei Dokumente aufgeteilt: UML-Infrastructure und UML-Superstructure.

[23] `http://www.omg.org/mof`

[24] `http://www.omg.org/spec/OCL`

- XML Metadata Interchange (XMI)[25],
- Diagram Definition (DD)[26].

Die MOF ist eine Metamodellierungssprache, also eine Sprache zur Definition von Modellierungssprachen. Die UML und andere OMG Standards, wie z.B. OCL oder das Common Warehouse Metamodel (CWM), basieren auf der MOF. Die OCL ist eine Sprache zur formalen Definition von Regeln oder Bedingungen, bei der Modellierung mit UML oder einer anderen OMG Modellierungssprache. Das Format XMI ist der Standard für Speicherung und Austausch von Modelldaten. Das Format DD ist zum Diagrammaustausch vorgesehen, es ist seit Version 2.5 Teil der UML Spezifikation und ersetzt das Diagram Interchange (DI) Format. Anders als mit XMI können damit Layoutdaten, wie Größe und Position von Elementen, gespeichert werden. Es ist ein weiterer Schritt hin zur angestrebten Interoperabilität zwischen Modellierungswerkzeugen.

Die unabhängige Standardisierung garantiert den Anwendern Kontinuität und die OMG legt bei ihren Standards großen Wert auf Interoperabilität. Die Vielfalt der Modellierungsmöglichkeiten macht die UML mächtig und ausdrucksstark und dadurch sehr vielseitig einsetzbar. Allerdings wuchs auch der Umfang der Spezifikation. Die Gesamtheit aller zugehörigen Spezifikationen umfasst hunderte Seiten und enthält unzählige Elemente und Details. Es wird deshalb mit Kritik an der UML nicht gespart [76, 71, 61, 170, 171]. Die Vielfalt der Sprache soll jedoch nicht als abschreckende Hürde angesehen werden. Laut Kobyrn [98] sind für wahrscheinlich 80% der Praxisanforderungen 20% der spezifizierten Elemente zur Modellierung ausreichend. Diverse Untersuchungen über die Verwendung der UML in der Praxis belegen das ebenfalls, z.B. Grossman et al. [70].

Die UML definiert 14 Diagrammarten, grob aufgeteilt in Strukturdiagramme und Verhaltensdiagramme, siehe Abb. 2.7. Strukturdiagramme bilden die Komponenten und den Aufbau und Verhaltensdiagramme das dynamische Verhalten eines Systems ab [165, 140].

Ursprünglich war die UML nur als Modellierungssprache für Softwaresysteme gedacht. In gewisser Weise war die UML also eine DSL. Mittlerweile, und vor allem seit dem Sprung auf die Version 2.0, wird in Bezug auf die Verwendungsmöglichkeiten aber von einer GPL gesprochen [65]. Die UML kommt heutzutage nicht mehr nur in der Softwareentwicklung zum Einsatz, sie wird auch in anderen Domänen verwendet, etwa zur Geschäftsprozessmodellierung oder zum Systems Engineering [195]. Der breite Einsatzbereich ergibt sich durch den schon erwähnten Umfang der Modellierungsmöglichkeiten.

Trotz des konzeptuellen Umfangs ist es der UML nicht immer möglich, sämtliche Anforderungen aller denkbaren Domänen abzubilden. Das ist dann der Fall, wenn Syntax und/oder Semantik der UML nicht ausreichen, um spezielle Konzepte einer Domäne auszudrücken, oder wenn UML Elemente eingeschränkt oder angepasst werden sollen [65]. Deshalb wurde für die UML schon in frühen Entwicklungsphasen ein Erweiterungsmechanismus

[25] http://www.omg.org/spec/XMI
[26] http://www.omg.org/spec/DD

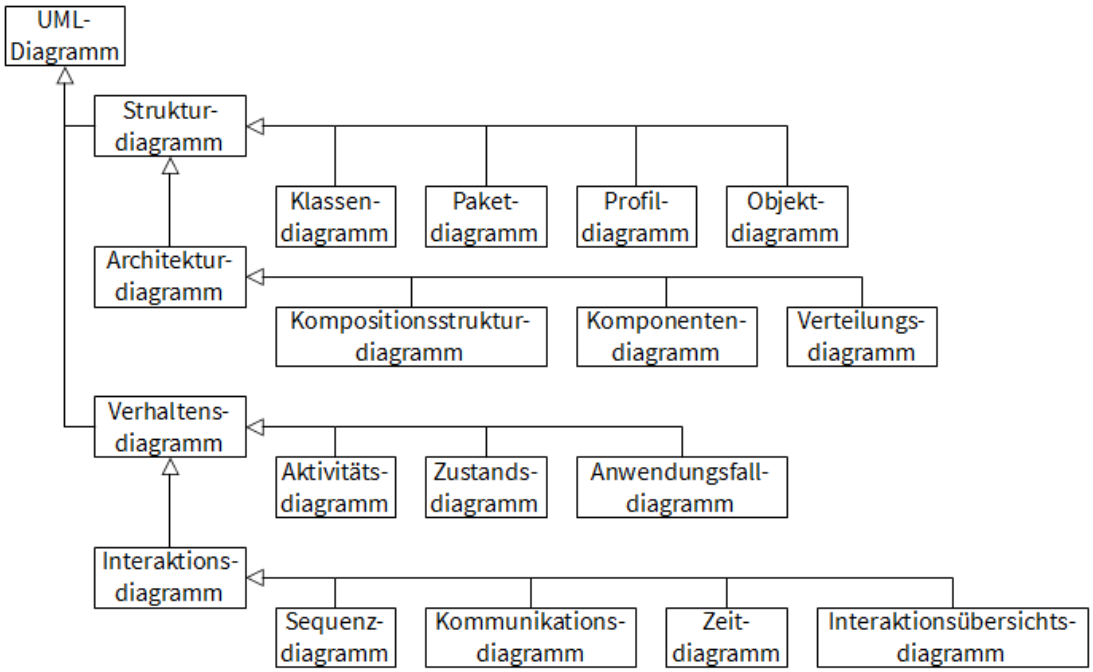


Abb. 2.7: Überblick der UML-Diagramme [140]

angedacht [78, 19, 65]. Mit den sogenannten UML Profilen wurde eine ausgeklügelte Methode zur Erweiterung der UML spezifiziert. Mithilfe von Profilen kann die UML an die Anforderungen spezieller Domänen angepasst werden, das Ergebnis ist dann wieder eine DSL. In den letzten zehn bis fünfzehn Jahren wurden DSLs immer populärer. Die Hürden zur Erstellung von DSLs wurden durch entsprechende Entwicklungswerkzeuge weiter gesenkt und damit stieg auch die allgemeine Akzeptanz.

Die OMG gibt prinzipiell zwei Möglichkeiten zur Erstellung einer DSL vor. Die erste und sicher aufwändigere Methode besteht in der Definition einer eigenen, neuen Sprache auf Basis der MOF. Die andere Möglichkeit ist die Erstellung eines UML Profils. Das ist meist der einfachere Weg, weil auf die Elemente der UML aufgebaut wird[27]. Profile sind dabei eigentlich nur Container, es handelt sich einfach um Gruppierungen oder Pakete, die meist mehrere Erweiterungen beinhalten. Die tatsächlichen Erweiterungen sind Stereotypes, Tagged Values und Constraints, sowie Beziehungen (Associations) zwischen Stereotypes und/oder Metaelementen, siehe Abb. 2.8.

Ein Stereotype ist eine Spezialisierung einer UML Metaklasse, also eine erweiterte Subklasse, im Beispiel etwa „Coloured“. Ein Stereotype kann selektiv auf passende Elemente angewendet werden oder als „required“ markiert werden. Damit erhält jede Modellinstanz der Metaklasse und alle Subklassen davon die Anpassungen [167]. Ein

[27] Die Vor- und Nachteile der jeweiligen Methode wurden von Bran Selic [167] und Fuentes-Fernández et al. [65] zusammengefasst.

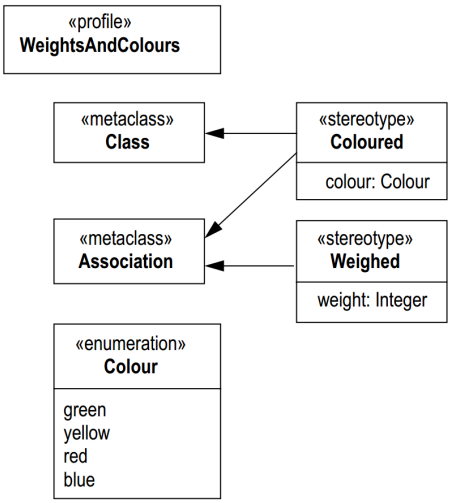


Abb. 2.8: Beispiel für ein UML-Profil [65]

Tagged Value ist eine Eigenschaft bzw. ein Attribut eines Stereotypes, im Beispiel etwa das Attribut „colour“ des Stereotypes „Coloured“. Ein Constraint ist eine Regel oder Bedingung, im obigen Beispiel könnte etwa definiert werden, dass Elemente nur mit einer Beziehung verbunden werden dürfen, wenn die Farben übereinstimmen, etwa die Farben der Elemente und die der Beziehung.

Mit diesen Elementen wird die Möglichkeiten geboten, die UML Syntax und Semantik zu erweitern, es darf aber die Semantik bestehender UML Elemente nicht verändert werden [An Introduction to UML Profiles]. Der Profilmechanismus ist im Prinzip ein Teil von MOF, das bedeutet, dass ein Profil nicht nur auf UML angewendet werden kann, sondern auf jede Sprache, die mit MOF definiert wurde, sogar auf andere Profile [167, 65]. Die OMG bietet selbst zahlreiche UML Profile für unterschiedliche Einsatzmöglichkeiten an, meist passend zu den eigenen Standards[28].

2.2.2 Die Meta Object Facility

Wie etwas weiter oben schon kurz beschrieben, ist die MOF eine Metasprache zur Definition von Sprachen, genauer Modellierungssprachen. Anders formuliert ist die MOF ein Meta-Metamodell, auf dessen Basis Metamodelle erstellt werden können. Der Wert einer einheitlichen Metasprache wurde früh erkannt, siehe dazu Abschnitt 2.1.4. Der Begriff und das Konzept MOF wurden von der OMG etwa zeitgleich mit der UML als Standard veröffentlicht. Die MOF wurde in der Folge zum Fundament zahlreicher OMG Standards ausgebaut. Wie die UML ist auch die MOF zum ISO Standard[29] erklärt worden.

[28] `http://www.omg.org/spec/#Profile`
[29] ISO/IEC 19502 (MOF 1.4) und ISO/IEC 19508 (MOF 2.4.2)

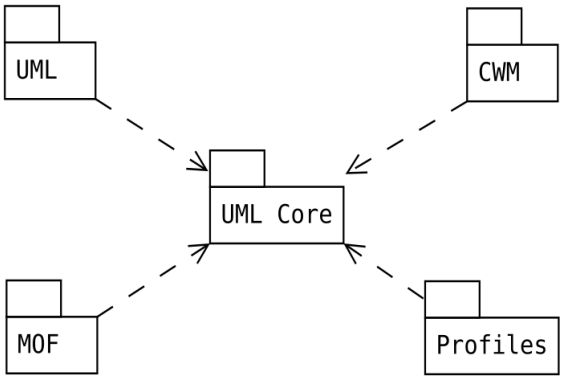


Abb. 2.9: Der UML Kern als gemeinsame Basis für Metamodelle [67]

Da die UML ein Metamodell darstellt und durch die MOF definiert wird, stellt die MOF wiederum ein Metamodell der UML dar und ist somit ein Meta-Metamodell, siehe dazu auch Abschnitt 2.1.4. Die MOF ist auf Ebene M3 angesiedelt und definiert sich daher selbst, sie greift aber interessanterweise auch auf Elemente der UML zurück. Somit ist ein Teil, der mit der MOF definierten UML, wiederum Teil der MOF, siehe Abb. 2.9. Dieser Kunstgriff soll gewährleisten, dass UML Werkzeuge auch für die MOF benutzt werden können, dedizierte MOF Werkzeuge werden damit unnötig, das erhöht die Akzeptanz bei Entscheidungsträgern.

Die gesamte Spezifikation der MOF ist, ähnlich der UML, sehr ausdrucksstark, dadurch aber auch umfangreich und komplex und daher nicht ganz einfach zu implementieren. Zur Vereinfachung wird die MOF in ihrer Spezifikation in zwei Versionen angeboten [67, 46]. Die Essential MOF (EMOF) stellt alle notwendigen Kernkonzepte zur Definition von Modellierungssprachen bereit. Der Zweck der EMOF ist einfache Metamodelle, basierend auf einfachen Konzepten, zu ermöglichen [147]. Die Complete MOF stellt die komplette Spezifikation der MOF dar, EMOF ist somit wiederum ein Teil der CMOF. Die CMOF ist ausgelegt für größere, anspruchsvollere Metamodelle, z.B. basiert die UML auf der CMOF [146].

2.2.3 Die System Modeling Language

Die Systems Modeling Language (SysML), eigentlich OMG SysML[30], ist eine grafische Modellierungssprache, ähnlich der UML aber mit einem Fokus auf Systems Engineering[31]. Werden modellbasierte Methoden zum Systems Engineering verwendet, dann wird allgemein von Model-Based Systems Engineering (MBSE) gesprochen, siehe auch Abschnitt 2.3.2. Das Produkt das beim MBSE am Ende der Entwicklungsphase steht, ist ein physisches, technisches System, z.B. ein Flugzeug, Auto oder Satellit. Das Produkt kann daher nicht, wie etwa in der modellgetriebenen Softwareentwicklung vorgesehen,

[30] http://www.omgsysml.org
[31] engl. für Systementwicklung

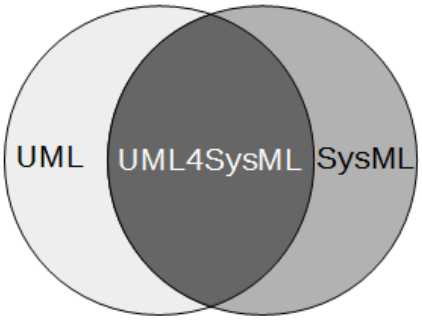


Abb. 2.10: SysML ist Wiederverwendung und Erweiterung der UML [195]

vollautomatisch aus dem Modell transformiert bzw. generiert werden. Das MBSE birgt dennoch ein riesiges Fortschrittspotential in sich. Ein vielbeachtetes Thema in Bezug auf SysML ist etwa die Simulation von Systemen anhand von SysML Modellen. Ein hinreichend detailliertes Modell vorausgesetzt, können Probleme schon sehr früh im Entwicklungsprozess erkannt und gelöst werden [195]. Dabei gilt, wie auch im Software Engineering, umso früher desto günstiger.

Die SysML hat sich seit der offiziellen Einführung 2007 mittlerweile zur Standardsprache der Systemmodellierung entwickelt. Auch die SysML wurde von der ISO zum Standard erklärt[32]. Sie entstand aus einer Initiative des International Council on Systems Engineering (INCOSE[33]) in Kooperation mit der OMG und wurde auf Basis der UML entwickelt. Die relativ einfache Erweiterbarkeit der UML war dabei von Vorteil, siehe Abschnitt 2.2.1. Die SysML besteht aus einer UML-Teilmenge die unverändert übernommen wurde, auch UML4SysML genannt, und einer Menge von angepassten UML Komponenten, sowie einer Reihe von neu eingeführten Elementen, siehe Abb. 2.10. Im Unterschied zur UML mit ihren vierzehn Diagrammen, kommt die SysML mit lediglich neun Diagrammen aus, siehe Abb. 2.11. Die aktuelle Version der Spezifikation ist 1.5 [147].

Die Entstehungsgeschichte der SysML ist kürzer und weniger spannend als die der UML, dafür ist die Domäne des Systems Engineering älter als die des Software Engineering. Der Begriff Systems Engineering geht, laut INCOSE, zurück auf die frühen 1940er Jahre, die Konzepte sind wohl noch älter [8]. Das Systems Engineering ist ein multidisziplinärer, ganzheitlicher Ansatz zur Entwicklung von komplexen Systemen. Es bezieht sich auf den gesamte Lebenszyklus eines Systems, von der Planung über die Inbetriebnahme bis zur Stilllegung und betrachtet sowohl technische, als auch wirtschaftliche Aspekte [63]. Software Engineering ist eine der vielen Disziplinen, die im Systems Engineering zur Anwendung kommen.

Das Systems Engineering ist, aufgrund der zum Teil sehr sensiblen Natur der Anwendungsbereiche, wie etwa Luft- und Raumfahrt, Automobil- aber auch Rüstungsindustrie, sowie

[32] ISO/IEC 19514 (OMG SysML 1.4)
[33] zu Deutsch etwa Internationaler Rat für Systementwicklung, `http://www.incose.org`

gesetzlicher Vorschriften und/oder Prozessnormen, zu umfangreicher Dokumentation verpflichtet. Daraus und aus den kommunikativen Anforderungen der zwangsläufigen Interdisziplinarität, resultieren dokumentenlastige, schwergewichtige Entwicklungsprozesse, siehe Abschnitt 2.4.1, und diese Prozesse sind aufwändig und daher teuer [54, 195]. Wie im Software Engineering gab es auch im Systems Engineering bereits vor UML und SysML eine Vielfalt von unterstützenden Diagrammen und Modellen [62, 116]. Da die Vorteile von gemeinsamen Industriestandards schon lange bekannt waren, wurde in zahlreichen Anläufen versucht, die Praktiken und Techniken des Systems Engineering zu vereinheitlichen und Prozessstandards zu entwickeln. Auch das Interesse an Standards zum Datenaustausch bzw. zur Interoperabilität von Werkzeugen wurde laufend größer. In der Industrie wurde das breite Spektrum der Vorteile und Möglichkeiten von modellbasierten Methoden im Systems Engineering wohlwollend begrüßt. Gegen die Entwicklung einer völlig neuen Modellierungssprache standen einige Vorteile der UML, wie etwa die Robustheit der Sprache, der enthaltene Erweiterungsmechanismus, der hohe Grad an Verbreitung und Akzeptanz und damit einhergehend die bestehende Infrastruktur an Werkzeugen und Bildungsangeboten, sowie die relative Investitionssicherheit, resultierend aus der unabhängigen Weiterentwicklung der Sprache durch die OMG [62].

In Bezug auf das Systems Engineering ist die Semantik der UML in einigen Teilen unzulänglich. Das Systems Engineering bringt Anforderungen mit, die mit der UML nicht erfüllt werden konnten. So war es nur eingeschränkt möglich physische Systeme oder Hierarchien zu modellieren. Weiters war es gänzlich unmöglich den Fluss von Energie (z.B. elektrische oder thermische Energie) oder Materie (z.B. Flüssigkeiten oder Gase) darzustellen. Anforderungen (engl. requirements) und Zusicherungen (engl. constraints) konnten ebenso wenig erfasst werden wie zeitliche und parametrische Abhängigkeiten oder Beziehungen [62, 75]. Daher wurde eine Erweiterung der UML über den Profilmechanismus realisiert. Dabei wurden viele Teile der UML wiederverwendet, die Schnittmenge aus UML und SysML wird als UML4SysML bezeichnet, siehe Abb. 2.10. Damit wurde die Wiederverwendung bestehender und bekannter Konzepte der UML ermöglicht, was sich positiv auf die Unterstützung durch etablierte Modellierungswerkzeuge auswirkte. Die SysML erweitert aber die UML nicht einfach, sondern schränkt den Umfang, wo dies möglich bzw. nötig ist, auch drastisch ein. So wurde die Anzahl der Diagramme von den vierzehn der UML auf lediglich neun in der SysML reduziert. Vier UML Diagramme wurden unverändert übernommen, drei Diagramme wurden erweitert und zum Teil umbenannt, der Rest wird für SysML nicht benötigt. Zwei Diagramme wurden allerdings auch ganz neu eingeführt, siehe Abb. 2.11. Bei Bedarf kann einem SysML Modell aber auch ein UML Diagramm hinzugefügt werden. Sollten z.B. spezielle Anforderungen ein UML Kommunikationsdiagramm erfordern, so ist dies erlaubt [75]. Die SysML wurde 2007 von der OMG als Standardsprache zur Systemmodellierung offiziell vorgestellt.

Zwar wurde die SysML an die Domäne des Systems Engineering angepasst, sie stellt aber dennoch eine GPML dar [147]. Um einen möglichst breiten Einsatzbereich abdecken zu können, wurde die SysML relativ allgemein gehalten, zu allgemein für eine DSL. Da der Profilmechanismus aber auch auf Profile angewendet werden kann, spricht nichts gegen

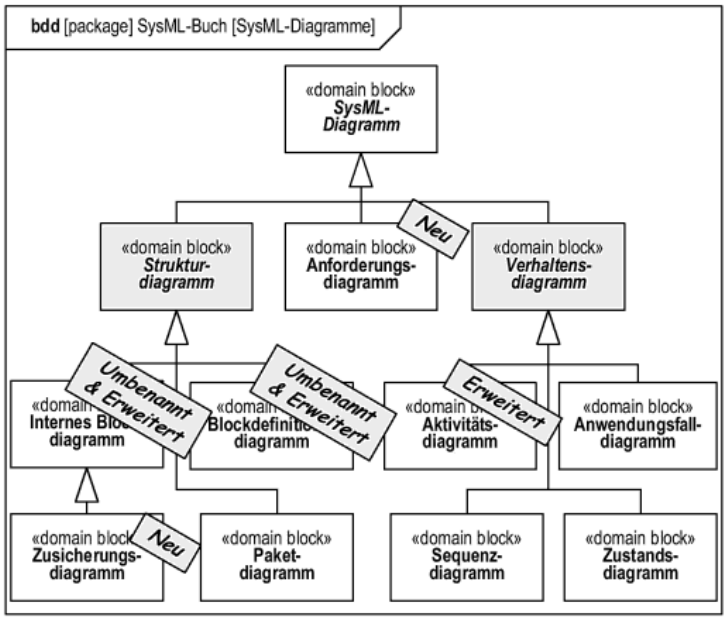


Abb. 2.11: Die SysML Diagramme als Blockdefinitionsdiagramm [195]

eine weitere Spezialisierung der SysML, sollten die Anforderungen einer Anwendung dies erfordern.

Die Diagramme der UML können aufgeteilt werden in Struktur- und Verhaltensdiagramme. Bei der SysML werden zur vollständigen Systembeschreibung noch Anforderungen und Zusicherungen benötigt. Diese vier Diagrammarten bilden die sogenannten vier Säulen der SysML, siehe Abb. 2.12. Die wichtigsten Unterschiede zur UML sind hier kurz zusammengefasst [195]:

- Statt Klassen und Objekten kommen Systembausteine bzw. Blöcke zum Einsatz.
- Objektflüsse, also der Transfer von Daten oder Materie, können im internen Blockdiagramm dargestellt werden.
- Kontinuierliche Funktionen können im Aktivitätsdiagramm durch Aktions- oder Objektknoten modelliert werden, es werden auch EFFBDs (Enhanced Functional Flow Block Diagrams[34]) unterstützt.
- Die neuen Diagramme: Anforderungsdiagramm und Zusicherungsdiagramm.
- Die Unterstützung des Datenaustauschformats ISO AP-233 ermöglicht Interoperabilität mit bestehenden Softwarewerkzeugen aus dem Ingenieurssektor.

[34] im Systems Engineering weit verbreitetes Verhaltensdiagramm [147, 195]

- Die SysML schreibt Diagrammrahmen zwingend vor, der Rahmen ist hier ein Teil des Diagramms und stellt sozusagen den Namensraum grafisch dar.
- Vorkenntnisse über UML oder Objektorientierte Modellierung sind nicht vorausgesetzt, so können z.B. Elektrotechniker oder Mechaniker oder generell Personen ohne Erfahrung in Softwareentwicklung die SysML zur Systemmodellierung einsetzen.
- Wertetypen sind von UML Datentypen abgeleitet, da Datentypen in der SysML nicht verwendet werden dürfen. Damit können Einheiten als Typ definiert werden die zur Definition von Anforderungen benötigt werden, z.B. Masse in kg.
- Die SysML unterstützt die Definition von parametrischen Beziehungen zwischen Eigenschaften von Systembausteinen, z.B. zur Definition von Zusicherungen (engl. constraints).

Genauso wie bei der UML, beschreibt auch die Spezifikation der SysML keine Methode. Die Verwendung einer Methode zur Systemmodellierung wird jedoch dringendst empfohlen, siehe Abschnitt 2.3.2. Relativ bekannte und erprobte Methoden sind OOSEM[35] des INCOSE [63] und SYSMOD von Weilkiens [195].

Wie bereits in den Unterschieden zur UML kurz erwähnt, setzt die SysML einen Diagrammrahmen zwingend voraus. Im Diagrammkopf sind folgende Angaben verpflichtend:

Typabkürzung [Modellelementtyp] Modellelement [Diagrammname]

Der Diagrammkopf in Abb. 2.11 beschreibt z.B. ein Blockdefinitionsdiagramm mit dem Inhalt des Pakets SysML-Buch, das Diagramm hat den Namen „SysML-Diagramme“. Der Grund ist ein eigentlich praktischer, wie Delligatti [54] beschreibt: „Jedes Diagramm repräsentiert ein Element das irgendwo im Systemmodell definiert wurde. Präziser gesagt, der Diagrammrahmen repräsentiert ein Element im Model. Und dieses Modellelement ist das Element dessen Typ und Name im Diagrammkopf aufscheint.“[36] Wie Delligatti weiter ausführt, definiert das Modellelement, das von einem Diagramm repräsentiert wird, einen Namensraum (engl. namespace), einen Container innerhalb der Modellhierarchie, für die im Diagramm enthaltenen Elemente. Damit wird über den Typ und den Namen des Modellelements im Diagrammkopf angezeigt, wo die Diagrammelemente im Modell gefunden werden können.

Als Abschluss dieses Abschnitts werden die neuen oder abgeänderten Diagramme der SysML kurz zusammengefasst [54, 195]:

- Das Aktivitätsdiagramm: Dieses Diagramm wurde komplett von der UML übernommen und um einige Eigenschaften erweitert [195], z.B. Unterstützung kontinuierlicher Systeme, erweiterter Kontrollfluss, Ablaufwahrscheinlichkeiten, Modellierungsregeln für Aktivitäten (Aktivitätsbaum).

[35] Object-Oriented Systems Engineering Method
[36] sinngemäß aus dem Englischen übersetzt

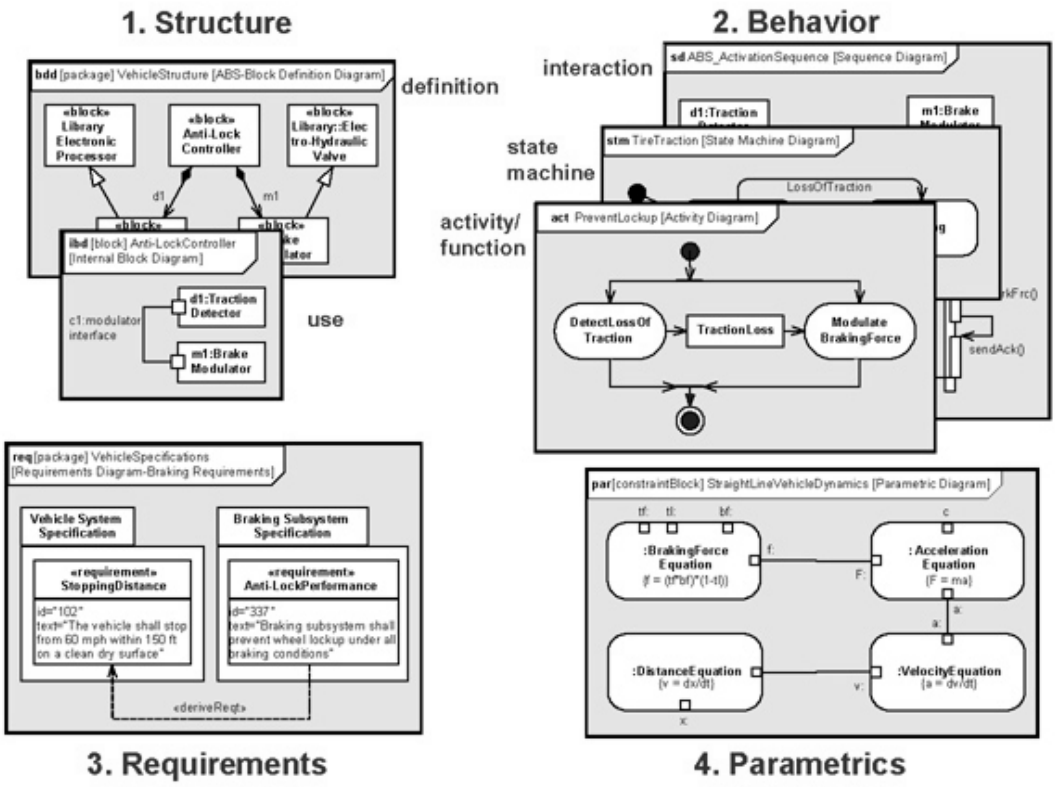


Abb. 2.12: Die 4 Säulen der SysML [10]

- Das Anforderungsdiagramm: Es ist das erste neue Diagramm der SysML und dient der Modellierung von Systemanforderungen und Beziehungen zwischen Anforderungen oder Systemelementen.

- Das Blockdefinitionsdiagramm: Im Systems Engineering werden keine Klassen benutzt, sondern Systembausteine oder Blöcke. Dementsprechend wurde das Klassendiagramm zum Blockdiagramm. Mit diesem Diagramm wird die Struktur eines Systems abgebildet, es werden aber auch Wertetypen damit definiert.

- Das interne Blockdiagramm: Abgeleitet vom Kompositionsstrukturdiagramm erfüllt dieses Diagramm auch ähnliche Aufgaben, nämlich die Darstellung interner Strukturen und Eigenschaften von Blöcken. Es zeigt die Verbindungen interner Teile eines Blocks sowie deren Schnittstellen.

- Das Zusicherungsdiagramm: Es ist das zweite neue Diagramm der SysML und dient der Modellierung von Zusammenhängen zwischen Eigenschaften von Systembausteinen anhand von parametrischen Beziehungen.

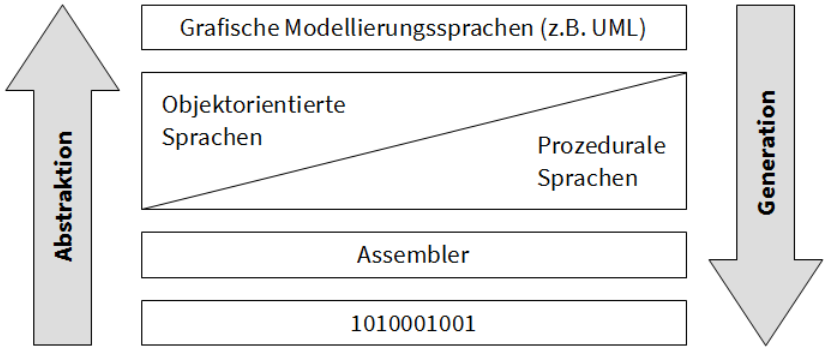


Abb. 2.13: Steigende Abstraktion der Programmiersprachen [195]

2.3 Modellgestützte Entwicklungsmethoden

Im Kontext der Entwicklungsmethoden, die sich in irgendeiner Weise auf Modelle beziehen, auf Modellen aufbauen oder Modelle benutzen, existieren in der Fachwelt unzählige Abkürzungen, z.B. MDE, MBE, MDD, MDSD, MDSE, MDA, MBSE, um nur ein paar zu nennen[37]. Das M steht dabei für Modell, darauf folgend stehen B oder D für basierend (vom englischen based) oder getrieben (vom englischen driven und der Bedeutung antreiben). Danach steht A für Architektur, D für Entwicklung (vom englischen development), mit E ist das Ingenieurwesen gemeint (vom englischen engineering) und S bezeichnet Software oder System. In jedem Fall folgt einem S an dritter Stelle noch ein abschließender vierter Buchstabe, meist D oder E, Bedeutung wie gehabt. Laut Brambilla et al. [27] werden diese Bezeichnungen oft unter „Model Driven Whatever“ zusammengefasst. Einige dieser Akronyme kommen in der Fachliteratur sehr häufig vor, klare Definitionen oder Abgrenzungen sind in den meisten Fällen aber nicht gegeben. Abhängig vom Einsatzbereich der Methoden, verlaufen auch Unterschiede und Trennlinien und bei genauer Betrachtung ergeben sich dann fließende Übergänge. Hier sollen im Folgenden die beiden Methoden vorgestellt werden, die im Kontext dieser Arbeit von Bedeutung sind.

2.3.1 MBE/MDE

Die größten Fortschritte in der Entwicklung der Programmiertechniken wurden durch Erhöhung der Abstraktionsebene erzielt. Die Abstraktion führte dabei immer weiter von der ausführenden Maschine weg, siehe Abb. 2.13. In knapp 50 Jahren von Lochkarten zu Programmiersprachen der dritten und vierten Generation. Die Entwicklung der modellbasierten Ansätze wird oft als der nächster Abstraktionsschritt angesehen [166, 158, 71, 84, 20].

[37] Jean Bezivin hat in einer seiner Arbeiten [42] und auf seinem Weblog „Models everywhere“ eine Reihe geläufiger Akronyme und deren Bedeutung aufgelistet [9].

Bezeichnungen wie modellbasierte Entwicklung (MBE, engl. Model-Based Engineering) und modellgetriebene Entwicklung (MDE, engl. Model-Driven Engineering) dienen als Sammelbegriffe für technische Entwicklungsmethoden, die sich auf Modelle beziehen oder auf Modellen beruhen. Derartige Bezeichnungen werden meist mit Softwareentwicklung in Verbindung gesetzt, eine Disziplin- oder Domänenausrichtung ist jedoch nicht direkt gegeben.

Ganz generell werden in den modellbezogenen Entwicklungsmethoden Modelle in den Vordergrund gestellt. In den sogenannten Code-zentrierten Entwicklungsmethoden kommen Modelle meist nur am Anfang der Entwicklung zum Einsatz, als Skizzen, Gedankenstützen oder zur Dokumentation. Änderungen werden dann oft nicht nachgezogen und die Modelle verlieren im Laufe der Entwicklung den Projektbezug. Beim Einsatz von modellgestützten Methoden sind Modelle primäre Artefakte. Alle Entscheidungen, Änderungen, Ergänzungen fließen direkt dem Modell zu, im Modell ist alles Wissen der Entwicklung gesammelt und abrufbar. Das Modell spiegelt zu jedem Zeitpunkt den Stand der Entwicklung wieder. Durch den formalen Charakter der Modelle, können alle Informationen maschinell weiterverarbeitet werden, z.B. zur Codeerstellung.

Brambilla et al. [27], sowie Whittle et al. [198] verstehen die Unterschiede zwischen den Ansätzen MBE, MDE, und MDD(MDA) in der Art und Weise und im Umfang der Modellnutzung. Die modellgetriebenen Entwicklungsansätze (MDD) haben hier einen klaren Softwarebezug. Sie sehen vor, dass Modelle den gesamten Entwicklungsprozess hindurch als primäre Artefakte dienen. Es wird praktisch nur mit Modellen gearbeitet. Alle Produkte, z.B. Programmcode, sollen (halb)automatisch mittels Transformation aus Modellen erzeugt werden. Statt MDD wird oft auch MDSD geschrieben, wobei das S für Software steht. Die Bezeichnung Model-Driven Architecture (MDA[38]) ist ein geschützter Name der OMG und beschreibt im Wesentlichen ein MDD Entwicklungsverfahren, basierend auf OMG Standards. Die MDE Ansätze sind dagegen weiter gefasst [94, 45, 85].

Nicht nur die konkrete Tätigkeit der Softwareentwicklung, sondern alle modellbezogenen Aspekte im Softwarekontext sind damit bezeichnet, als Beispiel werden Model-driven Reverse Engineering oder Model-driven Evolution genannt. Statt MDE wird auch oft MDSE geschrieben, auch hier steht das S für Software. Die MBE Ansätze sind noch genereller gefasst, ein Softwarebezug muss nicht gegeben sein. Modelle spielen eine große Rolle. Das Ergebnis wird zwar basierend auf Modellen entwickelt, es muss aber nicht direkt aus Modellen entstehen. Diese Beschreibung lässt sich gut durch ein Mengendiagramm darstellen, siehe Abb. 2.14. Begriffen wie MDSD und MDSE soll der Buchstaben S einen Softwarebezug verleihen, das S kann aber auch für System stehen, was die Bedeutung erheblich verschiebt, siehe nächster Abschnitt (2.3.2).

Im Folgenden soll der Fokus wieder stärker auf den Softwarebereich gelenkt werden, in diesem Kontext also auf MDE und MDD bzw. MD*. Die ersten Schritte in Richtung grafisches Programmieren stammen aus den 1980er Jahren, Computer-aided Software Engineering (CASE) hat einerseits viel Aufmerksamkeit auf sich gezogen, konnte aber

[38] OMG MDA, `http://www.omg.org/mda`

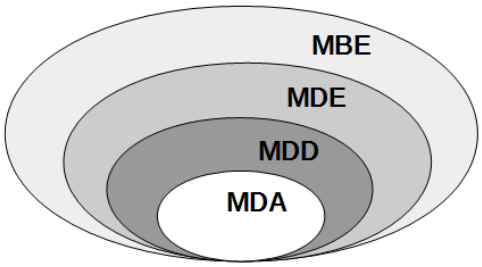


Abb. 2.14: Beziehung zwischen modellbasierten Methoden [27]

andererseits die hohen Erwartungen nicht erfüllen und war kommerziell wenig erfolgreich [158, 41]. Harel und Rumpe [73] vermuten sogar, dass das Scheitern des CASE Ansatzes eventuell nachhaltigen Schaden für die Bemühungen zur Etablierung der Modellierung bedeutet haben könnte.

Die Entwicklung objektorientierter Sprachen und Methoden führte letztlich zur Entstehung der UML, siehe Abschnitt 2.2.1. Und damit war, zumindest für die Softwarewelt, ein gemeinsamer Modellierungsstandard geschaffen, auf dem weiter aufgebaut werden konnte. Die OMG tat genau das und stellte im Jahr 2000 MDA vor. Sie formulierte MDA rund um UML und eine Reihe ihrer weiteren Standards, wie MOF, XMI und OCL usw. MDA bietet dabei einen konzeptuellen Rahmen, um den Einsatz von MDD zu unterstützen [166]. MDA ist aber nicht mit MDD gleichzusetzen, sondern nur eine spezielle, standardisierte Ausprägung davon. MDA ist auch nicht gleich MDE, da MDE, wie bereits weiter oben beschrieben, technisch weiter gefasst ist. MDA war auch nicht die Erfindungsstunde von MDE oder MDD, Ansätze in diese Richtung gab es bereits zuvor [85, 198]. Die OMG verfolgte mit MDA einfach ihr generelles Anliegen zur Standardisierung, um dadurch Interoperabilität (von Softwarewerkzeugen), Portabilität (Plattformunabhängigkeit) und Wiederverwendbarkeit zu ermöglichen. Eine weitere MDD Ausprägung stellen die Software Factories [69] von Microsoft dar.

Die MDE Ansätze verfolgen, unabhängig von der spezifischen Ausprägung, mehrere Ziele. Durch Anhebung der Abstraktionsstufe soll die Komplexität in Software verringert werden. Dieses Konzept wird als Trennung der Belange (engl, Separation of Concerns, oder kurz SOC) bezeichnet. Der Entwickler soll Lösungen auf einer Ebene erarbeiten, die nur die Aufgaben der jeweiligen Domäne enthalten, also Geschäftslogik. Problemstellungen aus darunterliegenden Ebenen, etwa der Plattformebene (Java, .Net) oder der Systemebene (Betriebssystem) sind abstrahiert. Die Wiederverwendbarkeit von Software soll gesteigert werden, die Wartung und Pflege vereinfacht. Die voll- oder halbautomatische Softwaregenerierung aus entsprechenden Modellen soll dazu beitragen. In Summe soll durch die MDE Ansätze die Produktivität der Softwareentwicklung gesteigert werden, was automatisch zu Zeit- und Kostenersparnis führen soll.

Die Grundprinzipien dieser Methodiken sind, wie Völter [190] festgestellt hat, alle gleich, bzw. basieren sie auf dem gleichen Ansatz, siehe Abb. 2.15 für einen Überblick. Bézivin [40]

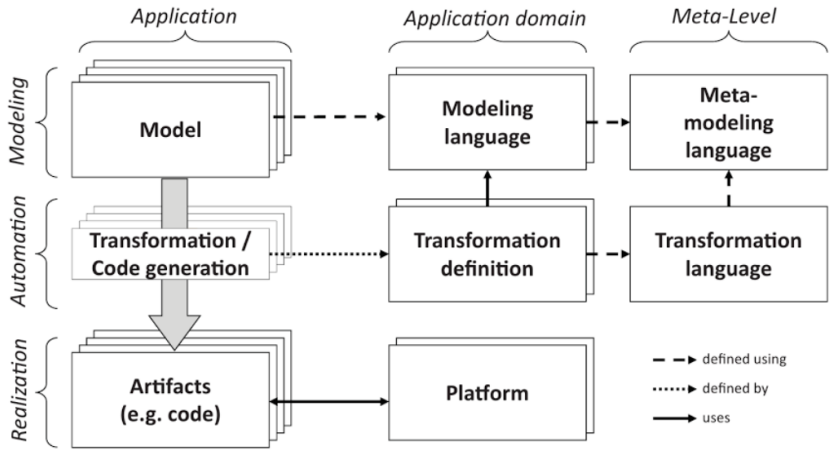


Abb. 2.15: Überblick der MDSE Methodik [27]

formulierte das zugrundeliegende Paradigma „Everything is a model“. Dieses Paradigma ist der Objektorientierung entlehnt und lautet dort „Everything is an object“. Ausgangspunkt ist das Modell, es soll so umfassend und detailliert sein, um ein Softwaresystem in allen Aspekten abzubilden. Daraus soll dann halb- oder vollautomatisch eine lauffähige Anwendung generiert werden können. Die Generierung erfolgt dabei meist in mehreren Transformationsschritten. In jedem dieser Schritte wird das Ausgangsmodell in ein spezielles, dem Einsatz entsprechendes Zielmodell übersetzt. Diese Mehrschrittigkeit ergibt sich aus dem Anspruch an Interoperabilität und Portabilität. Dass zuerst vollkommen technologie- und plattformneutrale Modell wird Schritt für Schritt durch Transformation an die Einsatzumgebung angepasst. MDA enthält dafür drei Abstraktionsebenen, siehe Abb. 2.16. Laut dem Paradigma von Bézivin sind die Transformationen ebenfalls als Modelle anzusehen.

Die Ebene CIM (Computation independant model[39]) ist soweit abstrahiert, dass darunterliegende Systemkontexte (Systemstruktur, Plattform, Betriebssystem) nicht spürbar sind. Ein CIM beschreibt ein Anwendungsumfeld und enthält Anforderungen, oft wird der Begriff Domänenmodell verwendet. Aus dem CIM wird in einem Transformationsschritt das PIM (Platform independant model[40])erzeugt. Es wird sich dabei der Ausführungsebene angenähert. Das PIM beschreibt die Strukturen und Funktionen eines Systems, dies jedoch immer noch auf hoher Abstraktionsstufe. Aus dem PIM wird wiederum in einem Transformationsschritt das PSM (Platform specific model[41]) erzeugt. Hier spielt erstmals die Technologie einer konkreten Plattform eine Rolle. Aus dem PSM soll in einem letzten Transformationsschritt lauffähiger Code erzeugt werden. Die bisherigen Transformationen liefen auf einer Modell-zu-Modell Basis ab, aus einem Modell höherer Abstraktionsstufe wurde ein spezielleres Modell erzeugt. Die letzte Transformation läuft

[39]) Zu Deutsch „ausführungsunabhängiges Modell“.
[40]) Zu Deutsch „plattformunabhängiges Modell“.
[41]) Zu Deutsch „plattformspezifisches Modell“.

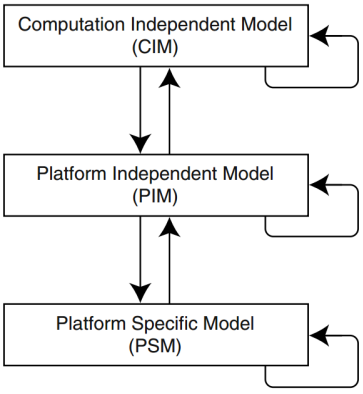


Abb. 2.16: MDA Architekturebenen und Modelltransformationen [113]

im Unterschied dazu auf einer Modell-zu-Code Basis ab, aus einem Modell wird Quellcode in Textform erzeugt. Die tatsächliche Modellierungsarbeit soll in den Ebenen CIM und PIM erfolgen, der Rest durch entsprechend der Plattform und des Systems angepasste Transformationen erledigt werden.

Diese Konzepte und Vorteile klingen in der Theorie alle sehr vielversprechend. Um die MD* Methoden gab es einen regelrechten Hype. Der Jubel ist leiser geworden, eine gewisse Ernüchterung hat sich eingestellt. Brambilla et al. beschreiben diesen Umstand sehr treffend mit einem Diagramm, siehe Abb. 2.17. Sie sagen, dass heute, auf Basis des gesammelten Wissens und der bisherigen Erfahrungen MDE betreffend, besser über eine Projekteignung entschieden werden kann und dass bald das Plateau der Produktivität erreicht sein wird. Aber was ist in der Industrie und der Wirtschaft davon angekommen bzw. spürbar geworden? Wird MDE in der Praxis tatsächlich verwendet? Wie bereits beschrieben, gibt es seit dem Scheitern der CASE Ansätze in weiten Teilen der Fachwelt gewisse Vorbehalte gegen das Programmieren mit Modellen. Whittle et al. [198] führten eine groß angelegte Studie zur Erhebung des Nutzungsgrads von MDE in der Praxis durch, mit zum Teil sehr überraschenden Ergebnissen.

So hat sich herausgestellt, dass gewisse MDE Formen weitverbreitet zum Einsatz kommen. Es wird berichtet von „[...] industrieweiten Anstrengungen zur Definition von präzisen Modellen für ganze Anwendungsdomänen[...]“ bis zur „[...]eingeschränkten Codeerzeugung für einzelne Anwendungen[...]“. Whittle et al. fanden heraus, dass der erfolgreiche Einsatz von MDE meist auf DSLs basiert. Und auch, dass die automatische Codeerzeugung kein Hauptargument für den Einsatz von MDE ist, vielmehr liegt offenbar der größte Vorteil in der ganzheitlichen Unterstützung zur Trennung der Belange (SOC) und zur Dokumentation einer guten Softwarearchitektur. Wie Whittle et al. feststellten, geht der beobachtbare Trend stark in Richtung DSLs. Völter [191] sieht darin eine natürliche Evolution des Themas.

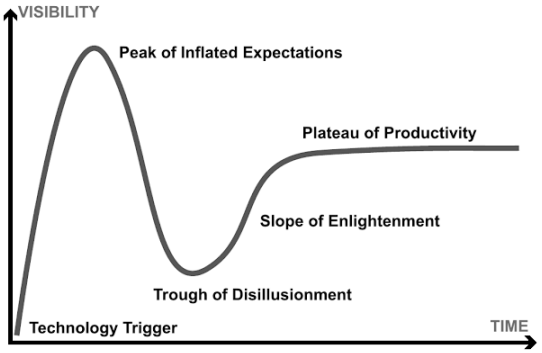


Abb. 2.17: Technology hype cycle [27]

2.3.2 MBSE/MDSD

Modellbasierte Systementwicklung (MBSE[42]), manchmal auch als MDSD (Model-driven systems design) bezeichnet [55], ist eine noch recht junge Form des Systems Engineering. Das Systems Engineering hingegen ist eine relativ alte Disziplin, Weilkiens sieht die Anfänge zum Teil schon vor über 5000 Jahren im ägyptischen Pyramidenbau [195]. Der INCOSE spricht von der Entstehung des Begriffs in den Bell Telephone Laboratories der frühen 1940er Jahre, das Konzept wird in die frühen 1900er Jahre datiert. Engineering als Hauptwort bedeutet im Englischen Ingenieurswesen oder technische Wissenschaft, kann aber auch als „die Tätigkeit des technischen Entwickelns“ gedeutet werden. Ganz grundlegend beschreibt der Begriff Systems Engineering also Prozesse und Methoden, um erfolgreich komplexe Systeme zu entwickeln, eine vereinfachte Darstellung des technischen Prozesses im Systems Engineering ist in Abb. 2.18 dargestellt. Wichtiges Merkmal dabei sind die Feedback-Pfade, durch die der Prozess einen iterativen Charakter erhält. Korrekturen an Anforderungen, Spezifikationen und Design werden dadurch im Prozess vorgesehen und somit einfacher zu handhaben.

Das Systems Engineering ist dabei als interdisziplinärer Ansatz zu verstehen und umspannt den gesamten Systemlebenszyklus von der anfänglichen Planung über die Realisierung bis zur Entsorgung [195, 192]. Dabei kann ein System jedes technisch realisierbare Objekt darstellen, etwa ein Smartphone oder auch ein Spaceshuttle. Beim Systems Engineering werden sowohl technische, als auch wirtschaftliche Aspekte betrachtet. Das umfasst die Aufgabenbereiche Projekt-, Risiko- und Requirementsmanagement, Requirements Engineering, Systemarchitektur, Systemverifikation, Systemvalidierung und Systemintegration. Die Details bleiben dabei außen vor. Nur Aspekte die auf Systemebene relevant sind, werden betrachtet [195].

Im Laufe der Zeit haben sich in den jeweiligen Einsatzgebieten technische Normen, internationale Standards und gesetzliche Auflagen entwickelt. So ist bei der Entwicklung eines Flugzeugs für den Personenverkehr mehr Aufwand in die Einhaltung von Richtlinien

[42] engl. Model-based systems eingineering

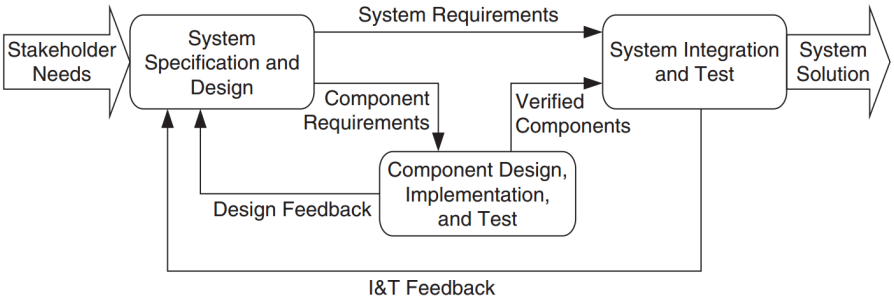


Abb. 2.18: Vereinfachter technischer System Engineering Prozess [63]

und Vorschriften, was etwa Dokumentation und Rückverfolgbarkeit betrifft, zu investieren, als bei der Entwicklung eines Taschenrechners. Aus den diversen Anforderungen ergaben sich zwangsläufig Entwicklungsansätze mit sehr starkem Bezug zu Dokumenten, sie stellen die primären Artefakte dar. Delligatti beschreibt recht eindrucksvoll, wie in derartigen Ansätzen aus kleinen Änderungen, die in einer Fülle von Dokumenten nachgezogen werden müssen, sehr schnell große Probleme entstehen können, und zwar durch hohen Zeitaufwand, Fehleranfälligkeit und letztlich daraus resultierenden Kosten [54].

Mit dem INCOSE[43] wurde 1995 eine internationale Organisation geschaffen, deren Ziele „die Entwicklung und Förderung des Systems Engineering“ [195] sind. Ähnlich der OMG ist der INCOSE eine Non-Profit Organisation, unter den Mitgliedern finden sich Firmen aus Industrie und Wirtschaft, sowie Bildungsinstitute und Privatpersonen. Als ein zentrales Anliegen des INCOSE wurde der Übergang von dokumentenzentrierten zu modellbasierten Methoden im Systems Engineering formuliert. Dazu wurde eine Arbeitsgruppe, mit Fokus auf Model Driven Systems Design (MDSD), ins Leben gerufen [62, 21].

Diverse Modelle und Diagramme kommen im Systems Engineering schon seit Jahren zur Anwendung. Generell waren die Einsatzmöglichkeiten von Modellen aber sehr eingeschränkt. Mit Modellen wurden immer nur Teilaspekte von Systemen betrachtet und analysiert. Es wurden noch keine kompletten Systemmodelle erstellt, die alle Aspekte gleichermaßen abdecken. Dieser Ansatz wurde erst mit dem MBSE eingeführt [63, 116, 62].

Als die drei Säulen des MBSE bezeichnet Delligatti [54] das Vorgehen nach einer Modellierungsmethode, den Einsatz einer Modellierungssprache und damit einhergehend ein entsprechendes Modellierungswerkzeug, also Software. Modellierungsmethoden haben den Sinn, durch vollständige Schritt-für-Schritt beschriebene Prozesse ein planvolles, strukturiertes und dokumentiertes Vorgehen zu ermöglichen. Die Ergebnisse dieser Prozessschritte werden dann mit einem Modellierungswerkzeug in der gewählten Modellierungssprache festgehalten. So entsteht schrittweise ein komplettes Systemmodell. Dieses Modell soll das geplante System in einem Umfang darstellen, der den Systemanforderungen gerecht wird,

[43] Nachfolgeorganisation der 1990 gegründeten US-amerikanischen NCOSE (National Council on Systems Engineering)

und je nach definierten Anforderung an das Modell einen bestimmten Detailgrad bieten. Das Systemmodell wird beim MBSE zum primären Artefakt erklärt. „Das Modell wird zur Quelle aller relevanten Informationen“ [195]. Alle anderen Artefakte rücken in den Hintergrund, werden sekundär. Bei Bedarf sollen sie automatisch aus dem Systemmodell generiert werden. Die Dokumente verschwinden dadurch nicht, aber sie sind nicht mehr die Quelle der Information. Sie werden zu verschiedenen Sichten auf das Systemmodell [54, 195], siehe dazu auch Abschnitt 2.1.3.

Zwei relativ bekannte und unabhängige Modellierungsmethoden sind SYSMOD von Weilkiens [195] und OOSEM vom INCOSE [63]. Als Modellierungswerkzeug findet die Open Source Software Papyrus immer mehr Beachtung, siehe dazu auch den Abschnitt 2.5.5.

Delligatti warnt aber auch vor dem Mythos des MBSE. Und zwar, dass MBSE jede Projektaufgabe einfacher macht und in jedem Abschnitt des Systemlebenszyklus Kosten spart, dass MBSE also sozusagen „automagisch“ zu weniger Problemen und besseren Lösungen führt. Aber, so Delligatti weiter, MBSE kann die Schwierigkeit des Systementwurfs gar nicht eliminieren. Brooks spricht in diesem Zusammenhang von einer essentiellen Komplexität, die Systemen innewohnt. Sie kann nicht reduziert werden [32]. Durch den Einsatz von MBSE wird also die eigentliche Entwicklungsarbeit bei der Erzeugung eines Systems nicht weniger oder einfacher. Und anders als in der modellgetriebenen Softwareentwicklung, deren Ziel es ist aus dem Modell, mehr oder weniger automatisch, das gewünschte Softwareprodukt zu generieren, kann im MBSE aus dem Modell kein fertiges System generiert werden. Der praktische Nutzen des MBSE liegt im Potential des Anwendungsspektrums, das durch die Nachhaltigkeit eines ganzheitlichen Systemmodells ermöglicht wird. Nur einige der Vorteile, die aus der konsequenten Anwendung von MBSE erwachsen, werden im Folgenden beschrieben:

- Durch eine automatische Verarbeitung können Systemmodelle analysiert, verifiziert und validiert werden. Die Simulation von Modellen kann bereits in frühen Entwicklungsphasen Schwachstellen oder Fehler aufzeigen. Weiters können benötigte Dokumente automatisch aus dem Modell erzeugt werden.
- Das Modell eines Systems kann immer wieder verwendet werden, z.B. das Systemmodell einer Standardkomponente, die in vielen Systemen verbaut wird.
- Änderungen können rasch und zentral durchgeführt werden.
- Unterschiedliche Sichten auf das Modell ermöglichen eine Nutzung in allen Produktlebenszyklen, z.B. für die Projektleitung, die Dokumentation, den Support oder die Ausbildung.

Wo liegt nun aber der Unterschied zwischen MBSE und MDSD? Manchmal wird der Begriff synonym verwendet [55]. Wie aber bereits im vorigen Abschnitt zu MBE/MDE beschrieben, wird das Design eines Systems im Allgemeinen dem weiter gefassten und

technisch tiefgreifenderen Begriff des Engineering unterstellt. MDSD wäre somit als Teilbereich von MBSE zu verstehen. Delligatti [54] meint, dass die Linie zwischen Design und Development mit zunehmendem Grad der MBSE-Entwicklungsreife verblast.

2.4 Versionierung

Bei der Entwicklung von Software ist Arbeitsteilung nicht mehr wegzudenken. Teamarbeit ist mittlerweile ein Projektstandard. Das Arbeiten in Teams erfordert meist einen gewissen Anteil an Kooperation und sollte, um nicht im Chaos zu enden, geplant, strukturiert und kontrolliert erfolgen. Die Versionierung nimmt dabei eine zentrale Rolle ein. Es erfüllt, grob umrissen, die folgenden Aufgaben: Herstellung einer konsistenten Sicht auf den Projektzustand, Verwaltung unterschiedlicher Entwicklungszweige, Archivierung der Versionshistorie und Unterstützung der Strukturierung von kooperativer Arbeit. In diesem Kapitel wird zuerst die Basis der Versionierung beleuchtet, anschließend die Grundlagen der Versionierung erläutert und danach der Bogen zur Modellversionierung als Spezialdisziplin der Versionierung gespannt.

2.4.1 Basis der Versionierung

Als Versionierung wird der Prozess der Verarbeitung von Daten mit einem Versionsverwaltungssystem bezeichnet. Systeme zur Versionsverwaltung, oder Versionsverwaltungssysteme (VCS[44]) sind ihrer Herkunft nach Werkzeuge des Softwarekonfigurationsmanagements (SCM[45]). SCM wiederum ist eine Ausprägung des Konfigurationsmanagements (CM[46]), zugeschnitten auf die Entwicklung von Software [26]. Konfigurationsmanagement hat in seinem Ursprung nichts oder nur wenig mit Software zu tun. CM entstand in den 1950er Jahren und stammt aus der Luft- und Raumfahrtindustrie des US-Militärs [115]. Die Großprojekte der Rüstungsindustrie erreichten damals Dimensionen, die ein standardisiertes Vorgehen notwendig machten. CM wurde in der Folge zu einem militärischen[47] und später auch zu einem internationalen Standard[48] entwickelt [37, 115]. Mittlerweile kann SCM als Teil von CM betrachtet werden, da kaum ein System auf Software verzichten kann.

Die Ziele, Aufgaben und Vorschriften von CM sind allgemein gehalten und ganz generell auf eine Produktentwicklung, bzw. ein Produkt bezogen. SCM hingegen ist klar auf Software als das Produkt der Softwareentwicklung zugeschnitten. Die zwei Hauptunterschiede zwischen CM und SCM sieht Tichy [185] darin, dass sich Software im Vergleich zu Hardware einfacher ändern läßt und sich daher auch häufiger ändert und das SCM vergleichsweise besser automatisierbar ist. Die Aufgaben von SCM beschreibt Tichy [186] wie folgt:

[44] engl. Version Control System
[45] engl. Software Configuration Management
[46] engl. Configuration Management
[47] US MIL-STD-480
[48] aktuell ISO 10007:2017

- Identifikation: Jedes Artefakt der Konfiguration muss eindeutige identifizierbar sein.

- Versionskontrolle: Jede Änderung oder Artefaktversion wird dauerhaft gespeichert (Delta-Storage zur Speicherplatzreduktion).

- Konfigurationsselektion und Systemerstellung: Änderungen bzw. Revisionen von Artefakten müssen für jede Systemerstellung ausgewählt werden, wie die Systemerstellung sollte das automatisch erfolgen.

- Änderungsmanagement: Jede Änderung benötigt eine formale Änderungsanfrage.

Ein Teil dieser Aufgaben kann mit einem VCS bewerkstelligt werden. VCS sind daher unverzichtbarer Bestandteil jedes SCM Systems. Im Englischen gibt es übrigens eine Bedeutungsüberlagerung, VCS werden hier oft auch als SCM bezeichnet, gemeint ist damit jedoch „Source Control Management“ oder „Source Code Management“, was in der Bedeutung wieder gleichwertig zu VCS ist.

Das erste System seiner Art, SCCS, wurde ab 1972 von Rochkind in den Bell Labs entwickelt und 1975 vorgestellt [154]. Fast zehn Jahre später folgte das System RCS von Tichy [184]. Darauf aufbauend wurde wiederum einige Jahre später CVS[49] entwickelt. CVS konnte sich lange Zeit überaus großer Beliebtheit erfreuen und war damals wohl das am weitesten verbreitete VCS, nicht zuletzt durch die Open Source Community. Mittlerweile ist CVS allerdings überholt und wird nicht mehr weiterentwickelt, die letzte stabile Version stammt aus dem Jahr 2008.

Ausgehend von diesen ersten VCS wurden bis heute einige sehr erfolgreiche Systeme entwickelt. Vorherrschend sind heute sicher die Open Source Lösungen der letzten Jahre. Systeme wie SVN[50] oder Git[51] sind aus dem Alltag von Softwareentwicklern nicht mehr wegzudenken. Das zur Zeit wohl populärste und am weitesten verbreitete VCS ist Git.

2.4.2 Begriffe und Techniken der Versionierung

Die oben beschriebenen Kernaufgaben haben sich seit den ersten VCS nicht geändert, allein die Konzepte und Techniken wurden weiterentwickelt. Versionierung kann entlang mehrerer Dimensionen differenziert werden. Altmanninger et al. [17] haben, bezugnehmend auf Arbeiten von Conradi und Westfechtel [51], sowie Mens [130] eine Terminologie, sowie eine Domänenanalyse zu VCS veröffentlicht. Die dort vorgestellte Einteilung und Terminologie soll hier weitestgehend beibehalten werden, wenn im Folgenden die wichtigsten Aspekte der Versionierung besprochen werden.

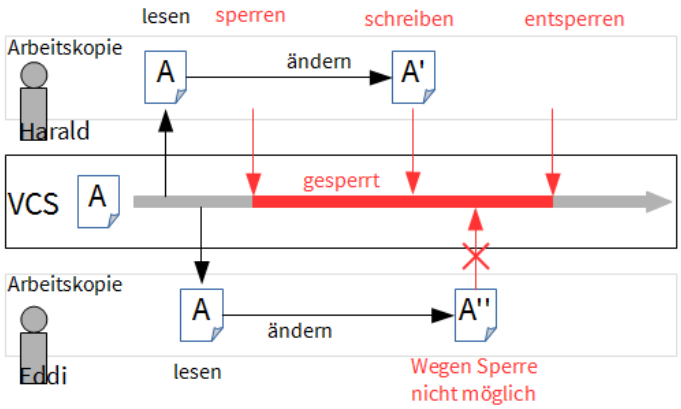


Abb. 2.19: Pessimistische Versionskontrolle

Pessimistische Versionierung

Bei der kooperativen Arbeit an Softwareartefakten ergeben sich grundsätzlich zwei Möglichkeiten. Ein Entwickler, z.B. Harald, sperrt das Artefakt an dem er arbeitet, damit erhält er exklusive Bearbeitungsrechte. Für andere Entwickler, z.B. Eddi, ist ein gesperrtes Artefakt nur lesbar, somit nicht veränderbar. Erst wenn Harald die Sperre wieder aufhebt, kann auch Eddi das Artefakt bearbeiten, jedoch ebenfalls nur exklusiv. Ein Artefakt kann also immer nur von einem Entwickler bearbeitet werden, siehe Abb. 2.19. Dieser Ansatz wird als pessimistische Versionskontrolle bezeichnet. Das ist die einfachste Möglichkeit zur Versionierung, wird aber mit steigender Entwicklerzahl relativ schnell unpraktikabel, z.B. wenn Sperren nicht wieder aufgehoben werden.

Optimistische Versionierung

Etwas anders verhält es sich bei der zweiten Möglichkeit, der optimistischen Versionskontrolle. Hier kann jedes Artefakt zu jeder Zeit von mehreren Entwicklern bearbeitet werden. Moderne VCS verwenden standardmäßig diese Art der Versionierung, bieten aber meist auch die Möglichkeit zum Sperren von Artefakten. Die Flexibilität, die mit optimistischer Versionierung gewonnenen wird, birgt aber ein potentielles Problem. Bei der gleichzeitigen Bearbeitung können recht einfach Konflikte zwischen unterschiedlichen Versionen von Artefakten entstehen, siehe Abb. 2.20. Harald und Eddi haben jeweils eine Kopie eines Artefakts, beide bearbeiten das Artefakt, Eddi speichert seine Änderungen zuerst ins VCS, sobald Harald seine Änderungen ins VCS speichern will, tritt ein Konflikt auf. Die geänderte Artefaktekopie von Harald würde die von Eddi bearbeitete Version überschreiben. Das Überschreiben ist natürlich nicht zielführend und daher keine Option.

[49] http://savannah.nongnu.org/projects/cvs
[50] https://subversion.apache.org
[51] https://git-scm.com

Die Änderungen beider Versionen müssen in einer Version vereint werden, dieser Vorgang wird als das Zusammenführen (engl. merge) bezeichnet.

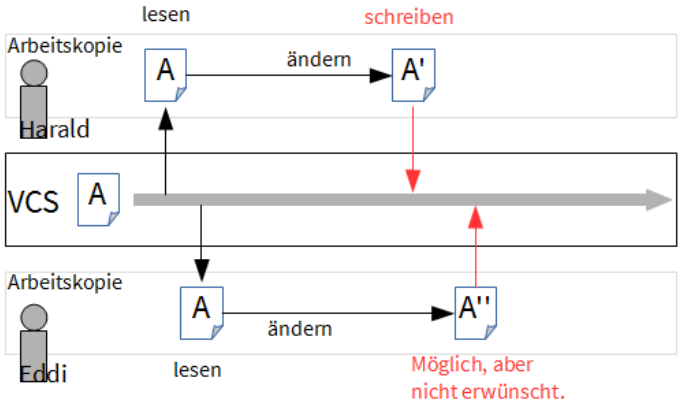


Abb. 2.20: Optimistische Versionskontrolle

Zusammenführung (Merge)

Beim Zusammenführen von Artefakten gibt es drei unterschiedliche Varianten. Bei der rohen Zusammenführung (engl. raw merge) werden die einzelnen Änderungen zweier Artefakte nacheinander auf einem Artefakt durchgeführt, siehe Abb. 2.21(a) Diese Art der Zusammenführung kann zum Datenverlust durch Überschreiben bzw. zu Inkonsistenzen führen. Die Zwei-Weg-Zusammenführung vergleicht beide Artefaktversionen und erzeugt daraus die neue Version. Diese Art der Zusammenführung ist ebenfalls nicht ideal. Da nur zwei Zustände verglichen werden, kann nicht eruiert werden, ob Elemente gelöscht oder hinzugefügt wurden, siehe Abb. 2.21(b). Bei der Drei-Weg-Zusammenführung werden die beiden Artefaktversionen mit ihrer gemeinsamen Vorgängerversion (engl. ancestor) verglichen, somit können alle vorgenommenen Änderungen beider neuen Versionen erkannt werden, siehe Abb. 2.21(c). Der Nachteil des Mehraufwands beim Artefaktvergleich wird durch bessere Änderungserkennung und damit höhere Genauigkeit und Treffsicherheit wettgemacht. Die Drei-Wege Variante ist den beiden anderen Varianten klar vorzuziehen und deshalb das vorherrschende Verfahren in modernen VCS.

CVCS versus DVCS

Versionsverwaltungssysteme können zentralisiert oder verteilt aufgebaut sein. Ein zentralisiertes VCS (CVCS[52]) hat eine ausgeprägte Client-Server Struktur. Der Server verwaltet die Projektablage bzw. den Artefaktspeicher[53] und somit die Versionsgeschichte. Der Client erhält vom Server sogenannte Arbeitskopien der Artefakte und sendet Änderungen zurück zum Server. Zum Arbeiten mit dem Repository wird eine Verbindung zum Server

[52] engl. Centralized VCS
[53] engl. Repository

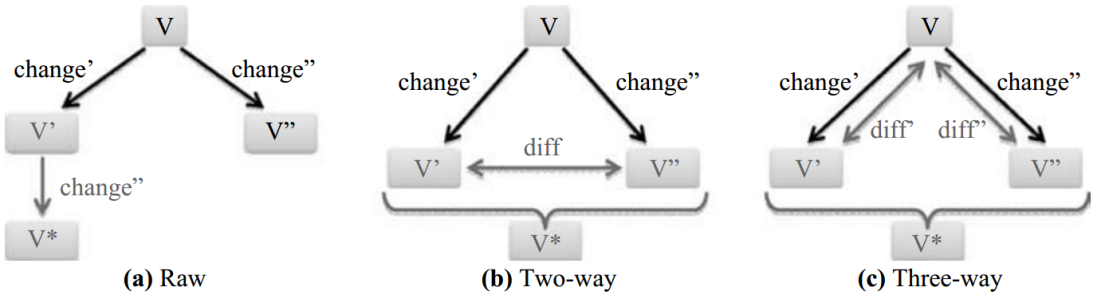


Abb. 2.21: Varianten der Zusammenführung [17]

benötigt. Bei verteilten VCS (DVCS[54]), auch dezentrale VCS genannt, gibt es eigentlich keinen zentralen Server, bzw. wäre theoretisch keiner notwendig. Denn jeder Benutzer hält eine vollständige Kopie eines Repositories, mit der gesamten Versionsgeschichte. So kann er selbständig ohne Verbindung zu einem Server arbeiten. Die Änderungen die ein Benutzer vornimmt, also die Fortschritte der Versionsgeschichte die er erstellt, werden dann von Repository zu Repository abgeglichen und weitergegeben. Um diesen Vorgang zu vereinfachen wird aus praktischen Gründen dennoch meist ein zentrales Repository verwendet.

Drei-Weg-Zusammenführung

Das Zusammenführen von Artefakten wird in modernen VCS überwiegend als Drei-Weg-Zusammenführung umgesetzt. Der Vorgang der Zusammenführung ist dabei unterteilt in die Phasen Vergleich, Konflikterkennung, Konfliktlösung und Zusammenführung. Die vier Phasen werden in den folgenden Abschnitten erläutert.

Vergleich Der Vergleich von Artefakten erfolgt auf Basis von Zuständen (engl. state-based) oder Änderungen (engl. change-based). Werden zur Ermittelung der Artefaktunterschiede die Zustände der Artefakte herangezogen, so wird der Zustand der Vorgängerversion jeweils mit dem Zustand der beiden veränderten Versionen verglichen. Bei der Ermittelung der Unterschiede anhand der Änderungen werden Änderungsprotokolle verglichen, also das Protokoll aller durchgeführten Änderungen zwischen Vorgängerversion und jeweiliger Nachfolgeversion. Diese Variante kann einerseits zu besseren Ergebnissen führen, ist andererseits aber von der Unterstützung des verwendeten Editors abhängig.

Zum Finden einer Übereinstimmung (engl. match) zwischen Artefaktelementen kommen ganz generell zwei Möglichkeiten in Frage, Heuristiken und UUIDs (engl. Universally unique identifiers[55]). Bei der Verwendung von Heuristiken werden strukturelle Ähnlichkeiten berechnet und bewertet, beim Vergleichen von Textdaten z.B. die Ähnlichkeit zweier Zeilen oder die Ähnlichkeit zweier Worte. Werden z.B. Datenstrukturen verglichen,

[54] engl. Distributed VCS
[55] zu Deutsch „universell eindeutige Identifikatoren“

können Benennung, Anzahl und Art der Elemente (z.B. Attribute) etc. zur Berechnung eines Ähnlichkeitswerts herangezogen werden. Die Verwendung von Heuristiken ist vom Editor unabhängig und sehr flexibel aber aufwändiger als die zweite Variante, UUIDs. Eine UUID ist ein Elementattribut, das zur Erstellung generiert wird und nicht geändert werden darf. UUIDs haben den Vorteil der klaren Identifikation von Elementen, sind aber vom Editor, bzw. von einem UUID-Generator, abhängig. Ein Vergleich auf Basis von UUIDs würde etwa zwei Elemente die syntaktisch und semantisch ident sind aber in unterschiedlichen Editoren erstellt wurden und deshalb unterschiedliche UUIDs haben, nicht als ident ansehen.

Zu vergleichende Artefakte liegen normalerweise in Textform vor, d.h. der Inhalt des Artefakts ist als Text dargestellt und gespeichert. Bei Quellcode sind das Zeilen von Anweisungen, bei XML Dokumenten Zeilen mit z.B. Strukturelementen oder Datenelementen. Zum Vergleichen von Quellcodeversionen oder anderen dedizierten Textinhalten ist der zeilenweise Textvergleich sehr gut geeignet. Bei strukturierten Formaten, wie z.B. XML Daten, kann der zeilenweise Textvergleich bereits zu Problemen führen, da Strukturen oft zeilenübergreifend zu bewerten sind. Sind Daten strukturiert aufgebaut, bietet sich die Möglichkeit des graphbasierten Vergleichs an. Damit wird direkt auf der „natürlichen“ Repräsentation der Datenstruktur gearbeitet. Baumbasierte Ansätze fallen ebenfalls unter diese Variante, da ein Baum als eine Spezialform eines Graphen angesehen werden kann [33]. Trotz der Vorteile beim Vergleich von strukturierten Daten sind graphbasierte Ansätze weniger oft im Einsatz als textbasierte Ansätze, die dann schlechtere Ergebnisse liefern. Der Grund liegt in der primären Anwendung der VCS für Quellcode und der dadurch viel höheren Verbreitung.

Konflikterkennung Konflikte können in mehrfacher Hinsicht klassifiziert werden. Generell können drei Operationen auf Elemente angewendet werden, um von einer Vorgängerversion ausgehend Änderungen zu erwirken. Diese grundlegenden Operationen werden als atomar bezeichnet, sie können nicht weiter unterteilt werden. Elemente können gelöscht werden (engl. delete), Elemente können hinzugefügt werden (engl. add) und Elemente können modifiziert[56] werden (engl. update). Tab. 2.2. stellt die Kombinationen jener Operationen dar, die zu Konflikten führen können, wenn sie in konkurrierender Weise auf Elemente eines Vorgängerartefakts angewendet werden. Dabei markiert das Zeichen 'X' einen Konflikt, '-' steht für kein Konflikt und 'na' bedeutet nicht anwendbar (engl. not applicable). Entsprechend dieser Aufstellung kann ein Konflikt der durch konkurrierende Updates hervorgerufen wird, als Updtate-Update-Konflikt bezeichnet werden.

Werden Konflikte auf Textbasis erkannt, also durch z.B. zeilenweisen Textvergleich, wird von textuellen Konflikten gesprochen. Sie werden rein anhand der Unterschiede der Textrepräsentation in der Form von Absätzen, Zeilen, Wörtern oder Buchstaben erkannt. Beim syntaktische Zusammenführen werden Konflikte erkannt, die auf Sprachebene auftreten, also die Syntax einer Sprache, z.B. Java, verletzen. Dieser Ansatz ist weit

[56] Im Sinne von Bearbeiten oder Verändern.

genauer und treffsicherer als reine Textvergleiche, da z.B. Kommentare und Formatierung nicht betrachtet werden müssen. Allerdings muss hier die verwendete Sprache bekannt sein und die Analyse der Syntax unterstützt werden, was den Einsatzumfang wiederum einschränkt. Ein semantischer Konflikten tritt auf, wenn Sprachvorschriften (statische Semantik) verletzt werden, z.B. Variablen nicht deklariert werden. Das Artefakt ist dann syntaktisch in Ordnung. Nur eine semantische Analyse, wie beim Kompilieren von Quellcode, findet auch derartige Konflikte. Der Aufwand steigt natürlich im Vergleich zur Erkennung von textuellen oder syntaktischen Konflikten.

	Update	Delete	Add
Update	X	X	na
Delete	X	-	na
Add	na	na	X

Tab. 2.2: Konflikte aus atomaren Operationen [17]

Die mit Tab. 2.2 beschriebenen Konflikte beruhen auf atomaren, also einzelnen Elementoperationen. Werden atomare Operationen zu einer Operation verbunden, also mehrere atomare Operationen als Einheit verstanden, dann wird diese Einheit als zusammengesetzte (engl. composite) Operation bezeichnet. Restrukturierung (engl. refactoring) oder Entwurfsmuster (engl. design patterns) sind Beispiele dafür. Die Erkennung solcher zusammengesetzter Operationen kann zu einer präziseren Konflikterkennung genutzt werden bzw. hilfreich in der Konfliktlösung sein. Da durch die Benennung einer zusammengesetzten Operation ein Sinn verknüpft ist, wird die Absicht des Benutzers erkennbar.

Es besteht ein direkter Zusammenhang zwischen der Genauigkeit, der Konflikterkennung und der generellen Anwendbarkeit. Umso genauer die Erkennung arbeitet, desto spezifischer ist der Anwendungsfokus. So funktioniert der Textvergleich beispielsweise mit jeder Art von Textartefakt, ist aber relativ grobkörnig was die Konflikterkennung betrifft. Die Erkennung von zusammengesetzten Operationen kann zu besserer Konflikterkennung führen, ist aber durch die verknüpfte Semantik nur sehr selektiv einsetzbar, z.B. nur für eine spezielle Sprache.

Konfliktlösung Bei zeilen-orientierten Artefakten, wie z.B. Quellcode, ist die manuelle Konfliktlösung durch den Benutzer üblich. Das wird bei großen Datenmengen oder auch bei Artefakten mit komplexer Struktur schnell zum Problem, einerseits wegen dem Aufwand, andererseits wegen der Fehleranfälligkeit. Automatisierung ist hier aber schwierig, weil dazu die Fähigkeit zum Fällen einer Entscheidung notwendig ist. Ein Benutzer kann entscheiden, ob eine Änderung wichtiger ist als eine andere oder ob eine Änderung Sinn macht. Ein Algorithmus kann das nicht oder nur sehr eingeschränkt. Möglichkeiten der Hilfestellung sind hier halb- oder vollautomatische Ansätze, die Regeln oder Strategien vorgeben, dass z.B. fremden Änderungen immer überschrieben werden.

Letztlich verlagern diese Ansätze aber das Fällen der Entscheidung nur in die Richtung desjenigen der die Regeln und Strategien vorab definiert.

Zusammenführung Die eigentliche Zusammenführung sollte ohne Konflikte kein Problem sein. Bei der Zwei-Weg-Zusammenführung kann das Löschen von Elementen nicht erkannt werden, hier wird die neue Version aus der Vereinigung der zwei verglichenen Versionen gebildet. Anders bei der Drei-Weg-Zusammenführung, hier kann das Löschen von Elementen erkannt werden. Die neue Version wird aus der Vereinigung aller Änderungen gebildet. Allerdings ist auch ohne Konflikte die Entscheidungsproblematik gegeben. Ein Benutzer kann die Änderungen entsprechend seiner Erfahrung und Absicht auswählen und zu einer sinnvollen Version vereinen. Ein Algorithmus zur automatischen Zusammenführung kann das nicht. Auch wenn das Ergebnis einer Validierung standhält, also nach syntaktischen und statisch-semantischen Gesichtspunkten in Ordnung ist, kann der Inhalt trotzdem praktisch sinnlos sein.

2.4.3 Modellversionierung

Nach der Besprechung der wichtigsten Grundlagen der allgemeinen Versionierung im letzten Abschnitt, wird in diesem Abschnitt der Fokus auf die Versionierung von Modellen gelenkt. In diesem Kontext ergeben sich automatisch eine Reihe von Aspekten, die eine spezielle Betrachtung und Bearbeitung notwendig machen.

Zwei Probleme haben sich als vordringlich für die Modellversionierung herausgestellt. Einerseits das Problem des sogenannten „Impedance Mismatch“ [137] und andererseits der „Representation Break“ oder Bruch in der Präsentation von Diagrammen und Differenzen [17, 33]. Beide Probleme werden im Folgenden ausführlich erläutert. Danach sollen die wichtigsten Methoden bzw. Konzepte der Modellversionierung beschrieben werden.

Modelle bestehen in aller Regel aus einer Menge von Diagrammen und/oder strukturierten Datensätzen. Die Form, die der Benutzer bei der Arbeit mit einem Modellierungswerkzeug üblicherweise sieht, entspricht einer grafischen oder textuellen konkreten Syntax. Diese Form entspricht aber nicht der Struktur mit der die Daten effektiv gespeichert werden, also der abstrakten Syntax in Textform. Modelldaten werden als Graphen gespeichert, Modelle sind also graphbasierte Artefakte. Wie im letzten Abschnitt kurz erwähnt wurde und in zahlreichen Arbeiten zu dem Thema [168, 64, 92, 23, 104, 15, 17, 34, 33, 90, 177, 178] detailliert erläutert wird, eignen sich die üblichen zur Versionierung von Quellcode verwendeten textbasierten und zeilenorientierten VCS nicht zur Versionierung von graphbasierten Artefakten. Im Vergleich zu Quelltextartefakten ist die Struktur von graphbasierten Artefakten sehr komplex und vor allem nicht zeilenorientiert. Eine Änderung in einem Textartefakt betrifft immer nur die bearbeiteten Zeilen. Einzelne Änderungen in einem Diagramm können zu mehreren Änderungen in der zugrundeliegenden Graphstruktur führen, auch an unterschiedlichen Positionen des Artefakts. Das führt bei der Versionierung mit herkömmlichen VCS zu ungenügenden Ergebnissen, da die Information, die in der Graphstruktur steckt bei der Reduktion auf eine Zeilenbasis verloren geht [34, 17]. In diesem Zusammenhang wird im Englischen von einem „Impedance Mismatch“ gesprochen

[137, 101, 112]. Einfache, zeilenweise Textvergleiche sind hier also nicht das passende Werkzeug. Oda und Saeki [139] sagen dazu, dass die Änderungen an Modellen nicht auf Zeilenebene, sondern auf Diagrammebene erfasst werden sollen, da Modelle auch anhand von Diagrammdokumenten bearbeitet werden[57].

Benötigt werden hier Systeme, die auf der graphbasierten Struktur arbeiten und daher die Syntax und Semantik der zugrundeliegenden Datenformate kennen und benutzen. Wie schon im vorigen Abschnitt kurz angemerkt, steigt mit zunehmendem Sprach- oder Domänenwissen die Präzision der Änderungs- und Konflikterkennung, die allgemeine Anwendbarkeit hingegen sinkt. Dieser Zusammenhang spiegelt sich auch in der veröffentlichten Fachliteratur zu dem Thema wieder. Auch der allgemeine Trend bei den modellgetriebenen Methoden ist erkennbar. Zu Beginn der Entwicklungen rund um die Modellversionierung standen hauptsächlich UML Klassendiagramme im Fokus des Forschungsinteresses, also Modellversionierung zugeschnitten auf genau eine Anwendung. Mit der Diversifizierung der Modellierungssprachen durch die DSL/DSML Praktiken hat sich dieser Fokus in Richtung generischer, d.h. sprachunabhängiger Modellversionierung verschoben. Es wird dabei besonderes Augenmerk auf die Fähigkeit zur Adaptierung der Systeme an Anforderungen beliebiger Sprachen gelegt, also auf die Anpassung an Sprachspezifika. Das Interesse wanderte somit von der spezifischen [143, 141] zur generischen Modellversionierung [29, 107] und weiter zur adaptierbaren, generischen Modellversionierung [35, 90].

Beim Speichern von Quellcode wird mit dem Text automatisch die Anordnung im Artefakt erfasst. Bei der Modellierung werden die Modelldaten aber von den Diagrammdaten (Anordnung der Elemente, Layout) getrennt gespeichert, zumindest bei Verwendung von OMG Standards. In Abschnitt 2.2 wurden die dafür entwickelten OMG Standards - XMI (Modelldaten) und DD (Diagrammdaten) - bereits erwähnt. Bei der automatisierten Weiterverarbeitung, z.B. Codegenerierung, hat das den Vorteil, dass die Modelldaten nicht durch Diagrammdaten, die dabei nicht benötigt werden, „verschmutzt“ sind. Bei der herkömmlichen, textbasierten Versionierung hat das den Nachteil, dass zwei Artefakte unabhängig voneinander betrachtet werden und dabei der Kontext verloren geht. Hier spielt auch die Benutzersicht, bzw. die Benutzbarkeit (engl. Usability) eine Rolle. Der Benutzer kann das Diagramm, das er grafisch erstellt hat, oft nur über eine für ihn ungewohnte Text- und/oder Baumdarstellung versionieren. Dieser Bruch in der Repräsentation führt für den Benutzer zu einer semantischen Lücke, das wiederum senkt die Benutzbarkeit und erhöht dabei die Wahrscheinlichkeit das Fehler gemacht werden [33, 17].

Der Benutzer, der Quellcode bearbeitet und versioniert, kann in allen Arbeitsschritten von einer konsistenten Sicht auf seine Artefakte ausgehen. Das Zusammenführen von Artefakten klappt im Regelfall vollautomatisch, Konfliktfreiheit vorausgesetzt. Die verfügbaren VCS sind über Jahrzehnte entwickelt worden und haben einen sehr zufriedenstellenden Stabilitätsgrad erreicht. Das kann von Werkzeugen zur Modellversionierung nicht uneingeschränkt behauptet werden. Die Modellierung benötigt ebenfalls Systeme,

[57] sinngemäß aus dem Englischen übersetzt

die dem Benutzer eine konsistente Sicht beim Bearbeiten und Versionieren der Artefakte bietet. Die darunterliegenden Abstraktionsebenen sollten für den Benutzer weitestgehend transparent gestaltet werden.

Modellversionierung unterscheidet sich in einigen gravierende Aspekten von der Textversionierung, hauptsächlich bedingt durch die zugrundeliegende graphbasierte Struktur von Modelldaten, den Repräsentationsbruch zwischen Diagrammdarstellung und Modelldaten und der Differenzierung von Modell- und Diagrammdaten. Diese resultierenden Unterschiede ziehen sich durch alle Phasen der Versionierung und prägen das Bild der Benutzererfahrung. Mit dem Fortschreiten des Forschungsstands und der einhergehenden Entwicklung verbesserter und angepasster Werkzeuge, kann mittelfristig mit einer Annäherung der Praxistauglichkeit von Modellversionierungswerkzeugen an den gewohnten Standard der Textversionierung gerechnet werden.

Im Folgenden sollen einige wichtige Konzepte und Methoden der Modelversionierung umrissen werden, mit deren Hilfe versucht wird diese Annäherung der Praxistauglichkeit zu gewährleisten. Die Modellversionierung kann, genauso wie die Textversionierung, in mehrere Phasen eingeteilt werden, siehe Abschnitt 2.4.2. Die im Kontext dieser Arbeit wichtigste Phase ist die erste, die Vergleichsphase. Modellvergleiche sind generell nicht nur für die Modellversionierung wichtig, sondern auch zum Testen von Modelltransformationen, zur Modellkomposition und zur Erkennung von Modellklonen [105, 178]. Zur Kalkulation werden entweder zwei oder drei unterschiedliche Modellversionen verglichen. Die Versionen stellen strukturelle Zustände dar, die verglichen werden. Daher werden diese Methoden als struktur- oder zustandsbasiert bezeichnet. Andere Formen sind änderungs- und opberationsbasierte Methoden, darauf wird im Anschluss eingegangen. Nach Brun und Pierantonio [38] kann diese Phase wieder in drei Phasen unterteilt werden. Das sind sinngemäß Kalkulation, Repräsentation und Visualisierung. Die Kalkulationsphase besteht wieder aus zwei Phasen, Elementabgleich (matching) und Differenzierung und kann auf Basis der abstrakten Syntax, der konkreten Syntax oder der Semantik erfolgen. Hauptsächlich werden abstrakte und konkrete Syntax herangezogen, damit lassen sich Modell-, sowie Diagrammänderungen erkennen. Die nachfolgenden Erläuterungen beziehen sich auf Vergleiche der Syntax. Auf semantische Methoden wird im Anschluss kurz eingegangen. Nach Stephen und Cordy [178] können zum Elementabgleich entweder statische Identifikatoren, Signaturen, Ähnlichkeitsheuristiken oder benutzerdefinierte, sprachspezifische Methoden herangezogen werden.

Statische Identifikatoren, auch als Element-IDs oder UUIDs bekannt, werden üblicherweise vom Editor oder einer Persistenzschicht vergeben. Vorteil von IDs sind die hohe Effizienz und Genauigkeit. Allerdings können IDs nur in sehr homogenen Umgebungen genutzt werden, da z.B. beim Einsatz von unterschiedlichen Modelleditoren nicht von einer einheitlichen Vergabe von IDs ausgegangen werden kann. Signaturen sind eindeutige Zusammensetzungen aus Merkmalen eines Elements, z.B. der absolute Speicherpfad einer Datei. Ähnlichkeitsheuristiken vergleichen meist verschiedene Merkmale von Elementen, etwa den Namen, die Anzahl der Eigenschaften oder Methoden einer Klasse, Methodensignaturen oder die Anzahl und Struktur von Relationen und berechnen daraus einen Wert

für die Ähnlichkeit. Liegt das Ergebnis über einem Grenzwert, werden die verglichenen Elemente als gleichwertig angesehen. Der Vorteil von Signaturen und Ähnlichkeitsheuristiken sind die Unabhängigkeit von spezifischen Modellierungswerkzeugen und Sprachen, der Nachteil sind die im Vergleich zur Verwendung von IDs niedrigere Genauigkeit und Effizienz.

Nach dem Elementabgleich können die erstellten Korrespondenzen verglichen und so die Differenz zwischen zwei Versionen ermittelt werden. Eigentlicher Sinn und Ziel ist der Rückschluss auf durchgeführte atomare Änderungen bzw. Benutzeraktionen. Die erhobenen Differenzen werden in weiteren Phasen der Versionierung als Basis für z.B. die Konflikterkennung benötigt. Dieses Vorgehen ist relativ aufwändig. Der Elementabgleich kann als das Problem der Graphenisomorphie aufgefasst werden, das als NP-hart gilt [104, 33], was wiederum hohe Rechenintensität bedeutet.

Die Informationen aus der Vergleichsphase werden in der nächsten Phase, der Repräsentation, in geeigneter Form für die Weiterverarbeitung aufbereitet. Aus den erhobenen Operationen können z.B. Skripte oder Patches erstellt werden, die eine Version in die andere Version verwandeln können [130]. Die mittlerweile bevorzugte Methode ist allerdings, die Informationen in einem Differenzmodell zu erfassen, ganz nach dem MDE-Motto „Alles ist ein Modell“ [41]. Damit bleibt für die Weiterverarbeitung die ganze Expressivität der Differenzen erhalten. Der Visualisierungsphase fällt die Aufgabe zu, die Differenzen möglichst passend, d.h. benutzergerecht anzuzeigen. Durch Verwendung eines Differenzmodells kann hier z.B. einfach auf eine Diagrammdarstellung gesetzt werden, was den oben erkläreten Repräsentationsbruch aufhebt. Diese Methode ist mittlerweile ebenfalls state-of-the-art.

Die obige Einteilung in Phasen trifft auf syntaktische bzw. zustandsbasierte Vergleichsmethoden zu. Die anderen Möglichkeiten sind änderungsbasierte oder operationsbasierte Vergleichsmethoden. Bei änderungsbasierten Methoden wird zusätzlich ein Protokoll der durchgeführten Änderungen benutzt, operationsbasierte Methoden sind eine Spezialform davon [17]. Operationsbasierte Methoden sind nur durch tiefe Integration in die Arbeitsumgebung realisierbar. Sie zeichnen alle Änderungen und sonstigen interessanten Informationen auf und analysieren diese in Anschluss. Dadurch ist es einfach möglich auch zusammengesetzte Änderungen, etwa Refactorings [60], zu unterstützen. Diese Methoden können um einiges genauer und effizienter arbeiten als zustandsbasierte Methoden, jedoch sind die Methoden durch die starre Bindung an die Arbeitsumgebung äußert unflexibel [33].

Eine Methode die sich von den bisher beschriebenen grundlegend unterschiedet ist semantischer Natur. Dabei wird versucht auf semantische Informationen, die in der Modellstruktur gebunden sind, zuzugreifen und diese für die Änderungs- oder Konflikterkennung auszunutzen. Die Expressivität gewisser semantischer Aspekte wird z.B. über das Erstellen von Instanzen(Objekten) bei Klassendiagrammen oder Ablaufmustern bei Aktivitätsdiagrammen verstärkt [125, 111]. Ein weiterer Ansatz beschreibt die Bildung von semantischen Ansichten auf Modelle [14]. Diese Methoden sind im Vergleich zu

den oben beschriebenen Methoden noch eher jung, lassen aber bereits auf weiteren Erkenntniszuwachs und interessante Entwicklungen hoffen.

Mit konkreten Umsetzungen der hier kurz angerissenen Methoden und Konzepte beschäftigt sich das Kapitel 3, Stand der Wissenschaft. Dort werden die wichtigsten Ansätze bzw. Systeme der Domäne vorgestellt und mit einem Fokus auf charakterisierende Merkmale und Funktionen besprochen. Der nächste Abschnitt stellt die Softwareprodukte vor, die im Zuge der Umsetzung dieser Arbeit verwendet wurden.

2.5 Softwarestack

In diesem Abschnitt werden einige wichtige Aspekte der Softwareprodukte vorgestellt, die für diese Arbeit verwendet wurden. Java bildet, bis auf eine Ausnahme, nämlich Git, die Basis des gesamten verwendeten Softwarestacks.

2.5.1 Java

Die Sprachspezifikation der Standardedition (Java SE)[58] beschreibt Java als eine universell einsetzbare, klassenbasierte, objektorientierte Programmiersprache. Java ist zum Erstellen von parallelen bzw. nebenläufigen Programmen geeignet und ist streng und statisch typisiert. Java ist eine relativ hoch abstrahierte Sprache, Spezifika der Rechner- oder Maschinenarchitektur sind in der Sprache nicht abgebildet, so wird etwa der Speicher automatische verwaltet und regelmäßig eine Speicherbereinigung durchgeführt. Der Java Quellcode wird zu Bytecode kompiliert, dieser wird dann auf einer Virtuellen Maschine, der Java Virtual Machine, ausgeführt. Java ist daher eine interpretierte, plattformunabhängige Sprache und gleichzeitig eine Anwendungsplattform.

Die Java SE wird als reine Laufzeitumgebung Java Runtime Environment (JRE) und als Entwicklungspaket Java Development Kit (JDK) angeboten. Im JRE sind alle Komponenten enthalten um Java Programme zu starten, der JDK enthält die JRE und überdies noch Entwicklungskomponenten, wie den Java Compiler. Weitere Editionen sind z.B. die Enterprise Edition (Java EE), die Micro Edition (Java ME) oder Java Card. Daneben bietet die Java Platform seit Version 6[59] die Möglichkeit, Skriptsprachen zu interpretieren, z.B. JavaScript oder PHP. Seit Version 7[60] können eigene Programmiersprachen für die JVM entwickelt werden, z.B. Ceylon. Damit ist die JVM zu einer Plattform für DSLs geworden und die Einsatzmöglichkeiten der Java Technologien wurde weiter ausgebaut und gestärkt.

[58] `https://docs.oracle.com/javase/specs`
[59] JSR 223 (Scripting for the Java Platform)
[60] JSR 292 (Supporting Dynamically Typed Languages on the Java Platform)

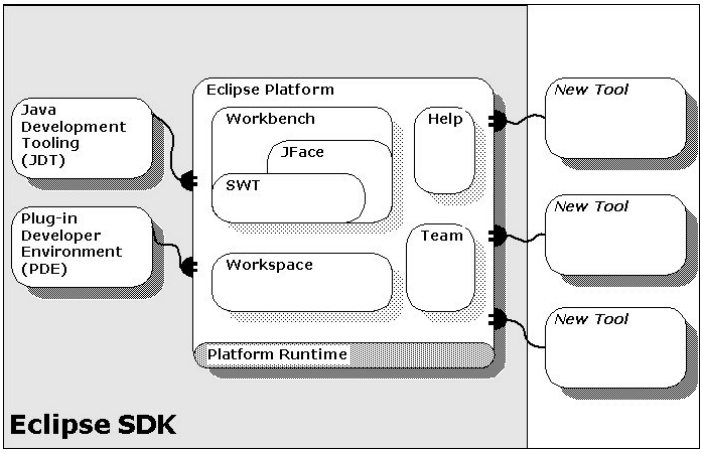


Abb. 2.22: Eclipse Platform mit Plugins [5]

2.5.2 Eclipse

Wird über Eclipse gesprochen ist eigentlich meist eine Eclipse IDE[61] (Integrated Development Environment) gemeint. Eclipse selbst bezeichnet das Open Source Projekt einer Gemeinschaft aus Unternehmen und Einzelpersonen, die sich um die Entwicklung der zahlreichen Eclipse Teilprojekte kümmert. Das zentrale Produkt des Eclipse Projekts, dessen ursprünglicher Zweck die Entwicklung einer Java IDE war, stellt mittlerweile eine Anwendungsplattform dar, die Eclipse Platform. Sie besteht aus mehreren Basiskomponenten die grundlegende Funktionalität zu Verfügung stellen, siehe Abb. 2.22.

Die Basis der Eclipse Platform bildet eine schlanke Laufzeitkomponente, die Eclipse Platform Runtime, die wiederum auf dem OSGi[62] Framework Equinox basiert. Sämtliche Funktionalität wird darauf aufbauend über Plugins realisiert. Die Eclipse Platform kann nicht nur als Basis für IDEs unterschiedlicher Programmiersprachen dienen, sondern generell als Basis für Anwendungen aller Art. Die Eclipse Platform Runtime ergänzt um einige Basisplugins, bilden gemeinsam die Eclipe Rich Client Platform (RCP). Die aktuelle Version der Plattform ist 4.7.1 (Oxygen).

2.5.3 Das Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) erweitert Eclipse um Modellierungsaspekte und Funktionalitäten modellgetriebener Konzepte. Es ist der Kern des Eclipse Modeling Projects[63] und die Basis vieler Plugins im Modellierungskontext, es stellt quasi einen Modellierungsstandard für Eclipse dar. Auf der Eclipse Webseite wird das EMF beschrieben als „[...] Modellierungsframework und Codegenerator, um Werkzeuge und andere Anwendungen, basierend auf strukturierten Datenmodellen, zu erstellen“. Trotz des Stel-

[61] Zu Deutsch „integrierte Entwicklungsumgebung“
[62] Open Services Gateway initiative, `https://www.osgi.org`
[63] `https://eclipse.org/modeling`

lenwerts den das EMF einnimmt, gibt es leider keine umfassende und frei verfügbare Dokumentation. Es wird häufig auf das EMF Standardwerk [176] verwiesen. Die aktuelle Version des EMF ist 2.12.0.

Das EMF besteht aus 3 Teilen [4]:

- EMF(Core) bildet mit der Modellierungssprache Ecore und der Laufzeitunterstützung für Modelle den Kern des Frameworks.
- EMF.Edit bietet generische, wiederverwendbare Komponenten zum Erstellen von Editoren.
- EMF.Codegen ist die Komponente mit deren Hilfe aus Modellen Java Code generiert werden kann.

Laut Ed Merks, Projektleiter des Eclipse Modeling Projects und des EMF, ist Ecore „eine de-facto Referenzimplementation von EMOF“ [12], siehe Abschnitt 2.2.2. Den Konzepten der Metamodellierung folgend, siehe Abschnitt 2.1.4), findet sich Ecore auf Ebene M3 und stellt somit eine Metasprache oder ein Metametamodell dar. Wie die MOF ist auch Ecore selbstdefinierend. Mit Ecore können Metamodelle erzeugt werden, aus denen anschließend Java Code generiert werden kann. Die Laufzeitunterstützung für Modelle erweitert Modellklassen um automatische Änderungsmeldung, z.B. für grafische Benutzeroberflächen (GUIs[64]), ermöglicht generischen Zugriff über eine effiziente reflektive Anwendungsschnittstelle (API[65]) und bietet Unterstützung zur Speicherung, z.B. im Format XMI.

Das EMF.Edit Framework bietet Komponenten an, die zum einfachen Aufbau von Editoren benutzt werden können. Es enthält wiederverwendbare Elemente zum generischen Zugriff auf Modellobjekte und ermöglicht das Zusammenspiel mit Standardkomponenten der Eclipse Benutzeroberfläche.

EMF.Codegen enthält Komponenten zur automatischen Codegenerierung. Angeboten werden drei aufeinander aufbauende Funktionalitätsstufen. Ausgehend von reinen Modellklassen, über Adapter zum Anzeigen und Editieren für die GUI Integration, bis zu kompletten Editoren, kann alles aus einem Modell erzeugt werden. Der Aufbau der Komponenten ermöglicht wiederholte Codegenerierung, ohne Verlust von eigenen Änderungen oder Erweiterungen.

2.5.4 EMF Compare

EMF Compare[66] ist ein Subprojekt des EMF Projekts. Das Plugins dient dem Vergleichen und Zusammenführen (mergen) von EMF Modellen. EMF Compare ist modular aufgebaut,

[64] engl. Graphical User Interface
[65] engl. Application Programming Interface
[66] `http://www.eclipse.org/emf/compare`

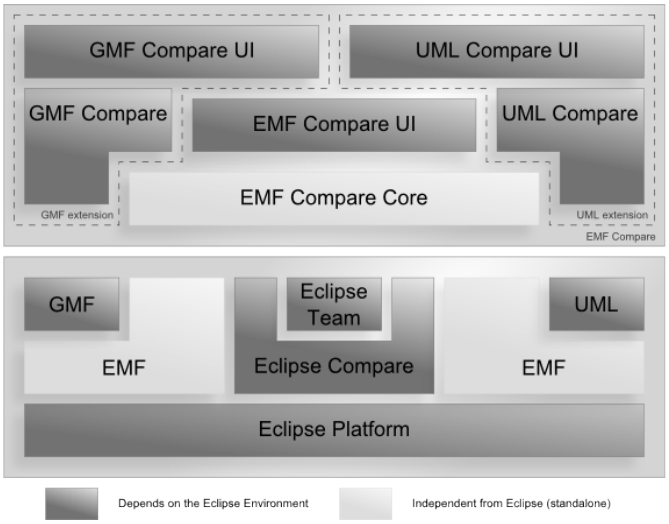


Abb. 2.23: Architektur von EMF Compare [6]

so kann es auf zwei Arten verwendet werden. Die Kernfunktionalität wurde, wie bei EMF, in purem Java umgesetzt, ohne Abhängigkeiten zu Eclipse. Dadurch kann es auch eigenständig verwendet werden, dann aber ohne die grafische Benutzeroberfläche, siehe Abb. 2.23. Der Rest der Komponenten setzt auf dem Kern auf und bietet das bekannte Eclipse Look and Feel. Durch die Eclipse Team API Anbindung, ermöglicht EMF Compare kooperatives Arbeiten im Eclipse Kontext, natürlich nur in Kooperation mit einem Versionskontrollsystem, z.B. CVS, SVN oder Git und einem entsprechenden Plugin. Momentan ist Version 3.3 aktuell.

EMF Compare wurde erstmals 2006 auf dem Eclipse Summit Europe vorgestellt [187]. Damals noch mit eingeschränkter Funktionalität und verbesserungswürdiger Performance aber einem interessanten Konzept zum generischen Elementvergleich. Es wurden nicht mehr IDs zur Paarbildung herangezogen, sondern, basierend auf der Informationstheorie von Shannon, ein Wert berechnet, der die Ähnlichkeit zweier Elemente repräsentiert. Zwei Jahre später, 2008, wurde eine komplett überarbeitete Version von EMF Compare vorgestellt [38]. Die neue Version beruhte auf einem modellbasierten Ansatz, der Elementvergleich wurde verbessert und der Vergleichsprozess in zwei Phasen aufgeteilt. Eine Phase zum Elementabgleich (matching) und eine Phase zur Differenzermittlung (differencing). Wurden zuvor Objektlisten zum Speichern der Vergleichsergebnisse verwendet, so setzte die Neuimplementierung auf ein Modell für Elementpaare und ein Modell für die gefundenen Differenzen. Der neue Elementvergleich verwendete auch Statistiken, Heuristiken und Metriken, um zu besseren Vergleichsergebnissen zu gelangen. Der modellbasierte Ansatz hat den Vorteil, dass die Vergleichsergebnisse in Modellform vorliegen und somit universell weiterverarbeitbar sind. So können z.B. einfach Änderungen an den Ergebnissen vorgenommen werden oder die Ergebnisse grafisch am Bildschirm angezeigt werden. Der in Phasen unterteilte Vergleichsprozess bietet den Vorteil der Modularisierung, einzelne

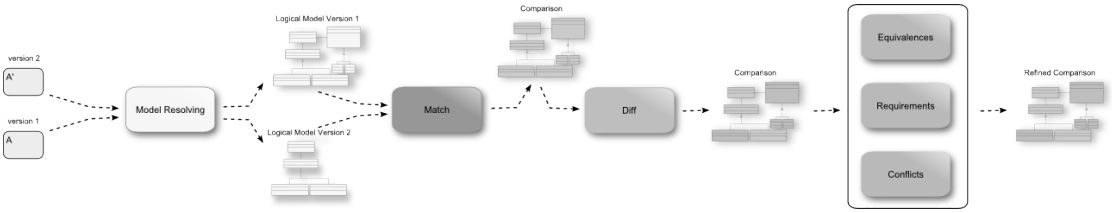


Abb. 2.24: Phasen des EMF Compare Vergleichsprozess [6]

Teile können damit einfach ausgetauscht, angepasst oder erweitert werden. Aufbauend auf diesen Ansätzen wurde EMF Compare seither weiterentwickelt. Aus den ehemals zwei Modellen für Elementpaare und Differenzen wurde mittlerweile ein großes, alle Informationen des gesamten Vergleichsprozesses beinhaltendes Vergleichsmodell, siehe Anhang A.

Auch der zweiphasige Ansatz des Vergleichsprozess wurde zwischenzeitlich weiterentwickelt. Der Vergleichsprozess ist jetzt in sechs Phasen eingeteilt. In jeder dieser Phasen wird mit dem Vergleichsmodell gearbeitet, es wird schrittweise erweitert bzw. verfeinert, siehe Abb. 2.24.

Im EMF Compare Entwicklerleitfaden [6] werden die sechs Phasen des Vergleichsprozess wie folgt beschrieben:

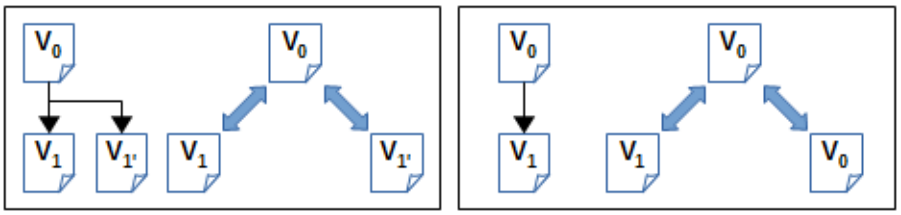
- Modellauflösung (Model Resolving): Alle Teile oder Fragmente der zum Vergleich ausgewählten Modelle werden zu einem logischen Modell vereint.

- Modellabgleich (Matching): Es werden Modellelemente gesucht, die zusammenpassen. Beispielsweise Klasse A aus Modell1 und die leicht veränderte Klasse A' aus Modell2. Dabei werden Zweier- oder Dreierpaare gebildet, je nach dem, ob es sich um einen 2-Weg oder 3-Weg Vergleich handelt.

- Modelldifferenzierung (Diff): Die Zweier- oder Dreierpaare aus der vorigen Phase werden verglichen und Differenzen ermittelt, z.B. der geänderte Name der Klasse A' aus Modell2.

- Äquivalenzen (Equivalences): In der vorigen Phase wurden Differenzen zwischen den Elementpaaren gesucht, jedoch können zwei Differenzen die selbe Änderung repräsentieren, z.B. Änderungen an gegenüberliegenden Referenzen. In dieser Phase werden alle Änderungen verbunden, die als äquivalent angesehen werden können.

- Anforderungen (Requirements): Beim Zusammenführen von Änderungen können Abhängigkeiten auftreten, z.B. kann das Hinzufügen von Attribut X zur Klasse Y erst erfolgen, wenn Klasse Y existiert. In dieser Phase werden alle Änderungen entsprechend ihren Voraussetzungen verbunden.



(a) normal (b) mit nur 2 Versionen

Abb. 2.25: 3-Wege-Vergleich

- Konflikte (Conflicts): Beim Vergleich mit Modellen aus einem Versionskontrollsystem können Konflikte zwischen lokalen und versionierten Modellen auftreten und zwar dann, wenn Änderungen aus der Versionsverwaltung noch nicht lokal übernommen wurden. In dieser Phase werden alle Änderungen auf derartige Konflikte überprüft.

EMF Compare ist durch diese Architektur umfassend erweiterbar, es bietet für jede der Prozessphasen ExtensionPoints an, siehe das Ergebnis der Erweiterungsanalyse weiter unten. Dadurch kann in jede der sechs Phasen eingegriffen werden und das Vergleichsmodell geändert werden. Gründe für das Eingreifen sind z.B. die Probleme die in Abschnitt 2.4.3 erläutert werden.

EMF Compare ist sprachunabhängig und unterstützt 2-Weg und 3-Weg Vergleiche. In der Phase zum Modellabgleich - oder eigentlich Elementabgleich - verwendet EMF Compare, je nach Konfiguration, Element-IDs und/oder Ähnlichkeitsheuristiken. EMF Compare erkennt ausschließlich folgende, atomare Änderungen: Hinzufügen (add), Entfernen (delete), Modifizieren (change) und Verschieben (move). Zusammengesetzte Änderungen werden nicht unterstützt.

Ein echter 3-Wege Vergleich ist nur durchführbar, wenn zu einer Anfangs- oder Ursprungsversion zwei nachfolgende Versionszweige existieren, siehe Abb. 2.25a. Wird ein Vergleich zwischen aufeinanderfolgenden Versionen durchgeführt, nicht zwischen Verzweigungen, ist keine Ursprungsversion verfügbar. In diesem Fall legt EMF Compare automatisch die ältere Version als Ursprungsversion fest. Dadurch sind auch bei linearem Versionsverlauf, also ohne Versionszweige, 3-Wege Vergleiche möglich, siehe Abb. 2.25b.

Analyse der EMF-Compare Erweiterungsmöglichkeiten

Dieser Abschnitt stellt die Ergebnisse der zweiten Analysephase vor. Aus Strukturgründen wurde der Abschnitt hier eingefügt.

Das eigentliche EMF Compare Paket[67] bietet selbst keine Erweiterungsmöglichkeiten an. Es stellt allerdings mit dem Vergleichsmodell den konzeptuellen und mit der funktionalen

[67] `org.eclipse.emf.compare`

Basis den operativen Kern des Vergleichsprozesses dar. EMF Compare ist zwar ein Eclipse Plugin. Da es aber keine Abhängigkeiten zur Eclipse Platform besitzt, kann es auch eigenständig verwendet werden. Ganz ohne Abhängigkeiten kommt EMF Compare jedoch nicht aus, benötigt werden die Pakete `Ecore`[68] und `XMI`[69] aus dem EMF Projekt. Die Möglichkeiten, um EMF Compare zu erweitern bzw. anzupassen, werden von den diversen nativen Erweiterungen über den Eclipse Plugin Mechanismus der `ExtensionPoints` realisiert. Momentan gibt es insgesamt 22 `ExtensionPoints` für EMF Compare, verteilt über 5 Plugins. Die verfügbaren `ExtensionPoints` werden hauptsächlich von den EMF Compare Paketen `RCP`[70] und `RCP.UI`[71] angeboten. Es stehen somit zwei Vorgehensweisen zur Erweiterung frei. Erstens das Erweitern durch direkte Bearbeitung eines Pakets und zweitens das Erstellen eines Plugins unter Verwendung eines oder mehrerer `ExtensionPoints`.

Da die oben beschriebenen Probleme (siehe Abschnitt 2.4.3) alle mit Diagrammen zu tun haben, war als Einstiegspunkt die Diagrammerweiterung[72] zu EMF Compare naheliegend. Dieses Plugin erweitert EMF Compare um das Konzept von Diagrammdifferenzen und entsprechen dem Aufbau von GMF Diagrammen um die Konzepte von Diagrammbausteinen. Die Diagrammerweiterung besteht streng genommen aus zwei Paketen, dem eben erwähnten, namensgebenden Hauptpaket und dem `edit` Paket[73]. In diesem Paket finden sich die notwendigen Erweiterungen um die Diagrammerweiterung mit der Eclipse UI zu verbinden [176], konkret sind das Klassen die zur korrekten Anzeige der Diagrammdifferenzen in der Vergleichsansicht benötigt werden. Die Diagrammerweiterung ist ein EMF Projekt, basiert also grundlegend auf einem `ecore` Modell. Das Modell[74] definiert Spezialisierungen der abstrakten Diff-Klasse des EMF Compare Vergleichsmodells. Diese sind notwendig, um die benötigten Konzepte im EMF Compare Kontext bekannt zu machen. Teile des Quellcodes eines EMF Projekts können anhand des Modells generiert werden, z.B. die notwendigen Modellklassen der Erweiterung sowie zugehörige `edit`, `editor` und `test` Projekte. Mit der Generierung eines `editor` Projekts wird ein voll funktionsfähiger Spezialeditor für das gegebene Modell erzeugt. Das `test` Projekt enthält generierte `jUnit` Tests für die Modellklassen. Das Modell der Diagrammerweiterung benötigt zur Quellcodegenerierung eine spezielle Eclipse Erweiterung für EMF[75]. Details dazu werden in der Beschreibung der Implementierungsphase im Abschnitt 5.2 erläutert.

Das Hauptpaket der Diagrammerweiterung benutzt drei `ExtensionPoints`. Der erste `ExtensionPoint`[76] dient der Registrierung von generierten EMF Packages, ist also kein EMF Compare `ExtensionPoint`. Erweiterungen dieser Art werden beim Generieren

[68] `org.eclipse.emf.ecore`
[69] `org.eclipse.emf.ecore.xmi`
[70] `org.eclipse.emf.compare.rcp`, RCP ist die Ankürzung für Rich Client Platform
[71] `org.eclipse.emf.compare.rcp.ui`, UI ist die Abkürzung für User Interface
[72] `org.eclipse.emf.compare.diagram`
[73] `org.eclipse.emf.compare.diagram.edit`
[74] `diagramCompare.ecore` im Plugin `org.eclipse.emf.compare.diagram`
[75] EMF Loophole for EMF 2.9, `http://mbarbero.github.io/emf-loophole/2.9`
[76] `org.eclipse.emf.ecore.generated_package`

von EMF Code automatisch angelegt. Der zweite `ExtensionPoint`[77] ist gleichzeitig der für die benötigten Zwecke interessanteste, da hier die Hauptfunktionalität der Erweiterung anknüpft. Der dritte `ExtensionPoint`[78] bietet die Möglichkeit die Zusammenführung von Modellen zu beeinflussen. Das `edit` Paket der Diagrammerweiterung benutzt zwei `ExtensionPoints`. Der erste[79] dient der Verknüpfung mit dem `edit` Paket von EMF, der zweite[80] der Verknüpfung mit dem `edit` Paket von EMF Compare.

Nachdem die beiden Pakete der Diagrammerweiterung alle notwendigen Teile einer Erweiterung für SysML Diagramme boten, wurde diese als direkte Vorlage zur Ausarbeitung von Lösungen herangezogen.

2.5.5 Papyrus

Zusammengefasst steht auf der Eclipse Produkt-[81], Projekt-[82] und Wiki-Webseite[83], dass Papyrus ein Open Source MBE Werkzeug mit Industriequalität zum Erstellen von EMF Modellen jeder Art darstellt. Papyrus unterstützt vor allem UML2 und verwandte Modellierungssprachen, z.B. SysML oder MARTE[84]. Weiters wird eine fortgeschrittene Unterstützung für UML-Profile geboten und die Papyrus Benutzeroberfläche kann umfangreich angepasst werden. Mit der Kombination aus Anpassungsmöglichkeiten und Erweiterungen über UML-Profile, können vollständige DSL Editoren realisiert werden. Papyrus ist ein Subprojekt des Model Development Tools (MDT) Projekts, das wiederum zum Eclipse Modeling Projekt gehört. MDT ist dabei kein eigenständiges Produkt oder Plugin, sondern ein sogenanntes Container Projekt und beherbergt Werkzeuge und Komponenten, wie eben Papyrus oder die UML2 und OCL Erweiterungen für Eclipse. Die aktuelle Version von Papyrus ist 2.0.

2.5.6 Git und EGit

Git[85] ist eine Open Source Software zur dezentralen Versionsverwaltung (DVCS), siehe Abschnitt 2.4.2. Einige Merkmale lassen Git aus der Menge der VCS hervortreten, dazu zählen die Datenhaltung, der Branching Mechanismus und der Index. In der Datenhaltung ähnelt Git eher einem Dateisystem, als einem typischen VCS. Es speichert jeweils Schnappschüsse von Zuständen, nicht Dateidifferenzen wie andere Systeme [47]. Das Abzweigen vom Entwicklungsstrang (Branching) und das spätere Zusammenführen (Merging) sind in Git sehr einfach gestaltet und gehören prinzipiell zur Arbeitsweise mit Git. Es wird dadurch auf einfache Art möglich, eine Vielzahl von parallelen Zweigen

[77] `org.eclipse.emf.compare.rcp.postProcessor`
[78] `org.eclipse.emf.compare.rcp.merger`
[79] `org.eclipse.emf.edit.itemProviderAdapterFactory`
[80] `org.eclipse.emf.compare.edit.adapterFactory`
[81] `https://eclipse.org/papyrus`
[82] `https://projects.eclipse.org/projects/modeling.mdt.papyrus`
[83] `https://wiki.eclipse.org/Papyrus`
[84] Modeling and Analysis of Real-time and Embedded systems, `http://www.omg.org/omgmarte`
[85] `http://git-scm.com`

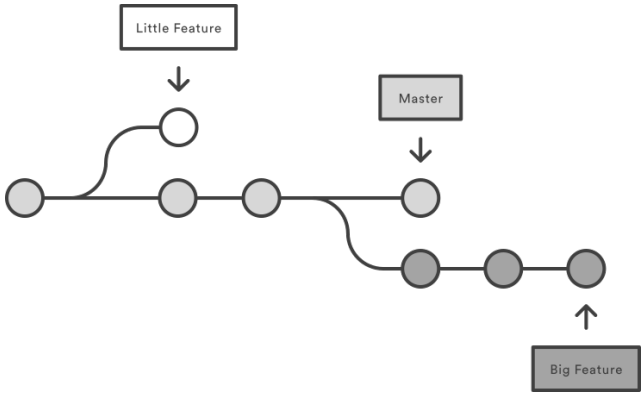


Abb. 2.26: Git Branches [7]

zu verwalten, um z.B. auf der Quelltextebene eine mehrstufige Kontrollstruktur zur Softwareentwicklung einzuführen, siehe Abb. 2.26.

Der Index, oder auch Staging Area, stellt eine Kontrollebene zwischen Arbeitskopie und Repository dar, siehe Abb 2.27. Es ist damit möglich, Änderungen vor der Übertragung ins Repository zu strukturieren, nicht alle Änderungen an einer Datei müssen auf einmal übertragen werden.

EGit[86] ist ein Plugin zur Einbindung der Git Funktionalität in die Benutzeroberfläche der Eclipse Plattform und basiert auf der Java Implementierung von Git, Jgit[87]. Genauer gesagt ist EGit ein Team Provider. Der Eclipse Workspace Komponente für kooperatives Arbeiten, Team, wird durch EGit das Zusammenspiel mit Git ermöglicht. EGit ist mittlerweile in fast allen Eclipse Paketen standardmäßig enthalten.

2.5.7 Die Initiative „Collaborative Modeling with Eclipse“

Ziel der Initiative „Collaborative Modeling with Eclipse“[88], ist die Bereitstellung einer kompletten und hochqualitativen, grafischen Modellierungsumgebung für EMF Modelle und Papyrus Modelle, zur effizienten Zusammenarbeit im Team, bestehend aus Open Source Software aus dem Eclipse Umfeld [102]. Zu diesem Zweck werden folgende Hauptkomponenten in einem Paket vereint:

- Eclipse Plattform: Die Eclipse Platform bildet die Basis für die Modellierungsumgebung, siehe Abschnitt 2.5.2.
- Papyrus: Wie bereits vorgestellt, siehe Abschnitt 2.5.5, stellt Papyrus eine grafische Modellierungsumgebung bereit.

[86] http://www.eclipse.org/egit
[87] http://www.eclipse.org/jgit
[88] http://www.collaborative-modeling.org

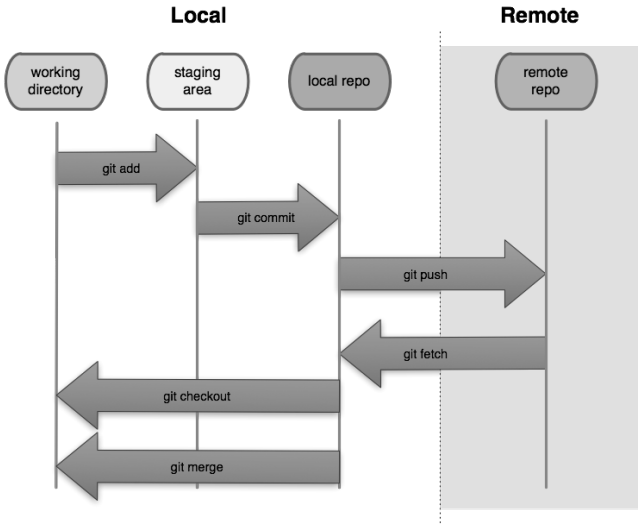


Abb. 2.27: Git Workflow [11]

- EGit: Zur Unterstützung von logischen Modellen wird eine speziell angepasste Version von Egit verwendet, siehe Abschnitt 2.5.6.
- EMF Compare: Um logische Modelle für EGit zu unterstützen, muss auch EMF Compare in einer speziellen Version vorliegen, siehe Abschnitt 2.5.4.

Das Paket steht auf der Webseite der Initiative zum Download bereit. Die aktuelle Version basiert noch auf Eclipse 4.6 (Neon).

KAPITEL 3

Stand der Wissenschaft

Im letzten Abschnitt wurden die Probleme und Herausforderungen der Modellversionierung besprochen. Warum herkömmliche VCS, deren primäre Aufgabe die Versionierung von Quellcode darstellt, zur Versionierung von Modellen nicht unbedingt geeignet sind, kann in so gut wie jeder der im Folgenden beschriebenen Arbeiten gelesen werden und wurde ebenfalls im letzten Abschnitt ausführlich dargelegt. Dieser Abschnitt ist nun den verschiedenen Ideen und Lösungsansätze gewidmet, die zur Behebung der Einschränkungen herkömmlicher Textversionierungssysteme entwickelt wurden. Im Folgenden werden bestehende Ansätze und Systeme zur Modellversionierung oder Teilaspekten davon vorgestellt.

In den letzten rund 15 Jahren sind eine Vielzahl von Publikationen veröffentlicht worden, die sich dem Thema der Modellversionierung, Teilaspekten davon oder generell dem Thema nahestehenden Betrachtungen widmen. In einer ganzen Reihe von Untersuchungen werden diese verschiedenen Ansätze und Systeme zur Modellversionierung verglichen und diskutiert, sowie zum Teil auch Empfehlungen und Anforderungen für weitere Entwicklungen aufgezeigt [168, 64, 92, 23, 104, 15, 17, 34, 33, 90, 177, 178]. Die beiden Arbeiten von Stephen und Cordy [177, 178] stellen - nach derzeitigem Stand - die jüngsten und umfangreichsten Beiträge dar.

Die Vielzahl der Arbeiten zu dem Thema Modellversionierung[1] kann grundsätzlich in zwei Gruppen unterschieden werden. Erstens die Arbeiten, die vollständige Modellversionierungssysteme vorstellen oder beschreiben, und zweitens Arbeiten, die aus Lösungsansätzen zu Teilbereichen der Modellversionierung bestehen, wie z.B. Differenzierung, Konflikterkennung oder auch semantische Aspekte. Der Fokus dieser Arbeit liegt, bedingt durch die Aufgabenstellung, klar auf den Aspekten des Modellvergleichs. Es werden hier daher einerseits Arbeiten vorgestellt die Modellversionierungssysteme behandeln, sowie

[1] einen guten Überblick bietet die „Bibliography on Comparison and Versioning of Software Models“ der Universität Siegen [189]

andererseits Arbeiten, die sich mit Elementabgleich und Differenzierung von Modellen beschäftigen. Arbeiten die andere Teilaspekte zum Inhalt haben, werden deshalb nicht eingehend besprochen.

Viele der vorgestellten Systeme sind sich relativ ähnlich und unterscheiden sich oft nur in Details. Deshalb wird hier versucht, anhand der verfügbaren Informationen, technische und methodische Charakteristiken als Unterscheidungsmerkmale herauszuarbeiten. Das sind z.B. die zugrundeliegende Modellierungstechnologie, unterstützte Modellierungssprachen, die Operationsermittelung oder die verwendete Vergleichsmethode. Die vorgestellten Arbeiten sind anhand der Sprachunterstützung in Abschnitte unterteilt. Innerhalb der Abschnitte ist die Reihenfolge chronologisch angelegt. Damit wird die in Abschnitt 2.4.3 beschriebene, schrittweise Entwicklung von sprachspezifischen Systemen über generische Systeme zu schlussendlich generischen und adaptierbaren Systemen noch einmal hervorgehoben und verdeutlicht. Die Beschreibung des in dieser Arbeit verwendeten Systems EMF Compare wird hier nicht wiederholt, siehe dazu Abschnitt 2.5.4.

Zwei Arbeiten, deren Einfluss im Bereich der Modellversionierung noch immer spürbar ist, datieren knapp nach der letzten Jahrtausendwende. Systeme zur Softwareversionierung waren bereits weit verbreitet. Die UML war in etwa zehn Jahre alt und modellbasierte Methoden wurden langsam populär. Dabei wurde ein Problem spürbar, das in der Softwareentwicklung, zumindest für Quellcode, eigentlich als praktisch gelöst angesehen wurde. Die Versionierung von Modellen und Diagrammen stellte eine nicht zu unterschätzende Herausforderung dar. Die Arbeit von Mens [130], obwohl eigentlich gar nicht auf Modellversionierung, sondern auf Software Merging fokusiert, also auf das Zusammenführen von Softwareversionen, behält bis heute ihre Aktualität. Beinah jeder von Mens besprochene Aspekt kann auf die Versionierung von Modellen umgelegt oder angewandt werden. Die zweite in diesem Kontext unbedingt zu erwähnende Arbeit kommt von Alanen und Porres [13] und wird im Abschnitt über generische Ansätze vorgestellt.

3.1 Sprachspezifische Ansätze

Die in diesem Abschnitt vorgestellten Systeme bzw. Ansätze sind für den Einsatz mit einer bestimmten Modellierungssprache vorgesehen. Die Spezialisierung auf eine Sprache hat den Vorteil, ein System genau auf diese Sprache abstimmen zu können, also z.B. auf Eigenheiten der Sprachen, spezielle Konzepte etc. Als Nachteil ergibt sich die Inflexibilität eben mit nur einer Sprache arbeiten zu können. Das ist vor allem im Hinblick auf die sich rasant entwickelnden Verbreitung von DSLs ein Nachteil.

Ohst et al. Einer der ersten auf UML Modelle zugeschnittene Ansatz kommt von Ohst et al. [142]. Das unbenannte System führt 2-Wege Vergleiche durch und benutzt IDs zur Elementidentifikation. Der Elementvergleich wird nach Art einer Tiefensuche durchgeführt. Das System setzt eine feingranulierte Dokumentstruktur voraus, das bedeutet, dass jedes Element eines Modells eine eigene Objektinstanz darstellt, z.B. Methoden und Attribute einer Klasse. Es werden nur Modelldaten betrachtet. Diagrammdaten, die

das Layout betreffen, werden verworfen. Die Ergebnisse werden in einem vereinigten Dokument[2] gespeichert, das gleichzeitig zur Visualisierung dient. Das vereinigte Dokument stellt die Schnittmenge der Modellelemente, sowie die jeweiligen Änderungen farblich unterscheidbar in einem Diagramm dar. Das System ist nur für UML-Klassendiagramme ausgelegt und gilt daher als sprachspezifisch.

CoObRA CoObRA[3], vorgestellt von Schneider et al. [161], ist ein Framework zur Versionierung von UML Modellen. Es wurde im Umfeld des UML-CASE-Tools Fujaba[4] entwickelt, soll aber auch anderweitig einsetzbar sein. Das Framework arbeitet operationsbasiert und verwendet statische IDs zur Identifikation von Elementen. Das System basiert auf einer Client/Server Architektur, der Client zeichnet alle Modelloperationen auf und der Server speichert die Daten in einem zentralen Repository. Das System bietet Undo/Redo Funktionalität auf Basis der aufgezeichneten Operationen und verwendet den optimistischen Versionierungsansatz. Das Framework wurde in die Generatorkomponente von Fujaba integriert, somit können Anwendungen, die mit Fujaba erzeugt werden mit CoObRA Funktionalität ausgestattet werden.

UMLDiff UMLDiff wurde von Xing und Stroulia [203] entwickelt. UMLDiff ist ein Algorithmus mit dem Fokus auf die Analyse von Software Design Evolution. Im Prinzip geht es dabei um das Finden und Aufzeigen von strukturellen Versionsunterschieden in objektorientierter Software. Als Input sind dementsprechend auch keine Diagrammversionen vorgesehen, sondern Quellcodeversionen aus einem VCS. Aus dem Anwendungsbereich ergibt sich automatisch, dass der Elementabgleich nicht über statische IDs erfolgen kann, denn Quellcodeelemente haben keine zugeordneten IDs. Der Abgleich erfolgt daher über Heuristiken zur Namens- und Strukturähnlichkeit. Dazu werden aus dem Quellcode, über Reverse Engineering, Klassenmodelle erzeugt und die Heuristiken darauf angewendet. Zur Ermittlung der Strukturähnlichkeit werden nicht nur die Attribute und Methoden der Klassen verwendet, sondern auch die Beziehungen zu anderen Klassen. Diese Beziehungen werden, z.B. über implementierte Interfaces, Methodenaufrufe und Schreib- bzw. Lesezugriffe, auf Attribute ermittelt. Die Genauigkeit der Ergebnisse kann über einen Grenzwert gesteuert werden. Umso höher der Grenzwert ausfällt, desto genauer sind die Ergebnisse. Integriert wurde UMLDiff in ein Werkzeug namens JDEvAn[5]. Trotz des Namens ist das Programm nicht auf die Analyse von Java Software beschränkt, über einen Erweiterungsmechanismus können andere Programmiersprachen ergänzt werden. UMLDiff erkennt in der beschriebenen Variante nur atomare Änderungen, wurde aber in einer weiteren Version um das Auffinden zusammengesetzter Änderungen, also Refactorings, erweitert [204]. Ermöglicht wird das durch Datenbankabfragen der zugrundeliegenden Datenstruktur. UMLDiff kann zwar, wie bereits erwähnt, für andere Programmierspra-

[2] engl. unified document
[3] **Co**ncurrent **Ob**ject **R**eplication fr**A**mework
[4] „**F**rom **U**ML to **J**ava **a**nd **b**ack **a**gain“, `http://www.fujaba.de`
[5] **J**ava **D**esign **Ev**olution and **An**alysis

chen angepasst werden, nicht jedoch für eine andere Modellierungssprache, da für die Analyse der Struktur die UML verwendet wird.

UMLDiff$_{cld}$ Ein schönes Beispiel für sprachspezifische Modelldifferenzierung liefert Girschick [68] anhand seines UMLDiff$_{cld}$ genannten Algorithmus. Trotz der beinah identen Namensgebung steht der Ansatz in keinerlei Verbindung zu UMLDiff von Xing und Stroulia [203]. Der Namenszusatz cld steht für Classdiagram, der Ansatz ist also speziell auf Klassendiagramme zugeschnitten. Zur Elementidentifikation werden keine IDs verwendet, stattdessen wird versucht, möglichst viele elementspezifische Informationen für den Elementabgleich zu verwenden. Dafür werden dann auch semantische Elementaspekte aus Klassendiagrammen einbezogen. Der Abgleich erfolgt Ebene für Ebene über eine feingranulierte Breitensuche. Die Vergleichsfunktion verwendet mehrere unterschiedliche strukturelle Ähnlichkeitsheuristiken. Die Art und Anwendbarkeit einer Heuristik ist jeweils vom aktuellen Element abhängig. Die Ergebnisse der angewendeten Heuristiken wird summiert, liegt der Gesamtwert über einem Grenzwert, werden verglichene Elemente als Elementpaar angesehen. Somit werden nicht nur identische Elemente erkannt, sondern auch ähnliche Elemente. Ob benutzerdefinierte Anpassungen möglich sind, geht aus der Beschreibung nicht hervor.

ADAMS Ursprünglich von Barone et al. [22] als COMOVER[6] vorgestellt, beschreiben De Lucia et al. [117] ein komplettes Client/Server Versionierungssystem für UML Modelle namens ADAMS. Das System sieht eine zentrale und feingranulare Verwaltung von Artefakten vor. Das bedeutet, dass jedes Modellelement separat versioniert wird und dabei eine eindeutige ID (UUID) erhält. Über diese ID sind die Elemente dann während der 3-Wege Differenzberechnung identifizierbar. Ein Vorteil von zentraler und feingranularer Verwaltung ist die Wiederverwendbarkeit von versionierten Elementen über Diagrammgrenzen hinaus. Damit sinkt automatisch die Gesamtzahl der versionierten Elemente pro Projekt. Das System ähnelt in weiten Teilen dem Odysses VCS von Oliveira et al. [144]. Der Hauptunterschied liegt in der Übertragung von Änderungen zwischen Client und Server. Während beim Odyssey VCS alle Elemente übertragen werden, bildet der ADAMS Client die Versionsdifferenz und sendet diese als Delta an den Server. Ein weiterer Unterschied liegt in der Unterstützung von 3-Weg Differenzen, während das Odyssey VCS nur 2-Weg Differenzen bildet.

3.2 Generische Ansätze

Die in diesem Abschnitt vorgestellten Systeme und Ansätze sind unabhängig von einem speziellen Metamodell einsetzbar. Hierbei drehen sich die zu Beginn des letzten Abschnitts erläuterten Vor- und Nachteile um. Die Systeme sind sehr flexibel im Umgang mit unterschiedlichen Sprachen, jedoch können spezifische Eigenheiten diverser Sprachen nicht berücksichtigt werden. In diese Kategorie wurden aber auch die generisch-adaptierbaren

[6] **CO**ncurrent **MO**del **VER**sioning

Ansätze aufgenommen. Hierbei werden die Vorteile der Generizität kombiniert mit Anpassungen an spezielle Eigenschaften von Sprachen.

Alanen und Porres Die Arbeit von Alanen und Porres [13] war eine der ersten die sich mit der Differenzbildung und dem Zusammenführen von Modellen beschäftigt hat. Die Besonderheit liegt in der Generizität des Ansatzes. Alanen und Porres beschreiben in dieser Arbeit mehrere Algorithmen, die zwar unabhängig von einem Metamodell funktionieren sollen, allerdings eine statische ID (UUID) vorschreiben. Die Methode wird lediglich beschrieben, eine prototypische Umsetzung wird nicht erwähnt. In einer weiteren Arbeit [126] beschreiben sie dann ein komplettes Versionierungssystem, ebenfalls sprachunabhängig und statische IDs voraussetzend.

Oda und Saeki Oda und Saeki [139] skizzieren bereits im Jahre 2005 ein komplett an eine Sprache anpassbares System. Im Unterschied zu anderen sprachspezifischen Ansätzen wird das System aber nicht anhand der Anforderungen einer Sprache entwickelt oder entsprechend angepasst, sondern es wird für jede Sprache ein komplett neues System erstellt und zwar modellbasiert. Oda und Saeki schlagen die automatische Generierung von angepassten Systemen für jede beliebige Modellierungssprache vor. Dazu wird in einem ersten Schritt ein Metamodell in einer meta-CASE Anwendung erstellt und dann daraus in einem zweiten Schritt das entsprechende sprachspezifische System generiert. Der Versionierungsansatz wird als operationsbasiert beschrieben. Ausgehend von einer Basisversion (baseline) werden alle durchgeführten Operationen aufgezeichnet und gespeichert. Eine bestimmte Version kann dann aus der Basisversion und den nachfolgend gespeicherten Operationen rekonstruiert werden.

Odyssey-VCS Oliveira et al. [144] stellen ein Client/Server Versionierungssystem namens Odyssey-VCS vor. Die Basis des Systems stellt das mittlerweile nicht mehr verfügbare MOF-Repository MDR[7] dar, das als zentraler Datenspeicher fungiert. Das darauf aufbauende System sieht eine feingranulare Verwaltung von Modellelementen vor. Feingranular bedeutet in diesem Zusammenhang, dass jedes Modellelement separat versioniert wird. Von der höchsten Ebene, der Modellebene abwärts, erhält jedes enthaltene Element, also z.B. jedes Package, jede Klasse, jedes Attribut, jede Methode und jede Assoziation eine eigene ID (UUID) und eine Versionsnummer. Über die einmal vergebene ID sind die Modellelemente bei jeder Aktion identifizierbar. Mit jeder Änderung wird die Versionsnummer erhöht. Es werden immer alle Elemente vom Client zum Server übertragen, nicht wie bei anderen Systemen nur die Änderungen (Deltas). Der Server speichert allerdings nur geänderte Elemente. Um Speicherplatz zu sparen wird auf unveränderte Elemente per Hardlink verwiesen. Das System bietet darüber die Möglichkeit das Systemverhalten bezüglich der Versionierung auf Elementebene zu konfigurieren. So können z.B. Attribute oder Methoden von der logischen Versionierung ausgenommen werden. Es werden dann zwar trotzdem alle neuen Versionen gespeichert, Versionierungsdaten werden jedoch nicht angelegt.

[7] **M**eta**D**ata **R**epository

SiDiff und SiLift Kelter et al. [93] stellen einen generischen Ansatz zur Differenzermittelung von UML Diagrammen vor. Der Ansatz arbeitet mit einem eigenen Datenmodell. Zu vergleichende Modelle werden vor dem Vergleich in dieses Datenmodell übersetzt. Dabei ist die Elementzuordnung konfigurierbar, also an unterschiedliche Sprachen anpassbar. Der Ansatz erstellt 2-Weg Vergleiche, der Elementabgleich verwendet eine gewichtete Ähnlichkeitsheuristik. Elemente und Kriterien, sowie Gewichte der Heuristik können vom Benutzer an spezielle Anforderungen angepasst werden, z.B. einer DSL. Der Vergleich arbeitet die Elementhierarchie von unten nach oben[8] ab. Wird ein Elementpaar gefunden, so werden die enthaltenen Elemente in umgekehrter Reihenfolge, also von oben nach unten[9] reevaluiert. Das Ergebnis ist eine Tabelle mit Elementpaarungen, Differenzen zwischen Elementen können hier einfach ermittelt werden. Analog zum Ansatz von Ohst et al. wird mit diesen Ergebnissen ein vereinigtes Modell erstellt. Der Ansatz wurde als Fujaba Plugin umgesetzt. Der generische Charakter wird vor allem über das eigene Datenmodell, die Ähnlichkeitsheuristik, sowie die weitreichenden Anpassungsmöglichkeiten des Systems erreicht. Der Ansatz wurde in weiterer Folge als SiDiff vorgestellt und laufend weiterentwickelt [197, 159, 188, 160]. In der letzten bisher vorgestellten Ausbaustufe wurde SiDiff [90] um einen Signaturabgleich und eine Vorauswahl von Elementen erweitert. Diese Schritte sorgen für eine Verkleinerung des Suchraums der Ähnlichkeitsheuristik, was letztlich der Performanz zugute kommt. Mit dem Projekt SiLift [89] kann SiDiff um Funktionen zur semantischen Elementgruppierung und Erkennung von zusammengesetzten Änderungen erweitert werden.

DSMDiff Yuehua et al. stellen mit DSMDiff [114] eines der ersten komplett generischen Systeme zur Modelldifferenzbildung vor. Der Name nimmt den Fokus schon vorweg, nämlich Domain Specific Modeling, siehe Abschnitt 2.1.4. Damit bedeutet generisch in dem Fall unabhängig von einem speziellen Metamodell. Andere bisher vorgestellte Ansätze sind zumeist auf eine einzelne Sprache zugeschnitten. Das ist hauptsächlich, bis auf wenige Ausnahmen, die Sprache UML. Die meisten dieser Systeme benutzen bei der Paarfindung IDs zur Identifizierung von Elementen (matching), womit auch die Spracheinschränkung erklärt ist. DSMDiff geht bei der Analyse zum Elementabgleich anders vor. Elemente werden anhand ihrer Signatur und strukturellen Ähnlichkeit abgeglichen. Die Signatur eines Knotenelements besteht aus dem domänenunabhängigen Typen, dem domänenspezifischen Typen und dem Namen des Elements. Die Signatur einer Kante besteht zusätzlich aus den Signaturen des Quellknoten- und des Zielknotenelements. Strukturelle Ähnlichkeit wird über einen Abgleich der Kanten eines Knotenelements erreicht. Erster Schritt ist der Vergleich der Signaturen von Knotenelementen. Ist das Ergebnis nicht eindeutig, wird die strukturelle Ähnlichkeit der Kanten zur Entscheidung hinzugezogen. DSMDiff wurde für das Metamodellierungswerkzeug GME entwickelt, dieses Vorgehen dürfte aber auch auf andere Systeme übertragbar sein. Anpassungen an konkrete DSMLs sind nicht beschrieben, der Ansatz ist somit nicht sprachspezifisch.

[8] engl. bottom-up
[9] engl. top-down

Rivera und Vallecillo Das System, das Rivera und Vallecillo [153] vorstellen, ist kein vollständiges Versionierungssystem, sondern ein interessanter formaler Ansatz zum Modellvergleich bzw. zur Umsetzung von grundlegenden Operationen zur Modellversionierung. Verwendet wird dafür Maude[10], ein System zur logischen Termersetzung, EMF als Modellierungstechnologie und die Atlas Transformation Language (ATL)[11], eine Modelltransformationssprache. Vorgestellt werden die Operationen zum Modellelementabgleich (match), zur Differenzbildung (diff), zur Anwendung einer Differenz auf ein Modell (do), die inverse Operation (undo) und ein Operator zur sequentiellen Differenzcomposition (diffComp). Zwei zu vergleichende Modelle werden dabei automatisch mit ATL in Maude Spezifikationen übersetzt. Zum Elementabgleich werden statische IDs (UUID) verwendet oder eine Heuristik zur Strukturähnlichkeit, falls keine IDs vorhanden sind. Integriert wurde der Ansatz in der auf Eclipse basierenden Entwicklungsumgebung Maudeling[12]. Der Ansatz ist durch die Verwendung von EMF sprachunabhängig, somit generisch. Obwohl das Thema nicht explizit angesprochen wird, sollten sprachspezifische Anpassungen prinzipiell möglich sein.

Odyssey-VCS 2 Die erweiterte Nachfolgeversion von Odyssey-VCS, also Odyssey-VCS 2 [135], baut nicht mehr auf MDR auf sondern auf EMF. Diese Änderung ist durch einen Versionssprung zu erklären, EMF unterstützt UML 2.x, MDR nicht. Der Rest des Systems ist im Prinzip gleich aufgebaut. Es wurde jedoch um einige Verbesserungen und Fähigkeiten erweitert. Die reflektive EMF API ermöglichte eine effizientere Elementverarbeitung und einen generischen 3-Wege Algorithmus zur Versionszusammenführung. Die Möglichkeit zum Branching und zur pessimistischen Versionierung durch Sperren von Elementen wurden eingebaut, ebenso Schnittstellen zur Systemerweiterung.

GenericDiff GenericDiff, vorgestellt von Xing [202], ist keine sprachunabhängige Variante von UMLDiff, auch wenn der Name das nahelegt. Es ist im Vergleich ein komplett neuer Ansatz, der sich einer Methode der Graphentheorie bedient. Dabei geht es um das Finden eines größtmöglichen, identischen Subgraphen in zwei zu vergleichenden Graphen. Im Unterschied zu UMLDiff ist GenericDiff an kein Metamodell gebunden, als Input werden direkt zwei Modelle entgegengenommen. Um diese Modelle interpretieren zu können, benötigt GenericDiff aber domänenspezifische Information in Form von entsprechender Konfigurationen und Anpassungen an das entsprechende Metamodell. Im Unterschied zu den anderen bisher vorgestellten Systemen, ist dieses System also tatsächlich generisch, also sprachunabhängig und gleichzeitig aber an spezielle Anforderungen anpassbar. Erste Ergebnisse, im Vergleich zu UMLDiff, waren eher unzufriedenstellend. Durch diverse Anpassungen wurden diese Ergebnisse deutlich verbessert, sodass GenericDiff in etwa gleich gute Ergebnisse wie UMLDiff erzielt [201].

[10] `http://maude.cs.illinois.edu`
[11] `http://www.eclipse.org/atl`
[12] `http://atenea.lcc.uma.es/index.php/Portada/Resources/Maudeling`

AMOR Mit AMOR haben Brosch et al. [35] ein sprachunabhängiges aber an konkrete Sprachanforderungen anpassbares System vorgestellt. Vorgeschlagen wurde das System bereits etwas füher von Altmanninger et al. [16]. In einer etwas späteren Arbeit [33] wurde das System bis ins kleinste Detail erklärt. AMOR setzt durchgehend auf bewährte Methoden der Domäne. Sprachunabhängigkeit wird durch den Aufbau des Systems auf der MOF-Implementierung EMF erzielt. Das System enthält kein eigenes Modellrepository und setzt auch keine spezielle Client/Server-Architektur voraus, vielmehr funktioniert es mit aktuellen VCS zur Textversionierung. Durch den zustandsbasierten Modellvergleich ist AMOR unabhängig von Modellierungseditoren. Modelle werden nach dem 3-Weg Prinzip verglichen, zum Elementabgleich werden, wenn möglich, IDs verwendet. Darüber hinaus können sprachspezifische Besonderheiten, bei der Paarbildung über benutzerdefinierte Regeln, eingebunden werden. Wie diese Regeln erstellt werden können, wird nicht beschrieben. Die Ergebnisse der mehrphasigen und feingranulierten Vergleichsprozedur werden jeweils als Modell abgebildet. AMOR erkennt neben atomaren Änderungen auch zusammengesetzte Änderungen, z.B. Refactorings. Da solche zusammengesetzten Änderungen sprachabhängig sind, müssen diese vorab vom Benutzer definiert werden. Dazu enthält das System einen speziellen Operationseditor, der alle Änderungsschritte aufzeichnet, analysiert und zur weiteren Verwendung in der Änderungs- und Konflikterkennung bereitstellt [88]. Auch in die Konfliktauflösung und die Zusammenführung von Modellen fließen benutzerdefinierte Aspekte ein. Das System ist durch die Unabhängigkeit von konkreten Modellierungssprachen, Modellierungswerkzeugen und Versionierungssystemen, sowie die weitreichenden Anpassungsmöglichkeiten enorm flexibel und stellt dadurch gewissermaßen ein Idealsystem zur Modellversionierung dar.

EMFStore Koegel und Helming [100] haben ein Modell-Repository für EMF Modelle namens EMFStore entwickelt. Es handelt sich dabei um ein EMF Repository, ähnlich CDO[13] aber mit erweiterter Funktionalität. Das System ist operationsbasiert, das heißt, es zeichnet alle durchgeführten Operationen auf. Eine Analyse zur Differenzermittelung entfällt daher. Wie bei operationsbasierten Ansätzen notwendig, besteht das System aus einer Client/Server Architektur. Die Versionierung erfolgt feingranuliert, jedes Element erhält eine eindeutige ID (UUID). Der Client verbindet sich mit dem Modellierungswerkzeug und zeichnet die durchgeführten Änderungen auf. Dazu bedient sich EMFStore einer EMF-Funktion, die automatisch über alle durchgeführten Änderungen informiert. EMFStore ist somit auf Modellierungswerkzeuge angewiesen, die EMF verwenden. Der zentrale Server speichert die durchgeführten Operationen nach Erhalt in einem Versionsbaum. Durch die Verwendung von EMF als Basis, gilt das System als generisch. EMFStore bietet viele Möglichkeiten für Erweiterungen bzw. Anpassungen. Dadurch können die Anforderungen verschiedenster DSLs erfüllt werden. Somit kann EMFStore auch als sprachspezifisch angesehen werden.

[13] **C**onnected **D**ata **O**bjects, `https://www.eclipse.org/cdo`

3.3 Semantische Ansätze

In diesem Abschnitt werden Systeme bzw. Ansätze vorgestellt, die sich semantischer Analysen bedienen, um Modelldifferenz zu erstellen oder zu verbessern. Anders als bei rein strukturellen Analysen fließen hier auch Bedeutungsaspekte von Modellelementen in die Differenzierung ein.

SMoVer Reiter et al. [152] beschreiben einen Ansatz zur Konflikterkennung auf semantischer Basis, der von Altmanninger [14] zum System SMoVer[14] weiterentwickelt wurde. Der Ansatz ist so interessant, dass er kurz besprochen werden soll, auch wenn dabei nicht der Modellvergleich im Vordergrund steht. Zur Erkennung von semantischen Konflikten werden die zu vergleichenden Modellversionen in sogenannte „semantische Ansichten“[15] transformiert. Das sind definierte Ansichten auf ein Modell mit Augenmerk auf spezielle semantische Aspekte. Ein Modell wird dabei anhand eines speziellen Metamodells transformiert, wodurch Bedeutungen sichtbar werden können, die im ursprüngliche Metamodell nicht sichtbar sind. Semantische Ansichten sind vergleichbar mit einem Perspektivwechsel, der den Blick frei macht auf zuvor verdeckte oder nicht sichtbare Informationen. Derartige Ansichten lassen sich anhand von drei semantischen Aspekten kategorisieren: equivalente Konzepte, statische Semantik und Verhaltenssemantik. Eine semantische Ansicht, mit Fokus auf equivalente Konzepte, kann Konflikte widerlegen, die aufgrund von syntaktischen Unterschieden in semantisch gleichbedeutenden Modellversionen auftreten. Durch Ansichten zur statischen Semantik können Verletzungen der statischen Regeln einer Modellierungssprache erkannt werden. Eine Ansicht zum Aspekt der Verhaltenssemantik kann Konflikte in Modellversionen erkennen, die sich durch verhaltensbasierte Seiteneffekte beeinflussen oder widersprechen, syntaktisch aber konfliktfrei sind. SMoVer verwendet zum Elementabgleich statische IDs. Modelldifferenzen werden über Modellzustände ermittelt. Der Ansatz ist durch den EMF Unterbau sprachunabhängig, also generisch. Da die semantischen Ansichten für jede Modellierungssprache erstellt oder angepasst werden können, ist SMoVer gleichzeitig sprachspezifisch.

Maoz et al. bzw. CDDiff/ADDiff Einen sehr interessanten Ansatz zur Differenzbildung von Diagrammversionen stellen Maoz et al. [125] vor. Dieser Ansatz beruht nicht auf dem elementweisen Vergleich der syntaktischen Diagrammdaten, sondern bedient sich der Semantik, die einem Diagramm innewohnt. Da die Semantik eines Modelldiagramms aber nicht einfach als vergleichbares Artefakt vorliegt, wird der Weg über Instanzbildung eingeschlagen[16]. Maoz et al. beschreiben diese Technik an Hand von Klassendiagrammen [124] und Aktivitätsdiagrammen [123]. Sie stellen dafür semantische Operatoren vor, die, angewendet auf zwei Diagrammversionen, sogenannte „Differenzzeugen“[17] produzieren. Ein Differenzzeuge belegt einen semantischen Unterschied zwischen zwei Diagrammversionen.

[14] **S**emantically enhanced **Mo**del **Ver**sion Control System
[15] engl: semantic view
[16] von Ebene M1 nach Ebene M0, siehe Abschnitt 2.1.4
[17] engl. diff witnesses

Für Klassendiagramme werden dabei Instanzierungen also Objektmodelle gesucht, die mit der einen Diagrammversion erzeugt werden können, mit der anderen Version jedoch nicht. Im Falle von Aktivitätsdiagrammen werden Ablauffolgen gesucht, die nur mit einer der beiden Versionen möglich sind. Interessant ist dabei, dass sich zwei Diagrammversionen syntaktisch unterscheiden aber trotzdem semantisch gleichbedeutend sein können. Maoz et al. betonen, dass diese Technik nicht dazu gedacht oder geeignet ist, syntaktische Vergleiche zu ersetzen. Vielmehr soll ein weiterer Vergleichsaspekt abgedeckt werden, um die Ergebnisse von Diagrammvergleichen zu verbessern bzw. präziser zu gestallten. Da der semantische Differenzoperator für jede Sprache bzw. Diagrammart angepasst werden muss, handelt es sich hier um einen sprachspezifischen Ansatz. In Anlehnung an die semantischen Operatoren ADDiff und CDDiff von Maoz et al. entwickeln Langer et al. [111] eine generische Umsetzung auf Basis von Ausführungsverfolgung[18]. Sie benutzen dafür die Metamodellierungssprache xMOF [127], die es ermöglicht Verhaltenssemantik zu modellieren. Maoz und Ringert [122] stellen ebenfalls eine generische Weiterentwicklung vor. Mit dem selbstentwickelten, sprachunabhängigen Framework Diffuse kombinieren sie syntaktische und semantische Aspekte der Modelldifferenzierung.

SiLift Kehrer et al. [91, 89] stellen mit SiLift eine semantische Technik vor, die helfen soll, dem Benutzer eine nachvollziehbare und verständliche Sicht auf Diagrammänderungen zu bieten. Wie in Abschnitt 2.4.3 bereits beschrieben, setzt der Benutzer beim Ändern eines Diagramms meist nur atomare Aktionen, die im zugrundeliegenden Datenmodell aber zu einer Vielzahl von Änderungen führen. Diese Änderungen werden nur indirekt vom Benutzer ausgelöst, z.B. durch das Erstellen einer Assoziation zwischen zwei Klassen. Solche Änderungen werden daher als untergeordnete Änderungen[19] bezeichnet. Die Anzeige aller untergeordneten Änderungen hat für den Benutzer keinen Mehrwert, sondern das Gegenteil. Da diese Änderungen im Normalfall nicht sichtbar sind, stellt die Anzeige einen Bruch in der Darstellung[20] und somit einen potentiellen Nachteil dar. Deshalb sollen diese untergeordneten Änderungen möglichst verborgen und durch einen aussagekräftigen und nachvollziehbaren Stellvertreter ersetzt werden, der die tatsächliche Menge der untergeordneten Änderungen in sich vereint. Das Problem des Erkennens von zusammengehörigen Gruppen untergeordneter Änderungen stellt im Prinzip einen Musterabgleich[21] dar. SiLift ermöglicht auch die Erkennung und Ersetzung von zusammengesetzten Änderungen[22], das sind Änderungen, die aus mehreren atomaren Änderungen bestehen, z.B. Refactorings. SiLift benötigt dazu eine Sammlung von Erkennungsregeln[23], diese Erkennungsregeln werden automatisch aus Bearbeitungsregeln[24] erstellt. Jede Bearbeitungsregel stellt entweder eine atomare oder eine zusammengesetzte Änderung dar. Ganz nach den MDSD Prinzipien werden diese

[18] engl. execution traces
[19] engl. low-level changes
[20] engl. representation break
[21] engl. pattern matching
[22] engl. composite changes
[23] engl. recognition-rules
[24] engl. edit rule

Bearbeitungsregeln zuerst modelliert und dann in Erkennungsregeln transformiert. SiLift benutzt dazu die Modelltransformationssprache Henshin[25]. Anhand des Regelwerks kann SiLift an unterschiedliche Sprachen angepasst werden, SiLift kann daher als generisch und sprachspezifisch bezeichnet werden.

EMF Diff/Merge EMF Diff/Merge[26] ist ein weiteres Projekt aus dem Eclipse Modeling Projekt und versucht die erzielbaren Ergebnisse auf EMF Modellen durch die Möglichkeit von weitreichenden Anpassungen zu verbessern. Das System unterstützt benutzerdefinierte Anpassungen in allen Phasen der Versionierung, sowie Konsistenzregeln und definierbare Geltungsbereiche für alle Erweiterungen. Das System möchte kein Ersatz für EMF Compare sein, sondern eine Aufwertung der bestehenden Funktionalität durch ein Framework zur Erkennung und Verarbeitung von semantischen Differenzen bieten.

3.4 Sonstige Ansätze

Dieser Abschnitt beherbergt Systeme bzw. Ansätze, die keiner anderen Kategorie eindeutig zuzuordnen sind oder deren Funktionalität nicht in erster Linie der Versionierung dienen.

Pounamu Zur etwa gleichen Zeit wie Oda und Saeki stellen Mehra et al. [128] das System Pounamu vor. Es handelt sich ebenfalls um ein meta-CASE Tool zur Entwicklung von DSLs. Der Versionierungsteil wird bei der Erstellung einer CASE Anwendung automatisch entsprechend angepasst. Modellvergleiche werden nach dem 2-Weg Prinzip über Java-Objekte ausgeführt. Die XML Datenstrukturen von Modellen werden vor dem Vergleichen in Java Objektgraphen transformiert, die Objekte werden anschließend verglichen. Differenzen werden in Pounamu Befehlsobjekte transformiert, diese Objekte stellen atomare Änderungen dar, z.B. Element hinzufügen bzw. entfernen. Im Hintergrund verwendet Pounamu ein per Plugin angepasstes CVS Repository zur Speicherung sämtlicher Modelldaten im XML Format.

Cicchetti et al. Cicchetti et al. [50] präsentieren kein System zur Modellversionierung, sondern stellen einen Ansatz zum Modellvergleich nach dem MDE Prinzip „everything is a model“ [41] vor. Dementsprechend sollen Modelldifferenzen, also die Ergebnisse von Modellvergleichen, als Modelle repräsentiert werden. Der Fokus liegt auf der Erstellung und Verarbeitung von Differenzmodellen. Ein Vorgehen beim Vergleich bzw. eine Vergleichsmethode wird nicht festgelegt. Letztlich ist es für den Ansatz unerheblich, auf welche Art die Daten des Differenzmodells ermittelt werden. Der Ansatz wird weitestgehend unabhängig von konkreten Umsetzungen skizziert, z.B. auch metamodellunabhängig. Kern des Ansatzes ist die automatische Erstellung eines Differenz-Metamodells aus dem Metamodell der zu vergleichenden Modelle. Das ursprüngliche Metamodell wird dabei um Metaelemente zur Darstellung von Differenzen erweitert. Auf Basis dieses Differenz-Metamodells wird

[25] `https://www.eclipse.org/henshin`
[26] `http://wiki.eclipse.org/EMF_DiffMerge`

dann ein Differenzmodell erzeugt, das alle notwendigen Informationen bereithält. Aus dem Differenzmodell können dann Modelloperationen zur weiteren Verarbeitung oder Konfliktanalyse ermittelt werden. Über Transformationen höherer Ordnung[27] können wiederum automatisch Transformationen erzeugt werden, die aus einem Vorgängermodell und den Modelldifferenzen das Nachfolgemodell erzeugen oder umgekehrt.

Selonen und Kettunen Selonen und Kettunen [169] stellen eine weitere interessante Möglichkeit zur Paarbildung von Modellelementen durch eine Analyse der Strukturähnlichkeit vor. Der Ansatz stellt also kein Versionierungskonzept dar, sondern beschäftigt sich mit einem Teilaspekt der Modellversionierung. Kern des Ansatzes ist die Definition der sogenannten Korrespondenzbeziehung zwischen zwei Elementen in konkurrierenden Modellversionen. Gemeint ist dabei die Korrespondenz im Sinne einer Entsprechung. Der Ansatz beschreibt eine Heuristik, die sich der abstrakten Syntax des Metamodells bedient. Modellelemente stehen anhand ihrer strukturellen Beziehungen zueinander in einem Kontext. Das Metamodell legt fest, wie Beziehungen von Modellelementen aufgebaut werden müssen, z.B. Pflichtelemente für eine Assoziation. Aus den Vorschriften des Metamodell können automatisch Regeln erstellt werden, die über die strukturellen Beziehungen von Elementen den jeweiligen Kontext überprüfen und abgleichen. Treffen die Regeln auf zwei Elemente in konkurrierenden Modellversionen zu, so besitzen sie den selben Kontext und können als Elementpaar angesehen werden. Metamodelle werden als MOF-Modelle vorausgesetzt, damit ist der Ansatz sprachunabhängig. Durch die Erstellung eigener Regeln oder die Verfeinerung automatisch erzeugter Regeln kann der Ansatz an beliebige Sprachen angepasst werden.

Epsilon Comparison Language Kolovos [103] stellt die Epsilon Comparison Language (ECL) vor, eine regelbasierte Sprache zum Vergleichen von Modellen. Die ECL baut auf Epsilon[28] auf. Epsilon stellt eine Infrastruktur zum Erstellen von Sprachen für das Modellmanagement dar. Kern von Epsilon ist die Epsilon Object Language (EOL[29]) die ihrerseits auf OCL basiert. Neben EOL und ECL enthält Epsilon noch eine Reihe weiterer Sprachen, z.B. die Epsilon Transformation Language (ETL[30]). Eine ECL Regel besteht grob aus einer Deklaration der anwendbaren Elementtypen und drei Teilen. Der erste Teil kann die Anwendbarkeit der Regel weiter einschränken, der zweite Teil ist für den Elementvergleich verantwortlich und der dritte Teil kann zusätzliche Aktionen ausführen, falls notwendig. Als Ergebnis wird ein Objektmodell (MatchTrace) erstellt, in das jeder Treffer (Match) eingetragen wird. Dieses Objektmodell kann beliebig weiterverarbeitet werden, z.B. als Bildschirmanzeige oder als Input für eine Transformation in ein anderes Modellformat. Die ECL ist keine fertige Lösung für ein konkretes Problem, sondern ein

[27] Das sind Transformationen die wiederum Transformationen erzeugen und/oder Transformationen als Input benutzen[32].

[28] **E**xtensible **P**latform for **S**pecification of **I**ntegrated **L**anguages for m**O**del ma**N**agement, `https://www.eclipse.org/epsilon`

[29] `https://www.eclipse.org/epsilon/doc/eol`

[30] `https://www.eclipse.org/epsilon/doc/etl`

flexibles Werkzeug zur Erstellung einer maßgeschneiderten Lösung für beliebige Modellierungssprachen. Die ECL ist somit gleichzeitig als generisch und sprachspezifisch zu betrachten.

RCVDiff Van den Brand et al. stellen mit RCVDiff [30] ein System vor, das den von den selben Autoren zuvor beschriebenen Ansatz in einem Prototypen umsetzt [28]. Das System hebt sich anhand von zwei Aspekten von anderen vorgestellten Systemen ab. Erstens entwerfen van den Brand et al. ein eigenes unabhängiges Metametamodell und stoßen dabei das übliche Instanzierungsprinzip für Metaebenen um, indem sie Modelle und Metamodelle gleichermaßen von einem Metametamodell ableiten. Zweitens bietet RCVDiff ein vom Benutzer konfigurierbares System für den Elementabgleich. Auf Metaelementebene kann zur Paarbildung aus vier Vergleichsmöglichkeiten gewählt werden. Zur Verfügung stehen Signaturabgleich, eine strukturelle Ähnlichkeitsheuristik, sprachspezifische Merkmale, sowie IDs. Der Ansatz bietet 2-Weg Vergleiche und wird als unabhängig von bekannten Metamodellierungsumgebungen, wie EMF oder GME, beschrieben. Es bleibt jedoch unklar zu welchen Systemen der Ansatz kompatibel ist.

Mirador Das System Mirador von Barrett et al. [24] stellt in der beschriebenen Version einen Prototypen dar, der eine hybride Strategie zum Elementabgleich vorstellt. Es wurden mehrere Abgleichstrategien implementiert und eine neue, auf der Elementgeschichte basierende Methode vorgestellt. Dazu werden historische Elementversionen analysiert, um Informationen für eine Ähnlichkeitsmetrik zu gewinnen. Die unterschiedlichen Strategien können einzeln oder gemeinsam angewendet werden. Bei der gemeinsamen Verwendung mehrerer Strategien kann der Benutzer den einzelnen Optionen Gewichte zuteilen, um das Vergleichsergebnis anzupassen. Das System kann als Fujaba Plugin oder eigenständig betrieben werden, benötigt aber das CoObRa Framework.

Zu Beginn der Entwicklung der modellbasierten Methoden wurde hauptsächlich die UML verwendet. Das Hauptaugenmerk lag daher, auch im Bereich der Versionierung, auf der Kompatibilität zur UML. Flexibilität und Anpassbarkeit an unterschiedliche Sprachen spielten kaum eine Rolle. Dementsprechend viele der Arbeiten und Systeme sind auf nur eine Sprache, meist UML, ausgerichtet [53, 93, 136, 142, 203]. Derart sprachspezifische Lösungen haben den Vorteil, dass sie an die Anforderungen der verwendeten Sprache angepasst sind und dadurch meist gute Ergebnisse erzielen können. Mit der zunehmenden Popularität und dem vermehrten Einsatz von DSLs, änderten sich die Anforderungen an Modellversionierungssysteme. Generische Lösungen können durch Verwendung von Metasprachen eine Vielzahl von unterschiedlichsten Sprachen bedienen. Sie haben aber den Nachteil, dass sie nicht an spezielle Anforderungen einzelner Sprachen angepasst sind. Generische Lösungen sind daher meist durch eingeschränkte Leistungen, bzw. im Vergleich zu sprachspezifischen Lösungen, schlechteren Ergebnissen gekennzeichnet. Der Trend geht klar in Richtung generischer, jedoch adaptierbarer Lösungen. Dabei wird grundlegende Funktionalität in sprachunabhängiger Weise erzielt und durch spezielle Anpassungen an einzelne Sprachen angepasst. Brosch et al. [33, 36] unterstreichen klar

die Vorteile dieses Ansatzes. Die Entwicklung von Systemen, wie EMF Compare, AMOR, SiDiff und SiLift unterstützen diesen Trend auch durch ihre technologische Flexibilität und die Unabhängigkeit von speziellen Editoren und Persistenztechnologien.

In den letzten fünf Jahren sind wenig wirklich neue Ansätze dazugekommen, die konzeptuellen Möglichkeiten scheinen zumindest was Strukturanalysen betrifft insgesamt recht gut abgedeckt zu sein. Jüngere Arbeiten verfeinern und/oder kombinieren bestehende Ansätze oft zu neuen Ansätzen, wie z.B. die Arbeit von Khelladi et al. [95] oder die Systeme DiCoMEF [108] und Mondo [106]. Zwei Ansätze, die eigentlich zurück zum Anfang der Versionierungsproblematik von Modellen gehen und einen anderen Weg einschlagen als alle hier vorgestellten Konzepte, sind die Arbeiten von van Rozen und van der Strom , sowie Asenov et al [18]. Sie beschäftigen sich mit der Verbesserung von Modellversionierung auf Textbasis [157] bzw. Zeilenbasis [18].

Auf der Seite der semantischen Methoden scheint es noch Potential für weitere Innovationen zu geben. Die Arbeit von Cicchetti und Ciccozzi [49] zur Trennung von linguistischen und ontologische Aspekten in der Modellversionierung zeigt jedenfalls einen Weg für mögliche Weiterentwicklungen auf dem Gebiet auf.

KAPITEL 4

Analyse von SysML-Diagrammvergleichen

In diesem Kapitel werden die Ziele und das Vorgehen der Analysephase beschrieben, sowie die Ergebnisse der Analyse vorgestellt. Diese Ergebnisse fließen unmittelbar in die Definition von Problemfällen im nächsten Abschnitt, sowie in die Zielsetzungen und in weiterer Folge in die Umsetzungsphase ein.

In der im Weiteren beschriebenen Analysephase wurden konkret zwei Ziele verfolgt. Zum einen sollten Probleme, die beim Vergleichen von SysML-Diagrammen auftreten, gefunden und analysiert werden. Zum anderen sollten aus der Menge der Erweiterungsmöglichkeiten von EMF Compare passenden Ansatzpunkte zur Behebung der gefundenen Probleme ermittelt werden.

Zum Einsatz kam dabei, wie bereits in Abschnitt 2.5 beschrieben, der Papyrus Modelleditor in einer speziell angepassten Version aus dem Paket der Collaborative Modeling Initiative[1]. Es handelt sich dabei um den Papyrus Editor in der jeweiligen Nightly-Version, sowie um diverse, speziell für den Modellvergleich mit Git angepasste Paketversionen. Weiterführende Informationen zu den verwendeten Programmpaketen und Versionen sind im Abschnitt 2.5 zu finden.

Als mit der Arbeit an der Analyse begonnen wurde, basierte die aktuelle Version des Collaborative Modeling Pakets auf Eclipse Mars (4.5). Eben diese Version wurde zur Erstellung der Modelle für die Analyse verwendet. Der Papyrus Editor in der für Eclipse Mars zur Verfügung stehenden Version (1.1.4) unterstützte zu dem Zeitpunkt lediglich SysML 1.1. Daher basieren auch sämtliche Modelle, Analysen und funktionale Erweiterungen auf diesem Standard. Ob die Ergebnisse der Arbeit auch kommende, höhere SysML Versionen unterstützen, hängt von der Kompatibilität zwischen den Standards bzw. den Versionsänderungen der SysML Erweiterung für Papyrus ab.

[1] `http://www.collaborative-modeling.org`

Nach der Veröffentlichung der nächsten Version des Collaborative Modeling Pakets, basierend auf Eclipse Neon (4.6), musste eine Inkompatibilität zwischen dem Papyrus Komponenten festgestellt werden. Modelle, die unter Eclipse Mars erstellt wurden, konnten unter Eclipse Neon nicht geöffnet werden. Leider funktionierte auch die halbautomatische Migration der Daten nicht wie gewünscht. Aus diesem Grund wurde die Entscheidung getroffen, die Arbeit auf Basis der Eclipse Mars Version des Collaborative Modeling Pakets fortzusetzen. Ein späterer Wechsel auf eine höhere Version wurde nicht durchgeführt, hauptsächlich aufgrund der Inkompatibilitäten der erstellten Modelle mit neueren Papyrus Versionen. Ein Umstieg hätte die Wiederholung der ersten Analysephase bedeutet.

In Anhang B findet sich eine Einführung in den Umgang mit EMF Compare und der Vergleichsansicht, sowie ausführliche Erläuterung der durchgeführten Schritte beim Diagrammvergleich. Es werden darin verschiedene notwendige Begriffe und Vorgehensweisen festgelegt und beschrieben, die für das Verständnis, sowie die Überprüfbarkeit und die Nachvollziehbarkeit der Analyse als wichtig erachtet werden.

Im Folgenden wird die erste Analysephase und die Ergebnisse daraus beschrieben. Die Ergebnisse der zweiten Analysephase finden sich aus Strukturgründen im Abschnitt 2.5.4.

4.1 Analyse zur Erhebung von Problemen beim Vergleichen von SysML-Diagrammen

Wie bereits erwähnt wurde die Version 1.1 der SysML verwendet, zur Elementreferenz wurde die Notationsübersicht von Weilkiens [194] verwendet. Da sich die von Papyrus in den verschiedenen Diagrammeditoren angebotenen Elemente nicht vollständig mit der verwendeten Elementreferenz deckten, wurden nur Elemente analysiert, die in der Elementreferenz zu finden waren.

Im Abschnitt 2.2.3 wurde bereits beschrieben, dass SysML neun Diagrammarten enthält. Allein vier Diagrammarten wurden unverändert aus UML übernommen. Die Analyse beschränkte sich aus diesem Grund auf die fünf Diagrammarten, die neu hinzugekommen sind oder für SysML erweitert wurden. Für jede analysierte Diagrammart wurde ein eigenes Git Repository angelegt. Der Grund dafür liegt in der EGit Benutzeroberfläche. Hier entsteht sehr schnell eine unübersichtliche Struktur bzw. Liste, wenn mit vielen Branches gearbeitet wird. Die Vergleiche wurden auf Basis von atomaren Änderungen vorgenommen, siehe Abschnitt 2.4.3. Jede vorgenommene Änderung wurde als eigener Zweig[2] im jeweiligen Repository angelegt. Überprüft wurden die vier von EMF Compare differenzierten Änderungsarten der Erzeugung (add), der Aktualisierung bzw. der Modifikation (update), der Verschiebung bzw. Verlegung[3] (move) und des Entfernens (remove) eines Elements.

[2] engl.: branch

[3] Im EMF Compare Kontext ist die Änderungsart „move“ gegeben, wenn ein Element den übergeordneten bzw. enthaltenden Container wechselt, z.B. wenn ein Block in ein (anderes) Paket verschoben wird.

Das SysML Pendant zu einer UML Klasse ist ein Block. Im Unterschied zur UML Klasse besteht ein SysML Block aus deutlich mehr Elementen. Die atomare Aktion des einfachen Erstellens eines SysML Blocks in einem Diagramm hat 65 Änderungen zu Folge. Insgesamt aber ist der Vergleich von SysML Diagrammen nicht schwieriger, als der Vergleich von UML Diagrammen. Das zugrundeliegende Prinzip bleibt dasselbe. Der Vergleich fällt jedoch aufwändiger aus, da eben viel mehr Elemente betrachtet bzw. vergleichen werden müssen.

Den Ausgangspunkt bildete jeweils ein leeres Papyrusprojekt, das entspricht einem Gesamtmodell ohne Nutzdaten, also auch ohne Diagramme. Diese Voraussetzung ist notwendig, um etwaige Probleme bei der Diagrammerstellung erkennen zu können. Bei der Projekterstellung ist darauf zu achten, dass kein Diagramm zur optional angebotenen automatischen Erzeugung ausgewählt werden darf. Den ersten Versionsschritt nach der Projekterzeugung bildete daher jeweils die Diagrammerstellung.

Nach der Erstellung eines leeren Diagramms, erkennt Eclipse keine speicherbaren Änderungen, deshalb muss das Modell über den Menübefehl „File > Save As...“ gespeichert werden, um Daten zu erhalten, die verglichen werden können. Alternativ kann auch einfach irgendein Element ins Diagramm eingefügt und nach dem Speichern wieder entfernt werden.

Den nächsten Schritt stellt dann jeweils die Erzeugung eines Elements bzw. die Erzeugung von mehreren Elementen im leeren Diagramm dar. Mehrere Elemente nur dann, wenn diese Elemente für das eigentlich zu analysierende Element vorausgesetzt sind, z.B. zwei Blöcke zur Analyse einer Assoziation. Danach folgt jeweils eine Modifikation, das Verschieben bzw. Verlegen[4] und abschließend die Entfernung des Elements. Wann immer es notwendig bzw. zielführend war, wurden Zwischenschritte eingefügt, z.B. um weitere benötigte Elemente oder Attribute zu erzeugen.

Aus diesem Vorgehen ergibt sich eine Versionsstruktur, die wiederholbare Vergleiche zwischen unterschiedlichen Versionspunkten ermöglicht, siehe Abb. 4.1. Analysiert wurden jeweils die Vergleiche zwischen zwei Zweigen, z.B. zwischen der Erzeugung eines Blockdiagramms (`create-bdd`) und der Erzeugung eines Blocks (`add-block`). Das erwartete Resultat wurde auf Basis einer pragmatischen Erwartungshaltung gebildet und zwar, dass für jede getätigte Benutzeraktion im Diagramm eine Differenz in der Vergleichsansicht angezeigt wird. Dabei wurden das erwartete Ergebnis, sowie das tatsächliche Ergebnis verglichen und protokolliert. Dabei ist zu erwähnen, dass EMF Compare Änderungen des Modells und Änderungen des Diagramms differenziert, siehe dazu die Erläuterungen in Anhang B.1.

[4] Diese Änderungsart wurde nur dann überprüft, wenn Elemente verschoben bzw. verlegt werden können.

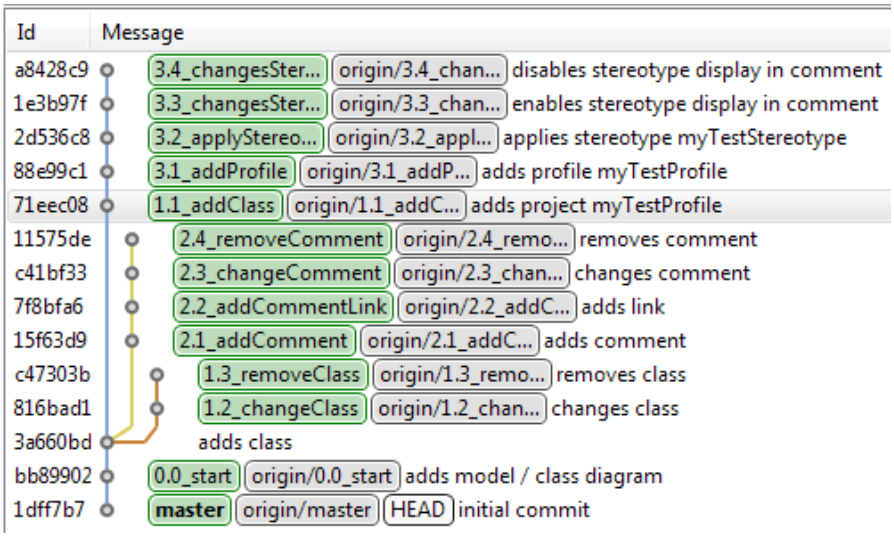


Abb. 4.1: Versionsstruktur mit schrittweisen, atomaren Änderungen

4.2 Ergebnisse der Analyse

Die Resultate der Analyse sind mehrere Vergleichsprotokolle in Form von Tabellen, für jede überprüfte Diagrammart wurde eine Tabelle erstellt, siehe Anhang C.

Die Analyse führte zur Feststellung eine Reihe von Problemen, quer durch alle Diagrammarten. Insgesamt wurden 103 Diagrammvergleiche durchgeführt, dabei traten 51 Mal Probleme auf. Es handelt sich dabei immer um unerwartete Diagrammdifferenzen (UDD). Diese resultieren aus Diagrammobjekten, die nicht direkt vom Benutzer erzeugt wurden - zumindest nicht unmittelbar - und deshalb auch nicht als Differenz erwartet werden. Hauptsächlich wird die Idealvorstellung der pragmatischen Erwartungshaltung einer 1:1 Relation von durchgeführten Diagrammänderungen zu dargestellten Änderungen in der Vergleichsanzeige verletzt.

Aus der Menge der gefundenen Probleme wurden zwei konkrete Problemfälle (PF) repräsentativ zur weiteren Bearbeitung ausgewählt. Zum einen sind das unbenannte UDDs, die in allen analysierten Diagrammarten vorkamen (PF1), siehe Abb. 4.2 im nächsten Abschnitt. Zum anderen sind das UDDs, die beim Erzeugen von SysML Ports auftraten (PF2).

Während der Detailanalysen bzw. der Implementierung von Erweiterungen zur Lösung der beiden oben genannten Problemfälle, wurden zwei weitere Probleme entdeckt. Diese Probleme dürften wahrscheinlich noch häufiger als die oben erwähnten UDDs vorkommen, da sie bei so gut wie jedem Vergleich auftreten. Es handelt sich dabei einmal um die doppelte Anzeige von verfeinerten Änderungen und weiters um die nicht korrekt dargestellte Beziehungsstruktur zwischen Makro- und Subdifferenzen. Diese beiden Probleme wurden ebenfalls zur weiteren Untersuchung und auch zur Behebung vorgesehen (PF3

und PF4). Die nunmehr vier Problemfälle wurden jeweils einer Detailanalyse unterzogen, die Ergebnisse werden im nächsten Abschnitt genauer vorgestellt.

Die erstellten Repositories bzw. die Diagramme ausgewählter Branches wurden nach der Analysephase für Testzwecke während der Implementierungsphase und danach zur Evaluierung der Ergebnisse wiederverwendet.

4.3 Konkrete Problemstellungen

Nach der Problemerhebungsanalyse, die im vorigen Abschnitt beschrieben wurde, geht es in diesem Abschnitt um die Ergebnisse der Detailanalysen, die zu den ausgewählten Problemfällen (PF1-PF4) durchgeführt wurden. Die Einteilung in Problemfälle dient hauptsächlich der Abgrenzung und eindeutigen Benennung der Probleme.

4.3.1 PF1: Unbenannte UDDs

Bei dem ersten ausgewählten Problemfall (PF1) handelt es sich um unbenannte UDDs, siehe Abb. 4.2. Dieses Problem ist am häufigsten aufgetreten (19 Mal) und ließ sich bei allen analysierten Diagrammarten feststellen. Beim Diagrammvergleich werden Änderungen angezeigt zu denen es keine Diagrammelemente zu geben scheint. Entsprechende Elemente tauchen weder in der Diagrammanzeige auf, noch lassen sie sich im Modellexplorer auffinden.

Ursprüngliche wurde von der Vermutung ausgegangen, dass dieses Verhalten eine SysML spezifische Besonderheit darstellt. Diese Vermutung konnte im Laufe der Detailanalyse nicht bestätigt werden. Im Gegenteil, das Problem stellte sich unter bestimmten Bedingungen als Normalfall und damit als weitaus verbreiteter als angenommen heraus.

Bereits im zweiten durchgeführten Diagrammvergleich (siehe Anhang C.1, Aktion: add block, Branch: 1.1) wurden unbenannte UDDs festgestellt, siehe Abb. 4.2. Bei nur einer Änderung im Diagramm, nämlich dem Hinzufügen eines Blocks, werden 4 Differenzen in der Vergleichsanzeige gemeldet. Zwei Differenzelemente wären erwartet, erstens das Modellelement und zweitens das Diagrammelement. Dass in Abb. 4.2 allein drei Diagrammdifferenzen angezeigt werden, deutet auf den Ursprung des Problems in der Diagrammdarstellung hin. Die Bezeichnung der unbenannten UDDs in der Vergleichsansicht lässt keinen Rückschluss auf den Ursprung bzw. die Zugehörigkeit der Elemente zu. Einzig die Markierung abhängiger bzw. in einer Beziehung stehender Elemente ließ vermuten, dass die Elemente „Block1“, „Block null“ und „Connector null“ irgendwie etwas miteinander zu tun haben müssen. Um herauszufinden was das Erscheinen der unbenannten UDDs auslöst, wurde eine Detailanalyse durchgeführt, eine genaue Beschreibung der Schritte und der Hilfsmittel findet sich in Anhang B.2. Die Detailanalyse ergab zweifelsfrei einen Zusammenhang mit Stereotypen. Die Klassennamen der gefundenen Elemente belegen das eindeutig. Das Differenzelement „Shape <null>“ stellt ein Objekt vom Typ `StereotypeComment` dar, das Element „Connector <null>“ ein Objekt vom Typ `StereotypeCommentLink`.

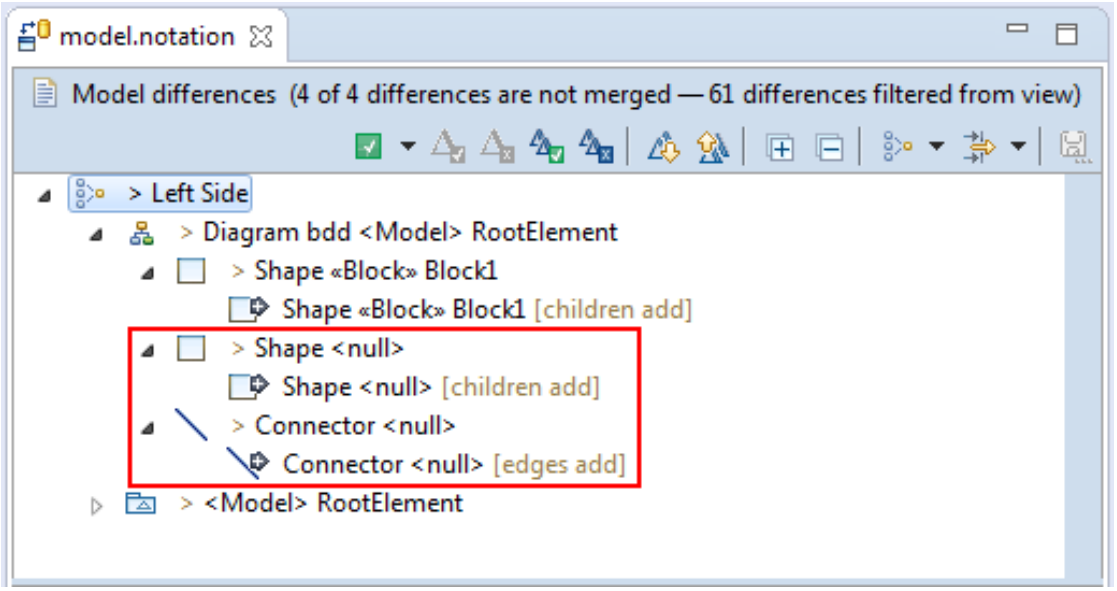


Abb. 4.2: unbenannte UDDs in der Vergleichsansicht

Da es offensichtlich einen Zusammenhang zwischen unbenannten UDDs und Profilen bzw. zugewiesenen Stereotypen gab, scheint dieses Phänomen nicht nur auf SysML beschränkt zu sein. Eine schnelle Analyse anhand eines einfachen UML Diagramms und eines einfachen Profils mit nur einem Stereotyp, bestätigt diese Vermutung. Zumindest für die beiden Sprachen UML und SysML trifft die Vermutung zu. In weiterer Folge konnten diese Anzeigeoption für Stereotypen auch in der UML Spezifikation gefunden werden[5]. Da SysML keine komplett eigenständige Sprache ist, sondern als ein Profil für UML realisiert wurde und somit auf UML aufbaut und auch große Teile von UML verwendet[6], liegt nur nahe, dass SysML diese Optionen von UML erbt. Tatsächlich werden die Möglichkeiten zur Darstellung von Stereotypen in der SysML Spezifikation sogar explizit genannt und beschrieben[7]. Es ist zu vermuten, dass dieses Problem relativ breit wahrgenommen werden kann, generell beim Einsatz von Profilen etwa oder beim Einsatz von anderen Sprachen, die ebenfalls auf UML basieren.

Das Problem der unbenannten UDDs trat in den getesteten Fällen immer dann auf, wenn in Papyrus Diagrammen mit Profilen gearbeitet wurde und Elemente mit Stereotypen versehen wurden. Diese Stereotypen wurden dabei entweder automatisch zugeordnet, z.B. beim Erstellen eines SysML-Blocks, oder manuell zugeordnet. Papyrus bietet die Möglichkeit zugeordnete Stereotypen unterschiedlich darzustellen, siehe Abb. 4.3. Erstens als Zusatz im Kopfbereich des Elements, zweitens in einer eigenen Abteilung und drittens als Kommentar. Und für genau den dritten Fall werden vorsorglich automatisch Elemente

[5] UML 2.5.1Spezifikation [148], Annex B.2.6
[6] UML4SysML, siehe dazu Abschnitt 2.2.3
[7] OMG SysML 1.5 Spezifikation [147], Kapitel 8 Blocks

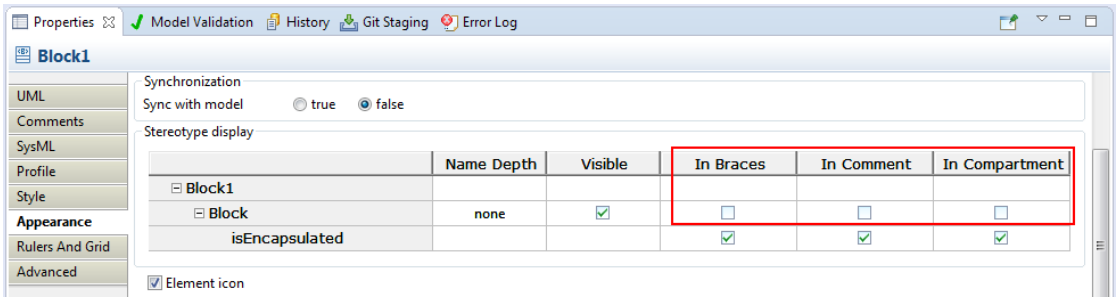


Abb. 4.3: Optionen zur Anzeige von zugewiesenen Stereotypen

erzeugt. Im Grunde handelt es sich um ein Kommentarsymbol und einen Verbinder, der den Kommentar an das eigentlichen Element heftet. Ohne Zutun des Benutzers werden Stereotypen im Kopfbereich des Elements angezeigt, über die Einstellungsmöglichkeiten der Elementeigenschaften[8] können aber alle Optionen unabhängig voneinander aktiviert oder deaktiviert werden, siehe Abb. 4.4.

Jene Elemente, die in der Differenzansicht als unbenannte UDDs auftauchen, werden erst mit Aktivierung der entsprechenden Option zur Anzeige von Stereotypen als Kommentar im Diagramm sichtbar. Ohne die Aktivierung dieser Option werden diese Elemente gar nicht benötigt. Warum Papyrus diese Elemente also „auf Verdacht“ automatisch erzeugt und nicht erst bei Aktivierung der entsprechenden Option einfügt, bleibt fraglich. Interessant, allerdings nicht Inhalt dieser Analyse, wäre hierbei die Frage nach dem Mehraufwand durch die automatisch erstellten Elemente beim Vergleichen von Diagrammen, sowie eine Abwägung zum Aufwand der Erzeugung der entsprechenden Elemente im Bedarfsfall, also bei Aktivierung der entsprechenden Option.

Die Behebung des Problems besteht nun darin die beiden oben beschriebenen und in Abb. 4.2 markierten Elemente in der Differenzliste zu finden und entsprechend zu deklarieren. Also eigentlich durch anzeigen von entsprechend bezeichneten Makrodifferenzen. Diese Makrodifferenzen können dann bei Bedarf per Filter entfernt werden. Hinzu kommt, dass die zugehörigen bzw. untergeordneten Elemente richtig gruppiert werden sollen, siehe auch die Beschreibung zu PF4. Als zugehörige bzw. untergeordnete Elemente können all jene Elemente bezeichnet werden, die benötigt werden, um einen `StereotypeComment` bzw. einen `StereotypeCommentLink` darzustellen, siehe Abb. B.4 in Anhang B.1.

4.3.2 PF2: SysML Ports

Bei dem zweiten zur Bearbeitung ausgewählten Problemfall (PF2) handelt es sich um SysML Ports. Diese Elemente stellen sozusagen Anschlüsse oder Interaktionspunkte in oder an Blöcken dar. Das Element `Port` ist eigentlich kein spezifisches Element von SysML, es kommt bereits seit Version 2 in der UML vor. Allerdings ist der Stellenwert

[8] View Properties > Tab Appearence > Abteilung Stereotype display

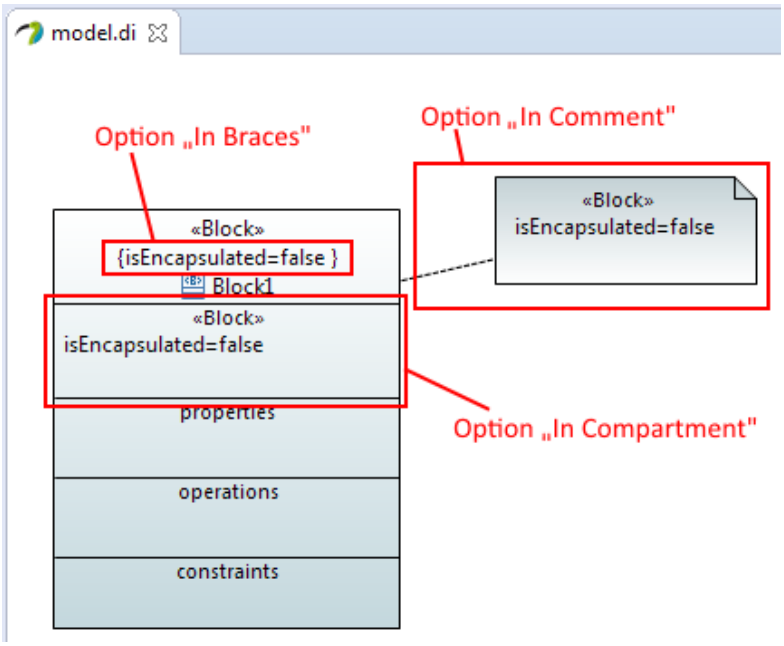


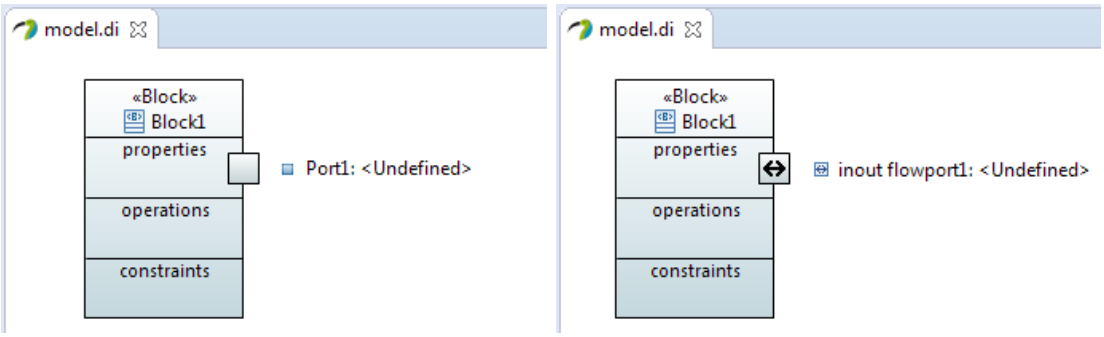
Abb. 4.4: Diagrammdarstellung der beschriebenen Anzeigeoptionen

von Ports in der UML denkbar gering[9]. Ports werden an Klassen und Komponenten unterstützt und können damit in drei Diagrammarten vorkommen. Dazu gehören das Klassendiagramm, das Komponentendiagramm und das Kompositionsstrukturdiagramm [195][10]. SysML übernimmt das Element als Anschluss bzw. Interaktionspunkt für Blocks. Bis zur Version 1.2 benutzt SysML den UML Port in unmodifizierter Form mit der Bezeichnung `StandardPort`, siehe Abb. 4.5a, sowie eine stereotypisierte Version mit der Bezeichnung `FlowPort`, siehe Abb. 4.5b. Der `StandardPort` bieten der Umgebung Dienstleistungen über Schnittstellen an oder fordern diese ein. Über einen `FlowPort` werden Objekte mit der Umgebung ausgetauscht, dabei kann es sich um Materie, Energie oder Daten handeln [54]. Ab der SysML Version 1.3 gibt es diese Formen von Ports nicht mehr, sie werden durch den vollwertigen Port[11] und den Stellvertreterport[12] ersetzt.

Die Detailanalyse der SysML Ports gestaltete sich deutlich einfacher als die vorangegangene Analyse der unbenannten UDDs (PF1). Die Vorgehensweise und die Hilfsmittel, die während der ersten Analyse erarbeitet wurden, gelangten hier erneut zum Einsatz.

Bei Hinzufügen von `StandardPorts`, als auch `FlowPorts` kommt es zu ähnlichen problematischen Darstellungen in der Vergleichsanzeige, siehe Abb. 4.6. Bei einem `StandardPort` entstehen 29 Änderungen, ein `FlowPort` erzeugt 48 Änderungen. Mit dem Wissen aus der Analyse zu PF1 und der Tatsache, dass `StandardPorts` eigentlich reine

[9] In „Analyse und Design mit der UML 2.5“ erwähnt Oestereich [140] Ports nicht einmal.
[10] Papyrus erlaubt Ports allerdings nur bei Komponenten.
[11] engl.: `full port`
[12] engl.: `proxy port`



(a) `StandardPort` (b) `FlowPort`

Abb. 4.5: Diagrammdarstellung von SysML Ports

UML-Ports sind, kann hier bereits die Vermutung aufgestellt werden, dass die Diskrepanz in der Anzahl der Änderungen wohl auf ein Stereotyp zurückzuführen ist.

Die eigentlichen `Port` Differenzelemente werden an sich korrekt dargestellt. Auf der selben Ebene erscheinen jedoch vier UDDs. Wie bereits anhand von PF1 beschrieben, sind das Differenzen von Elementen die im Diagramm nicht sichtbar sind, siehe 4.2. Die `Port` Differenzelemente werden von `CSSShapeImpl` Elementen dargestellt, beim `Standardport` findet sich die Bezeichnung `shape_uml_port_as_affixed` im Attribut `type`, beim `Flowport` die Bezeichnung `shape_sysml_flowport_as_affixed`. Die Typbezeichnung des `Standardports` weist auf die Herkunft des Elements hin, wie oben beschrieben. Bei beiden `Port`-Typen werden vier UDDs angezeigt, siehe Abb. 4.6. Dargestellt werden diese UDDs von einem `CSSDecorationNodeImpl` Element mit Typbezeichnung `StereotypeLabel` und drei `CSSBasicCompartmentImpl` Elementen mit den Typbezeichnungen `StereotypeBrace`, `compartment_shape_display` und `StereotypeCompartment`. Beim `Flowport` kommen noch die beiden, bereits aus PF1 bekannten, unbenannten UDDs dazu.

Ähnlich den unbenannten UDDs in PF1, stammen auch die vier oben beschriebenen UDDs von Elementen, die im Diagramm normalerweise unsichtbar sind. Und auch diese UDDs können über diverse Diagrammeinstellungen sichtbar gemacht werden. Die Elemente mit Typ `compartment_shape_display` und `StereotypeCompartment` erscheinen im Diagramm, wenn die `Compartments` eines Blocks angezeigt werden, das kann im Diagrammeditor über die Kontexteingenschaften eines Elements eingestellt werden[13]. Das Element mit Typ `StereotypeBrace` kann über die Einstellungsmöglichkeiten der Elementeigenschaften sichtbar gemacht werden, zumindest die Methode `isVisible` retourniert `true`, im Diagramm erscheint das Element nicht. Das Element mit Typ `StereotypeLabel` hingegen liefert über die Methode `isVisible` immer `true`, obwohl es nicht sichtbar ist.

[13] Rechtsklick auf markiertes Element, Menü Filters -> Show/Hide Compartments

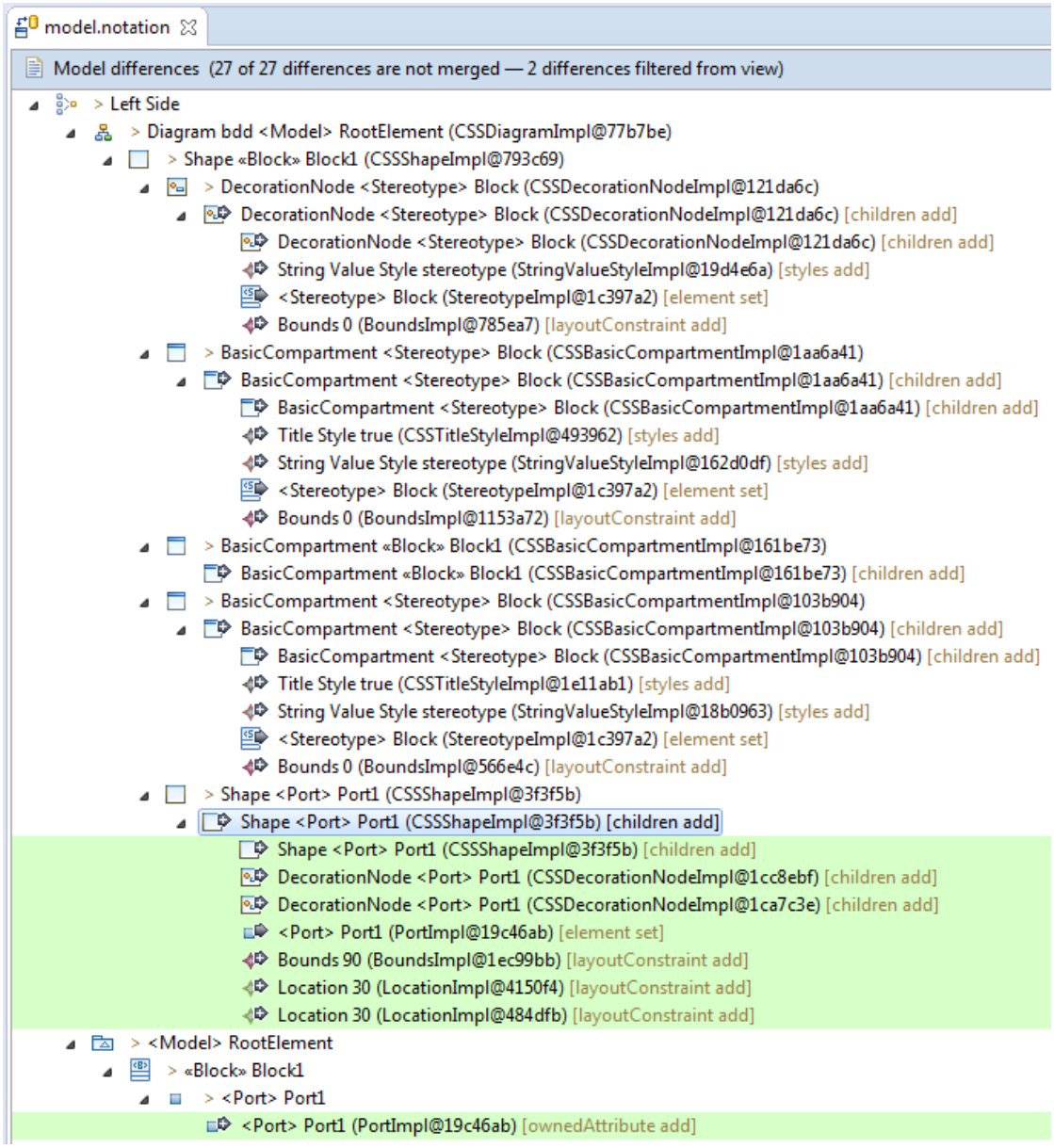


Abb. 4.6: Vergleichsansicht nach dem Erzeugen eines `StandardPorts`

Ein `FlowPort` ist eigentlich nur ein UML-Port mit zugewiesenem SysML-Stereotyp „FlowPort“. Es tritt daher die selbe Problematik auf, die bereits als PF1 beschrieben wurde. Der `StandardPort` besitzt kein SysML-Stereotyp, daher werden auch die unbenannten UDDs nicht erzeugt. Die vier anderen unsichtbaren Elemente haben wiederum mit der automatischen Erzeugung von Elementen im Papyrus Editor zu tun. Diese Elemente werden auf Vorrat erzeugt, ohne konkrete Notwendigkeit der Anzeige. Werden etwa bei einem Block alle `Compartments` sichtbar gemacht und erst danach ein `Port` hinzugefügt, so fallen diese vier unsichtbaren Elemente beim Vergleich als Differenzen weg, ein `StandardPort` erzeugt dann nur noch 9 Änderungen, ein `FlowPort` nur noch 28 Differenzen. Die Diskrepanz zu den vorigen Differenzzahlen ergibt sich durch die Gesamtzahl der Elemente, die für die automatischen Änderungen benötigt werden.

Das Problem tritt in ähnlicher Form auch unter UML auf, allerdings mit deutlich weniger Differenzelementen, nämlich nur 12. Hier wird nur das Element `CSSBasicCompartmentImpl` mit der Typbezeichnung `compartment_shape_display` automatisch erstellt, was zu insgesamt 3 Änderungen führt. Damit decken sich die Differenzsummen von UML-Port und SysML-StandardPort, da sämtliche Änderungen mit Stereotypen Bezug im UML Modell wegfallen.

Die Behebung des Problems besteht nun darin, die vier oben beschriebenen und in Abb. 4.6 erkennbaren Elemente in der Differenzliste zu finden und entsprechend zu deklarieren, also eigentlich im Anzeigen von entsprechend bezeichneten Makrodifferenzen. Diese Makrodifferenzen können dann bei Bedarf per Filter entfernt werden. Das bereits aus PF1 bekannte Problem soll hier bei der Lösung keine spezielle Beachtung finden, sondern idealerweise von der Lösung zu PF1 mit abgedeckt werden. Hinzu kommt, dass die zugehörigen bzw. untergeordneten Elemente richtig gruppiert werden sollen, siehe auch die Beschreibung zu PF4.

4.3.3 PF3: Doppelte Anzeige verfeinerter Differenzen

Zu diesem Problem kommt es bei jeder Verfeinerung von Diagrammdifferenzen. Betroffen sind alle Differenzen, die von einer EMF Compare Diagrammerweiterung erstellt werden. Eine Makrodifferenz wird meist anhand eines spezifischen Differenzelements erzeugt, außer z.B. bei einem `EdgeChange`[14]. Dieses spezifische Differenzelement wird in der Folge als Quelldifferenz bezeichnet. Weil die Makrodifferenz neu erstellt wird, gibt es kein entsprechendes Element im Diagramm. Für die Anzeige benötigt die Makrodifferenz aber ein referenziertes Diagrammelement, weshalb der Makrodifferenz im Verfeinerungsprozess das Diagrammelement der Quelldifferenz zugewiesen wird. Die Makrodifferenz und die Quelldifferenz referenzieren somit das selbe Diagrammelement. Die neu erstellte Makrodifferenz schlüpft sozusagen in das Gewand der Quelldifferenz. Damit ist die Quelldifferenz die primäre verfeinernde Subdifferenz[15] der Makrodifferenz. Die primäre Subdifferenz dient als Referenz auf das Haupt- oder Schlüsselelement eine Makrodifferenz, z.B. auf das `Shape` eines Blocks.

[14] `org.eclipse.emf.compare.diagram.internal.extensions.EdgeChange`
[15] engl.: prime refining

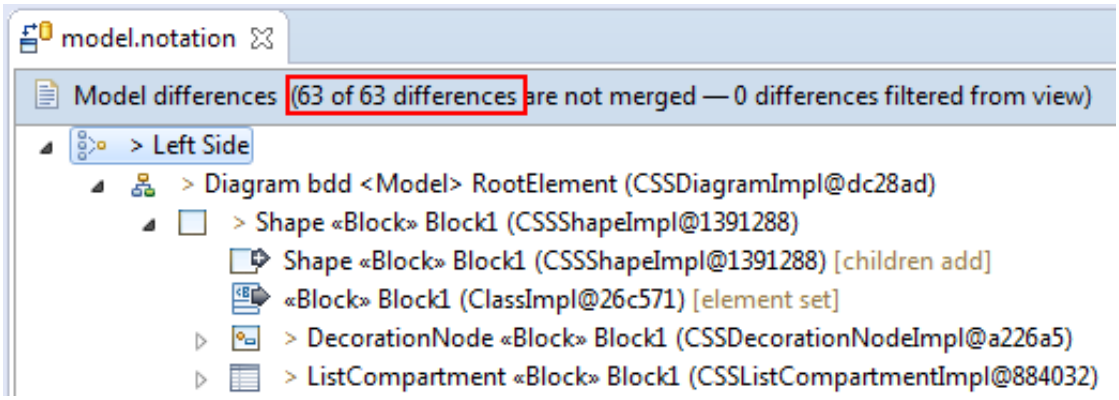


Abb. 4.7: Vergleichsansicht nach dem Erzeugen eines Blocks, ohne Verfeinerung, alle Filter deaktiviert

Bei der Erstellung der Baumstruktur für den `TreeViewer` der Vergleichsansicht wird nun die Makrodifferenz als lokale Wurzel ihrer Subdifferenzen in den Baum übernommen. Die Darstellung der Makrodifferenz fällt dabei der primären Subdifferenz zu. Danach wird über sämtliche Subdifferenzen iteriert. Jede Differenz, die in den Baum übernommen wird, zählt zu der Summe der angezeigten Differenzen. Und dabei kommt es zur Verdoppelung der primären Subdifferenz. Erstens wird die Makrodifferenz in der Gestalt der primären Subdifferenz dargestellt und zweitens wird die primäre Subdifferenz in der darunterliegenden Ebene noch einmal angezeigt. Da die Makrodifferenz ebenfalls gezählt wird, kommt es auch zur doppelten Zählung. Dieses Verhalten ist an der Gegenüberstellung von zwei Vergleichsansichten gut erkennbar.

Verglichen wurden die selben Versionszweige, Abb. 4.7 zeigt die Vergleichsansicht ohne Verfeinerung von Diagrammelementen, Abb. 4.8 zeigt die Ansicht mit Verfeinerung. Filter wurden beide Male deaktiviert, die angegebene Summe der Differenzen ist jeweils rot umrandet.

In der zweiten Abb. sind auch die betreffenden Elemente rot umrandet. Dieses Verhalten ist nicht SysML-spezifisch, es wurde bei allen durchgeführten EMF Compare Vergleichen festgestellt. Makrodifferenzen werden im Kontext dieser Arbeit aber nicht als tatsächlich neue Differenzen verstanden, sie fassen vielmehr bestehende und der Bedeutung nach zusammengehörende Differenzen zu einer stellvertretenden Differenz zusammen. Sie bilden sozusagen einen semantischen Stellvertreter in Form eines Differenzelements und der Funktion eines Containers. Die Anzeige beider Differenzelemente, also Makrodifferenz und primäre Subdifferenz, wird als redundant angesehen.

Die Behebung des Problems besteht nun darin, die doppelten Differenzelemente von der Anzeige und der Summenbildung auszuschließen, sofern dies mit vertretbarem Aufwand möglich ist.

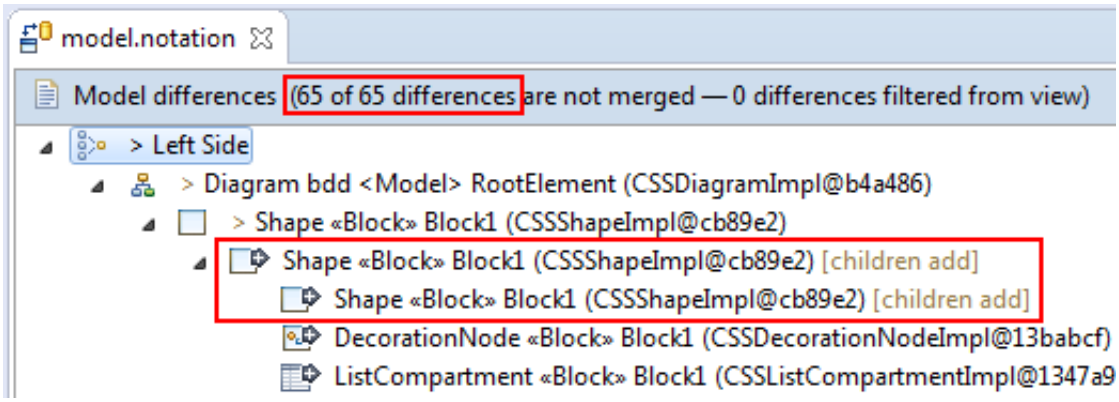


Abb. 4.8: Vergleichsansicht nach dem Erzeugen eines Blocks, mit Verfeinerung, alle Filter deaktiviert

4.3.4 PF4: Baumstruktur verfeinernder Differenzen

Die von EMF Compare gefundenen Differenzen werden, dem Vergleichsmodells entsprechend, intern gespeichert, siehe Abb. A.1 in Anhang A. Zu jedem gefundenen `Match`[16] können Differenzen (im Metamodell als `Diff` bezeichnet) abgelegt werden. Jeder `Match` kann, entsprechend der `Submatch`-Assoziation (eigentlich eine gerichtete Komposition), untergeordnete `Matches` besitzen. Diese untergeordneten `Matches` bzw. `Submatches` können wiederum Differenzen besitzen, es ergibt sich eine Baumstruktur aus `Matches`, `Diffs` und untergeordneten `Submatches` mit ihren jeweiligen `Diffs`, usw. Diese rekursive Struktur bildet die GMF Diagrammelemente, anhand der Eltern-Kind Beziehungen der Elemente, als Baumstruktur nach. Ein SysML Block besteht z.B. aus einem `Shape`, das `Shape` enthält mehrere `Compartments`, jedes `Compartment` besteht aus einer Bezeichnung usw.

Beim Aufbau des Vergleichsmodells werden durch die diversen nativen Erweiterungen[17] eigene, neue Differenzen erzeugt, die bestehende Differenzen kapseln, sogenannte Makrodifferenzen. Diese Makrodifferenzen fassen alle anderen notwendigen Differenzen zu einer einzelnen Differenz zusammen. Aus den 21 Differenzen einer hinzugefügten UML Klasse wird z.B. eine einzelne Differenz erzeugt und in der entsprechenden Elementebene ins Modell eingefügt. Diese Makrodifferenzen abstrahieren den tatsächlichen, darunterliegenden Aufwand, siehe als Gegenüberstellung die Abb. B.3 und B.4 in Anhang B.

Solche Makrodifferenzen stehen stellvertretend für eine Menge gefundener und konzeptuell zusammengehörender Differenzen, sie sind Containern nicht unähnlich. Makrodifferenzen fassen sozusagen Differenzen, die zu einem Konzept gehören, zusammen. Und bei eben dieser Zusammenfassung der zusammengehörenden Differenzen kommt es dazu, dass die Elementhierarchien außer Acht gelassen werden. Die zusammengefassten oder verfei-

[16] Matches werden auch für einseitig gefundene Elemente erstellt, also auch wenn kein passendes Element in der Vergleichsversion existiert. Die Bezeichnung `Match` ist daher ein wenig irreführend.

[17] z.B. `org.eclipse.emf.compare.diagram`, `org.eclipse.emf.compare.uml2`, etc.

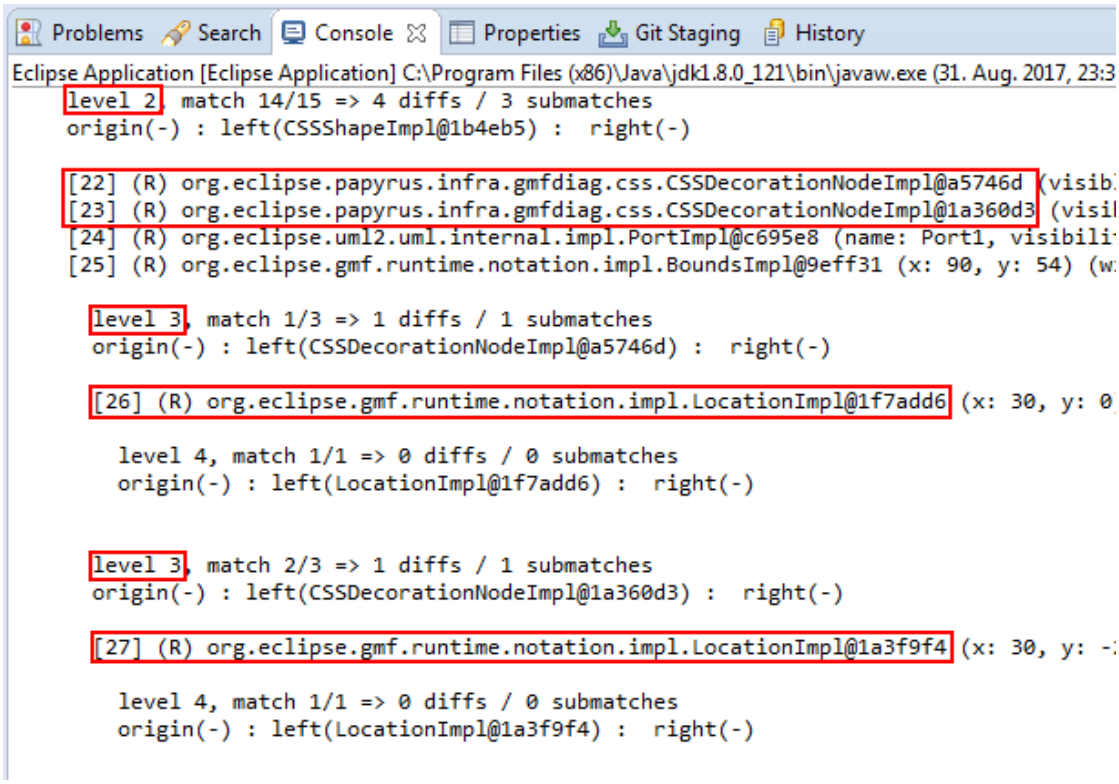


Abb. 4.9: Konsolenausgabe der Differenzstruktur eines `StandardPorts`

nernden Elemente werden einfach in einer Ebene dargestellt, ungeachtet der eigentlichen darunterliegenden Struktur. Zur Demonstration siehe die Abb. 4.9 mit der Konsolenausgabe des Vergleichs der Erzeugung eines `StandardPorts` und im Gegensatz dazu Abb. 4.10, mit der Vergleichsansicht desselben Vergleichs. Jeweils in rot eingezeichnet sind die korrespondierenden Elemente, sowie die tatsächliche Elementebene (level x). Die `Location` Elemente sollten in der Vergleichsanzeige eine Ebene unter den `DecorationNodes` eingezeichnet sein.

Dieser Umstand wird normalerweise vor dem Benutzer verborgen, der Benutzer sieht also unter Normalbedingungen diese verfeinernden Elemente nicht, dafür sorgen entsprechende Filter. Nur die Makrodifferenz, die alle anderen Differenzen zusammenfasst, wird angezeigt. Die angebotenen Filter können aber ganz einfach deaktiviert werden, entweder einzeln in der Vergleichsansicht oder global in den Grundeinstellungen[18]. Mit deaktivierten Filtern aber - und das stellt das eigentliche Problem dar - kommt die deformierte Struktur zum Vorschein.

[18] Eclipse Menü Window > Preferences > EMF Compare > Filters

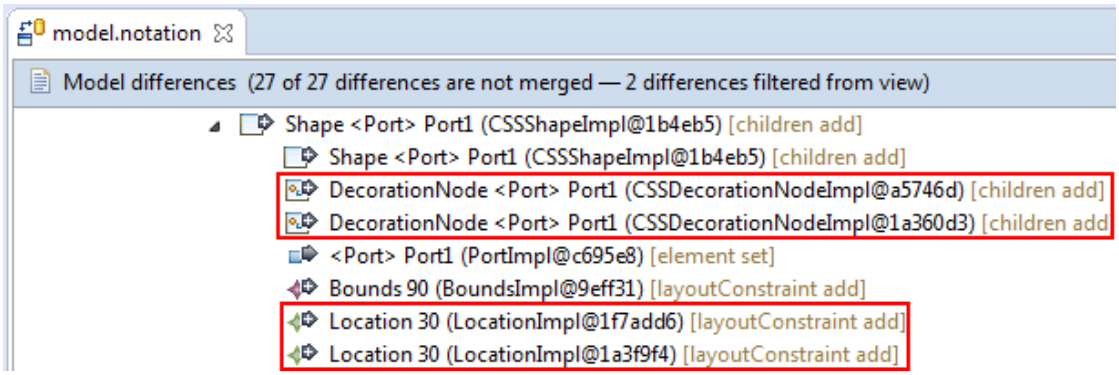


Abb. 4.10: Vergleichsansicht der Differenzstruktur eines `StandardPorts`

Zur Behebung des Problems sollen die zu erstellenden Erweiterungen derart ausgelegt werden, dass die Baumstrukturen von verfeinernden Subdifferenzen eingehalten bzw. entsprechend dargestellt werden, sofern dies mit vertretbarem Aufwand möglich ist.

KAPITEL 5

Verbesserung von SysML-Diagrammvergleichen

Nachdem im vorigen Kapitel die Schritte der Analyse zur Problemerhebung, die Ergebnisse der Analyse, sowie die ausgewählten Problemfälle dargelegt wurden, widmet sich dieses Kapitel den Zielvorstellungen zur Problemlösung, den konkreten Umsetzungen, sowie den Ergebnissen und der Evaluierung.

5.1 Definition der Zielvorstellungen

Wie bereits weiter oben kurz beschrieben, ist das erklärte Ziel dieser Arbeit, die aus der Menge der Probleme ausgewählten Problemfälle zu untersuchen und zu beheben. Dabei ergeben sich zu jedem Problemfall mindestens zwei Zustände im Kontext der Vergleichsansicht und zwar jeweils der Zustand vor der Behebung und der Zustand nach der Behebung des Problems. Die Behebung, bzw. der als korrekte Lösung anzusehende Zustand, ergibt sich aus dem ebenfalls bereits weiter oben definierten pragmatischen Idealzustand einer 1:1 Relation zwischen Diagrammänderung und angezeigter Differenz in der Vergleichsansicht. Der dritte und vierte Problemfall stellen hier eine Ausnahme dar. Zwar gibt es die Möglichkeit die Zustände vor und nach der Behebung zu vergleichen, allerdings weicht die Definition einer Zielvorstellung hier vom definierten pragmatischen Idealzustand ab. Da die Problemfalldefinition zu PF3 jeweils ein Duplikat einer verfeinerten Änderung beschreibt, soll als Idealzustand das Nichtvorhandensein entsprechender Duplikate herangezogen werden. Der Idealzustand für PF4 ist noch einmal abweichend. Hier kann der Soll-Zustand nur unter Zuhilfenahme des unter der Problembeschreibung zu PF1 erwähnten Vorgehens, mit angepasster Hilfsklasse und PostProcessor ermittelt werden. Der jeweilige Ist-Zustand wird dann anhand der Darstellung der Vergleichsansicht ermittelt.

Für die gewählten Problemfälle bedeutet das Folgendes:

- PF1 - Unbenannte UDDs: beim Erstellen bzw. Entfernen von einem Block oder mehreren Blöcken sollen jeweils nur der entsprechende Block bzw. die Blöcke als Änderung in der Vergleichsanzeige aufscheinen, nicht jedoch die automatisch erstellten Kommentarelemente, sowie die diversen Subelemente. Wurde die Anzeige von Stereotypen als Kommentar aktiviert, so soll dieser Umstand entsprechend beachtet werden und die entsprechenden Elemente in der Vergleichsanzeige aufscheinen.
- PF2 - SysML-Ports: beim Erstellen bzw. Entfernen von einem Port oder mehreren Ports, sollen jeweils nur der entsprechende Port bzw. die Ports als Änderung in der Vergleichsanzeige aufscheinen, nicht jedoch die diversen, automatisch erstellten Subelemente. Bei `FlowPorts` soll auch die Zieldefinition von PF1 zutreffen. Für den Fall, dass die betreffenden Elemente vorab explizit sichtbar gemacht wurden, soll das in der Verarbeitung beachtet werden und diese Elemente auch in der Vergleichsanzeige aufscheinen.
- PF3 - Doppelte Anzeige verfeinernder Differenzen: bei der Anzeige von Makrodifferenzen sollen nur die jeweils verfeinernden Differenzen angezeigt und gezählt werden, nicht jedoch die ersetzten Subdifferenzen.
- PF4 - Baumstruktur verfeinernder Differenzen: beim Navigieren in die verfeinernde Baumstruktur einer Makrodifferenz, sollen die Subdifferenzen entsprechend der Objekthierarchie, bzw. der Elementverschachtlung korrekt angezeigt werden, also jeweils nur direkte bzw. gleichrangige Subdifferenzen pro Ebene.

Zur Überprüfung der Ziele der Arbeit ergeben sich in diesem Sinne automatisch passende Bedingungen. Durch Nicht-Aktivierung bzw. Deaktivierung der zu erstellenden Erweiterungen kann jederzeit der Vorher-Zustand erzeugt werden. Durch Aktivierung der jeweiligen Erweiterung kann der Nachher-Zustand erzeugt werden. Als Testdaten sollen, wie bereits beschrieben, die Modelle aus den Repositories der Analysephase wiederverwendet werden. Unter Zuhilfenahme von Screenshots, zur Dokumentation der jeweiligen Zustände, soll die korrekte Funktionsweise demonstriert bzw. überprüft werden.

5.2 Implementierung

Dieser Abschnitt dient der Erläuterung des Vorgehens bei der Implementierung der Erweiterungen, die der Behebung der beschriebenen Problemfälle (PF1-4) dienen sollen. Wie bereits zu Beginn beschrieben, bzw. auch im Analyseteil erwähnt, soll hier nichts Neues erfunden werden. Es soll hingegen durch Erweiterung und Anpassung bestehender, erprobter Funktionalität ein Weg zur Lösung der beschriebenen Problemfälle gefunden werden. Die Eclipse Platform, sowie die bereits beschriebenen, nativen Erweiterungen des EMF Compare Pakets, stellen dazu umfangreiche Möglichkeiten zur Verfügung. Die folgenden Erläuterungen sind, wie auch die vorherigen Abschnitte, anhand der

Problemfälle aufgeteilt in die jeweilig Beschreibung des Vorgehens. Der gesamte Quellcode ist über den Onlinedienst BitBucket verfügbar[1].

5.2.1 PF1: Unbenannte UDDs

Die Implementierungsphase dieser ersten Erweiterung kann nicht klar von der Analysephase, bzw. den beiden Teilen der Analysephase abgegrenzt werden. Einige Erweiterungsteile wurden bereits während der sehr umfangreichen Analyse zu Experimentierzwecken erstellt und danach wiederverwendet bzw. weiterentwickelt.

Der Erweiterung zur Behebung des Problems mit UDDs wurde, wegen der direkten Abhängigkeit zum Papyrus Editor, der Name `org.eclipse.emf.compare.papyrus` gegeben. Da das Problem allein aus dem Diagrammkontext erwächst, wäre eine namentliche Bindung an diesen Kontext von Vorteil gewesen. Eine Erweiterung für Papyrus im Diagrammkontext[2] existiert aber bereits, weshalb die genannte Bezeichnung gewählt wurde.

Die Erweiterung besteht aus zwei Paketen bzw. Plugins, dem oben genannten Hauptplugin steht noch ein `edit` Plugin[3] zur Seite. Wie bereits im Abschnitt zur Analyse der Erweiterungsmöglichkeiten beschrieben, fällt dem Hauptplugin die eigentliche funktionelle Aufgabe der Problembehebung zu, das `edit` Plugin dient eigentlich nur der Anzeige von passenden Bezeichnungen und Icons.

Die Kernaufgabe der Erweiterung ist das Auffinden und Verfeinern von speziellen Differenzen, die durch die automatische Erzeugung von Diagrammelementen entstehen. Eine genaue Beschreibung der Problematik ist in der Detailanalyse zu PF1 gegeben. Durch die eingehende Analyse der Plugins der Diagrammerweiterung, waren die notwendigen Schritte zur Problembehebung und die dafür benötigte Funktionalität bekannt. Die gesuchten Differenzelemente sind Objekte der Klasse `ReferenceChangeImpl` und besitzen `value` Referenzen auf die problematischen Diagrammelemente. Diese Diagrammelemente sind ein Objekt der Klasse `CSSShapeImpl` mit Kennzeichnung `StereotypeComment` im Attribut `type` und ein Objekt der Klasse `CSSConnectorImpl` mit Kennzeichnung `StereotypeCommentLink` im Attribut `type`.

Die Erstellung der Erweiterung gliedert sich grob in vier Schritte:

- Ergänzung des Vergleichsmodells und Generierung des Modellcodes
- Erstellung von `PostProcessor`, `Factories` und `FactoryRegistry`
- Erzeugung und Anpassung des `edit` Plugins
- Erstellung eines `Filters` für die neuen Differenzarten

[1] `https://bitbucket.org/stefanschindler/emfcomparediagramextensions`
[2] `org.eclipse.emf.compare.diagram.papyrus`
[3] `org.eclipse.emf.compare.papyrus.edit`

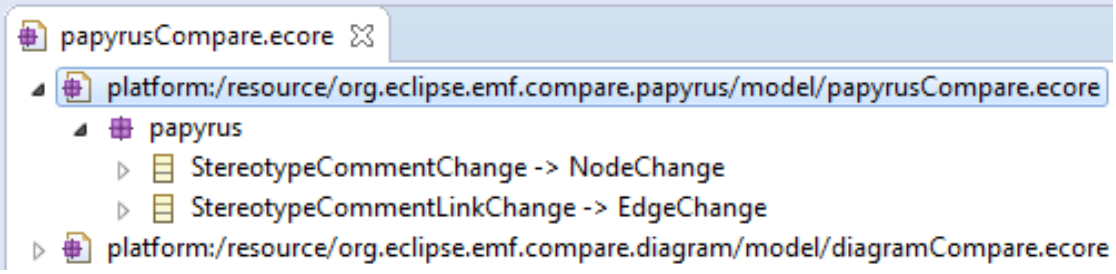


Abb. 5.1: PF1: Ergänzung des EMF Compare Vergleichsmodells

Den erster Schritt bildete die Erweiterung des Vergleichsmodells. Die Diagrammerweiterung ergänzt das Vergleichsmodell vom EMF Compare um Differenzklassen mit Diagrammbezug, siehe Abschnitt 2.5.4. Die hier erstellte Erweiterung bedient sich dieser Modellerweiterung und ergänzt ebenfalls das Vergleichsmodell. Erstellt wurden die zwei Differenzklassen `StereotypeCommentChange` als Subklasse der Diagrammdifferenz `NodeChange` und `StereotypeCommentLinkChange` als Subklasse der Diagrammdifferenz `EdgeChange`, siehe Abb. 5.1. Der entsprechende Quellcode der neuen Differenzklassen wurde anschließend mit dem EMF Generator erstellt. Aus diesen neuen Differenzklassen werden in weiterer Folge Makrodifferenzen erzeugt, die sämtliche zugehörigen Subdifferenzen zusammenfassen.

Im zweiten Schritt wurden die Klassen rund um die Erzeugung von Makrodifferenzen erstellt. Das sind ein eigener `PostProcessor`[4] für die Erweiterung, jeweils eine `Factory` [66] pro neuer Differenzklasse und eine `FactoryRegistry` Klasse.

Ein `PostProcessor` stellt eine einfache Möglichkeit dar, in den Vergleichsprozess einzugreifen. Jeder registrierte `PostProcessor` wird nach allen Teilphasen und ganz am Ende des Vergleichsprozesses über definierte Methoden aufgerufen. Ein `PostProcessor` hat die Aufgabe die Erzeugung von Makrodifferenzen an eine geeignete `Factory` zu delegieren. Eine `Factory` kümmert sich um die Instanzierung und die korrekte Ausformung vom Makrodifferenzen. Eine `FactoryRegistry` wiederum dient der Erstellung von `Factory`-Instanzen und liefert auf Anfrage eine `Map`, die zu jeder angebotenen Differenzklasse eine `Factory` referenziert.

Die diversen Erweiterungen greifen hauptsächlich über die ganz am Ende des Prozesses aufgerufene Methode `postComparison` in den Vergleich ein. Jeder `PostProcessor` besitzt eine Ordnungs- oder Rangzahl[5], die eine Reihenfolge der Aufrufe festlegt. Weiters kann ein `PostProcessor` in der Anwendung auf einen Namensraum[6] oder auf bestimmte Dateiressourcen eingeschränkt werden. Die Definition einer Einschränkung auf einen Namensraum bedeutet, dass der `PostProcessor` nur ausgeführt wird, wenn der Namensraum in den verglichenen Modellen verwendet wird, z.B. der Namesraum

[4] `ExtensionPoint: org.eclipse.emf.compare.rcp.postProcessor`
[5] das nicht-optionale Attribut `ordinal`
[6] engl.: `namespace`

eines UML-Profils. Da der geschilderte Problemfall auftritt, wenn Elemente mit Stereotypen versehen werden, erhielt der erstellte `PostProcessor` keine Einschränkung auf Namensraum oder Ressourcen, er soll immer aufgerufen werden. Die Rangzahl wurde auf 55 festgelegt. Damit wird der `PostProcessor` nach jenem der Diagrammerweiterung (Rangzahl 50) aufgerufen. Das ist notwendig, da das `StereotypeCommentLink` Element Subdifferenzen besitzt, die von der Diagrammerweiterung verfeinert werden.

Der Verfeinerungsprozess läuft ab wie folgt: Zuerst iteriert der `PostProcessor` über alle Differenzen. Jede Differenz wird mit jeder `Factory` auf Übereinstimmung überprüft. Wird eine Übereinstimmung gefunden, so ist die `Factory` für die Erzeugung einer Makrodifferenz zuständig und erhält das betreffende Differenzelement. Die `Factory` erzeugt eine neue Makrodifferenz und kümmert sich um die Referenzierung der verfeinernden Subdifferenzen. Sobald die `Factory` eine neue Instanz einer Makrodifferenz retourniert, fügt der `PostProcessor` das neue Differenzobjekt ins Vergleichsmodell ein. Nach dem Durchlaufen aller Differenzen werden die Abhängigkeiten von anderen Elementen bzw. Anforderungen[7] der neu erstellten Makrodifferenzen überprüft und festgelegt. Damit ist die Erstellung einer neuen Makrodifferenz abgeschlossen.

Nächster Schritt ist die Anzeige in der Vergleichsansicht. Hier kommt das bereits erwähnte `edit` Plugin ins Spiel. Erzeugt wird es, ähnlich dem Modellcode, vom EMF Generator. Nach der Generierung ist das Plugin bereits benutzbar, zeigt aber als Elementbezeichnungen und Symbole nur Platzhalter an. Statt den generierten Klassen wurden eigene erstellt, um Bezeichnungen und Symbole bestimmen zu können. Dazu wurde einmal die Sublasse `PapyrusCompareItemProviderAdapterFactorySpec` von der Klasse `DiagramCompareItemProviderAdapterFactorySpec`[8] abgeleitet. Diese `Factory` wurde als `Extension`[9] registriert und erzeugt den von der Klasse `ViewItemProviderSpec`[10] abgeleiteten und angepassten `PapyrusViewItemProviderSpec`. Dieser spezielle `ItemProvider` sorgt dafür, dass bei Diagrammelementen ohne referenziertes Modellelement statt „<null>“ die Typbezeichnung des Diagrammelements ausgegeben wird, also z.B. `<StereotypeComment>`. Eine weitere Factory, nämlich die Subklasse `PapyrusItemProviderAdapterFactorySpec`, wurde abgeleitet von der generierten Superklasse `PapyrusItemProviderAdapterFactorySpec`, und ebenfalls als `Extension`[11] registriert. Aufgabe dieser Factory ist die Erzeugung von speziellen `ItemProvider` Objekten für die beiden neu erstellten Differenzklassen `StereotypeCommentChange` und `StereotypeCommentLinkChange`. Diese speziellen `ItemProvider` haben die Aufgabe, abhängig von der Sichtbarkeit der Differenzelemente, entsprechende Symbole und Bezeichnungszusätze anzuzeigen. Denn wie in der Detailanalyse zu PF1 ausführlich dargelegt, sind die beschriebenen Diagrammelemente

[7] engl.: `requirements`

[8] `org.eclipse.emf.compare.diagram.internal.matchs.provider.spec.DiagramCompareItemProviderAdapterFactorySpec`

[9] `ExtensionPoint: org.eclipse.emf.compare.edit.adapterFactory`

[10] `org.eclipse.emf.compare.diagram.internal.matchs.provider.spec.ViewItemProviderSpec`

[11] `ExtensionPoint: org.eclipse.emf.edit.itemProviderAdapterFactories`

üblicherweise unsichtbar. Wird jedoch die entsprechende Filteroption deaktiviert, sollen die Differenzelemente in der Vergleichsansicht angezeigt werden. In diesem Fall wird mit einem speziellen Symbol, sowie dem Bezeichnungszusatz „Invisible“ auf die tatsächliche Unsichtbarkeit im Diagramm hingewiesen. Wird jedoch die Anzeigeoptionen für Stereotype manuell geändert, werden der Kommentar, sowie der Verbinder im Diagramm sichtbar und die Differenzelemente werden in der Vergleichsansicht nicht ausgefiltert. Der jeweilige `ItemProvider` zeigt daraufhin das Standardsymbol des Diagrammelements an und der Bezeichnungszusatz entfällt.

Letzter Punkt ist der erstellte `Filter`[12], `StereotypeCommentFilter`. Ein `Filter` hat kein Namensraum Attribut, um zu bestimmen, ob er aufgerufen werden soll. Es besitzt jedoch jeder `Filter` die Methode `isEnabled`, die vor dem eigentlichen Aufruf der Methode `apply` prüft, ob der `Filter` zur Anwendung kommen soll. Da der PF1 sämtliche Papyrus Diagramme betreffen kann, macht eine Einschränkung hier keinen Sinn, der `Filter` kommt immer zur Anwendung. Gibt ein `Filter` den Wert `true` zurück, wird das überprüfte Element von der Anzeige ausgeschlossen. Ein `Filter` erhält bei einem Aufruf kein Differenzelement sondern einen Baumknoten[13]. Als Nutzdaten enthält der Baumknoten dann ein Differenzelement. Da ein Filter auf jedes, zur Anzeige bestimmte Element angewendet wird, muss zuerst der Differenztyp bestimmt werden. Im konkreten Fall untersucht der Filter nur Differenzelemente der beiden neu erstellten Differenzklassen, siehe oben. Anschließend wird überprüft, ob das referenzierte Diagrammelement sichtbar ist oder nicht. Sind die Elemente nicht sichtbar, was üblicherweise der Fall ist, so werden die Differenzelemente ausgefiltert. Wurden die betreffenden Elemente jedoch, wie in der Detailanalyse zu PF1 beschrieben, per Einstellung sichtbar gemacht, werden die Differenzelemente angezeigt.

Damit ist die Umsetzung der Vorgaben für PF1 abgeschlossen. Eine Gegenüberstellung von Vorgaben und Ergebnissen wird im nächsten Kapitel durchgeführt.

5.2.2 PF2: SysML Ports

Die Implementierungsphase der Erweiterung für diesen Problemfall wurde mit dem Wissen und den Erfahrungen aus der Analysephase und dem ersten Implementierungsschritt durchgeführt. Da es sich im Prinzip um das selbe Vorgehen wie in der zuvor beschriebenen Implementierung zu PF1 handelt, werden die Schritte bzw. die eingesetzten Mittel hier nicht mehr ganz so ausführlich beschrieben. Vielmehr werden nur die Unterschiede der Umsetzung in Vergleich zu PF1 erläutert. Es wurden die selben `ExtensionPoints` wie für PF1 verwendet.

Zur Ergänzung des Vergleichsmodells wurden drei zusätzliche Klassen erzeugt, `SysML-DiagrammDiff` als Subklasse von `DiagrammDiff`, sowie `PortChange` und `Auto-Change` als Subklassen von `SysMLDiagrammDiff`, siehe Abb. 5.2. Die Differenzklasse

[12] `ExtensionPoint: org.eclipse.emf.compare.rcp.ui.filters`
[13] `org.eclipse.emf.edit.tree.TreeNode`

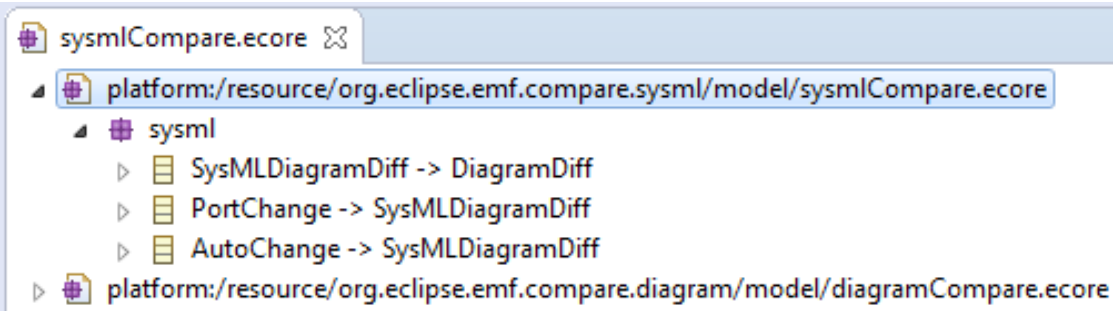


Abb. 5.2: PF2: Ergänzung des EMF Compare Vergleichsmodells

`PortChange` dient der Zusammenfassung von Portdifferenzen, die Differenzklasse `AutoChange` dient der Zusammenfassung der von Papyrus automatisch erzeugten Differenzelemente. Die Erzeugung des Modellcodes verlief analog zum bereits beschriebenen Vorgehen, mit einer Ausnahme. Da diesmal nicht direkt von konkreten `DiagrammDiffs` geerbt wird, wie zuvor von `NodeChange` und `EdgeChange`, war ein manueller Eingriff notwendig. Die Implementierungsklasse `DiffImpl`[14] setzt nicht alle Vorgaben des Interface `Diff`[15] um, sondern überlässt das spezifischen Subklassen, wie etwa `NodeChangeImpl`. Stattdessen wird von einigen Methoden eine `UnsupportedOperationException`[16] geworfen. Diese Methoden werden jedoch im Verfeinerungsprozess benötigt. Ein manuelles Einfügen der Methoden in eine generierte Superklasse wäre zwar möglich gewesen, stellt aber keine schöne Lösung dar. Weit besser wäre es, wenn die generierten Klassen von einer manuell erstellten Klasse erben könnten. Mit den puren EMF Mitteln ist das nicht möglich. Es existiert jedoch eine EMF Erweiterung namens `EMF-Loophole`[17], die den `Generation-Gap-Pattern`[18] umsetzt. Damit ist es möglich, generierte Klassen von manuell erstellten Klassen erben zu lassen und somit sehr einfach um Funktionalität zu erweitern. Im speziellen Fall wird eine Klasse eingefügt, von der die Implementaionsklassen `PortChangeImpl` und `AutoChangeImpl` die benötigten, aber von `DiffImpl` nicht angebotenen Methoden erben.

Der erstellte `PostProcessor` funktioniert analog zu dem aus PF1. Ursprünglich wurde der `PostProcessor` auf den SysML Namensraum eingeschränkt. Diese Einschränkung war allerdings unter gewissen Umständen kontraproduktiv. Werden z.B. die entsprechenden `Compartments` eines Blocks sichtbar gemacht[19], erstellt Papyrus ebenfalls Elemente, die als `AutoChange` in Frage kommen würden. Da diese Änderungen allerdings nur GMF Elemente im Diagramm betreffen, kommen keine Elemente aus dem SysML Namensraum zum Einsatz und der `PostProcessor` wird nicht benutzt. Aus

[14] `org.eclipse.emf.compare.impl.DiffImpl`
[15] `org.eclipse.emf.compare.Diff`
[16] `java.lang.UnsupportedOperationException`
[17] `https://github.com/mbarbero/emf-loophole`
[18] `http://heikobehrens.net/2009/04/23/generation-gap-pattern`
[19] siehe Detailanalyse zu PF2

diesem Grund wurde die Einschränkung des Namensraums für den `PostProcessor` von SysML auf GMF-Diagramme geändert.

Das `edit` Plugin wurde analog zu PF1 aus den ergänzten Differenzmodell erzeugt. Auf die mögliche Sichtbarkeit bzw. Unsichtbarkeit diverser Elemente wurde hier ebenfalls Rücksicht genommen. Eigentlich unsichtbare Elemente, die nur durch Deaktivierung des entsprechenden Filters zur Anzeige kommen, werden mit einem entsprechenden Icon und dem Zusatz „Invisible“ versehen.

Der für diese Erweiterung erstellte Filter `SysMLAutoChangeFilter` funktioniert ähnlich dem Filter aus PF1. Überprüft wird hier, ob ein Differenzelement von Typ `AutoChange` ist, wenn ja, wird es ausgefiltert. Im Unterschied zum Filter aus PF1 wird hier in der Methode `isEnabled` überprüft, ob der Filter zur Anwendung kommen soll. Dies soll nur erfolgen, wenn Differenzelemente aus den Namensräumen für SysML oder Diagramme an dem Vergleich beteiligt sind, zur Begündung siehe die Erläuterungen zum `PostProcessor` für PF2 weiter oben.

5.2.3 PF3: Doppelte Anzeige verfeinerter Differenzen

Die Implementierungsphase der Erweiterung für diesen Problemfall überschneidet sich mit den Implementierungen für PF1 und PF2. Nachdem das Problem erkannt war, wurde eine umfangreiche Analyse durchgeführt. Bereits während der Analysephase wurde mit Lösungsansätzen experimentiert. Ein `Filter`, ähnlich den Filtern für PF1 und PF2 erschien als Lösung ungeeignet, da die Verdoppelung der Differenzen damit erst nachträglich entfernt worden wäre. Das würde doppelten Aufwand bedeuten, einmal für die Erstellung der Differenzen in der Vergleichsanzeige und danach nochmals beim Ausfiltern. Das Entfernen der doppelten Differenzen aus dem Vergleichsmodell wurde ebenfalls erwogen, jedoch wegen möglichen Nebeneffekten bzw. Abhängigkeiten zu oder von anderen Differenzelementen wieder verworfen. Schlussendlich wurde nach der Analyse der Baumerstellung für die Vergleichsanzeige der Weg über die Erstellung einer Erweiterung in Form einer neuen Gruppierungsoption[20] gewählt. Es wurde kein separates Plugin dafür erzeugt, die Erweiterung wurde stattdessen in das Hauptplugin für PF1 integriert.

Als Vorlage wurde die Gruppenoption „By Side“ gewählt, die normalerweise automatisch voreingestellt ist. Erstellt wurden zwei Klassen, zuerst die Klasse `PapyrusBasicDifferenceGroupImpl` als Subklasse von `BasicDifferenceGroupImpl`[21]. Die Subklasse enthält, außer den vorgeschriebenen Konstruktoren nur eine Methode, `handleRefiningDiffs`, diese überschreibt die gleichnamige Methode der Superklasse. Diese Methode iteriert rekursiv über alle verfeinernden Subdifferenzen einer Makrodifferenz und erzeugt daraus Knoten für die Baumansicht. Einzige Änderungen an der Methode sind zwei bedingte Anweisungen, die ein Element unter gewissen Umständen überspringen und somit von der Aufnahme in den Baum ausschließen. Ausgeschlossen werden Subdiffe-

[20] `ExtensionPoint: org.eclipse.emf.compare.rcp.ui.groups`

[21] `org.eclipse.emf.compare.rcp.ui.internal.structuremergeviewer.groups.impl.BasicDifferenceGroupImpl`

renzen, die dasselbe Diagrammelement referenzieren wie die Makrodifferenz. Die zweite erstellte Klasse ist `PapyrusThreeWayComparisonGroupProvider` als Subklasse von `ThreeWayComparisonGroupProvider`[22]. Diese Klasse stellt die eigentliche Erweiterung für den `ExtensionPoint` von Gruppierungsoptionen dar. Die Klasse benutzt zur Baumerstellung die Klasse `PapyrusBasicDifferenceGroupImpl` und somit die eben beschriebene Funktionalität.

5.2.4 PF4: Baumstruktur verfeinernder Differenzen

Die Implementierungsphase dieses Problemfalls kann, ebensowenig wie die vorhergehende, von den Implementierungsphasen für PF1 und PF2 getrennt werden. Auch die Übergänge zwischen Analysephase und Implementierungsphase verschwammen, da immer wieder Probleme und Fehler auftraten, die weitere Analysen bedingten. Erstellt wurde schlussendlich eine Methode zur Zuweisung der verfeinernden Subdifferenzen einer Makrodifferenz. Die Methode durchläuft rekursiv alle Ebenen, die sich anhand des Vergleichsmodells aus Matches und Subdifferenzen ergeben. Dabei weist sie den Differenzelementen ihre Subdifferenzen entsprechend der Elementhierarchie zu. Somit wird ein Baum aus Subdifferenzen aufgebaut, der das darunterliegende Objektmodell wiedergibt. Die Umsetzung stellt keine eigene Erweiterung im Sinne eines Plugins dar, sondern kommt bei der Erzeugung der neu erstellten Differenzelemente aus PF1 und PF2 zum Einsatz und zwar in der jeweiligen `Factory`.

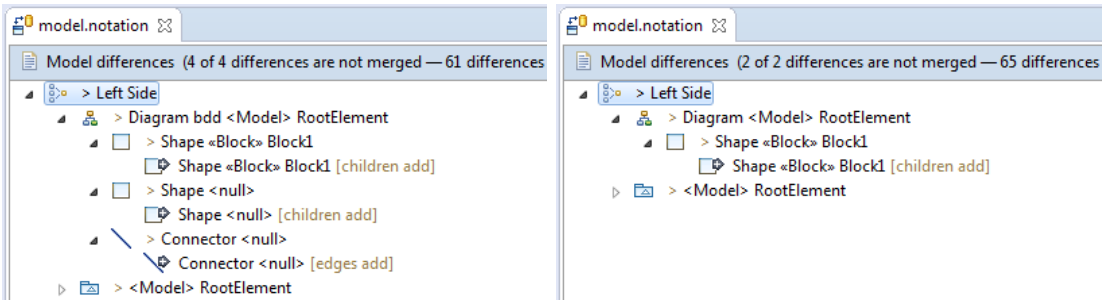
5.3 Ergebnisse und Evaluierung

Hier sollen kurz die Ergebnisse der Implementierungsphase dargestellt werden. Zu jedem beschriebenen Problemfall wurde eine Zielvorstellung definiert, und gegen diese Soll-Zustände werden die Ist-Zustände der Ergebnisse evaluiert. Zur Gegenüberstellung von Problem und Lösungsversuch sind Abbildungen eingefügt, jeweils eine Abbildung ohne Erweiterung (links) für den Vorher-Zustand und eine mit den entsprechenden Erweiterung (rechts) für den Nachher-Zustand. Die Diskussion der Ergebnisse erfolgt dann im nächsten Kapitel.

5.3.1 PF1: Unbenannte UDDs

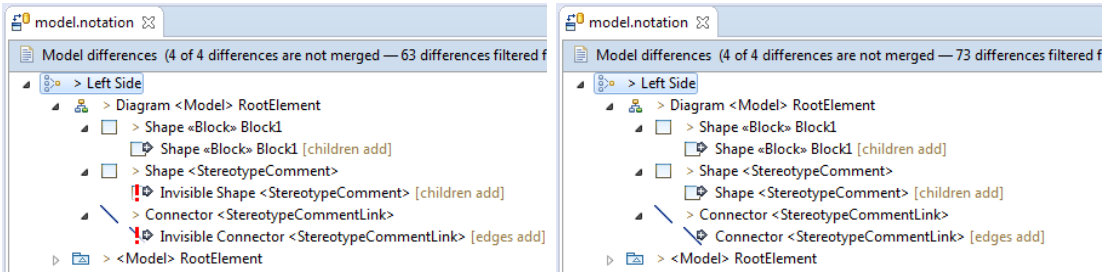
Wie aus den beiden Analysen zu PF1 und PF2 hervorging, ist dieses Problem nicht auf einzelne Elemente oder Sprachen reduzierbar, sondern tritt immer dann auf, wenn Diagrammelementen Stereotypen zugeordnet werden. Zur Problembehebung wurde eine Erweiterung implementiert, die einerseits betreffenden Elemente zu einer Makrodifferenz zusammenfasst, sowie andererseits entsprechende Bezeichnungen ausgibt. Bei Bedarf werden die Differenzelemente in der Vergleichsanzeige entfernt. Die Abb. 5.3a stellt den

[22] `org.eclipse.emf.compare.rcp.ui.internal.structuremergeviewer.groups.impl.ThreeWayComparison-GroupProvider`



(a) Vorher-Zustand (b) Nachher-Zustand

Abb. 5.3: PF1: Vorher/Nachher-Zustandsgegenüberstellung



(a) Filter deaktiviert (b) Stereotypkommentare aktiviert

Abb. 5.4: PF1: Nachher-Zustandsvariationen

Vorher-Zustand ohne Erweiterung dar. Die Abb. 5.3b stellt den Nachher-Zustand mit Erweiterung und aktivem Filter dar.

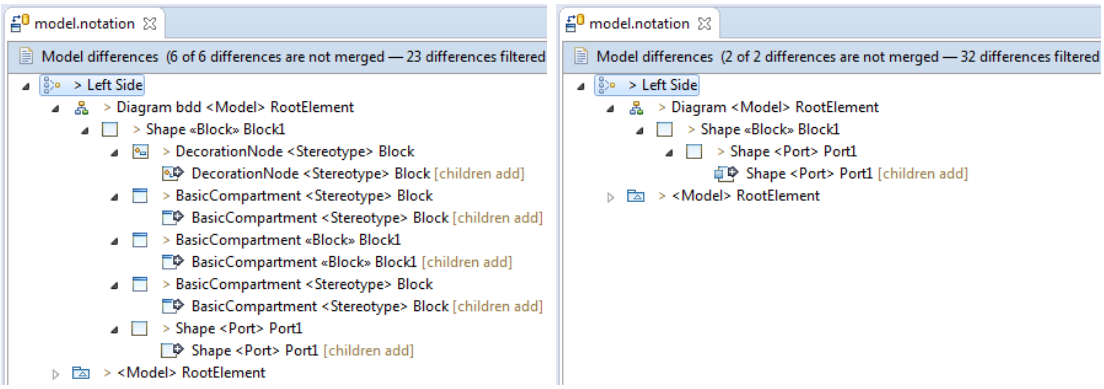
Die Abb. 5.4a stellt den Nacher-Zustand mit deaktiviertem Filter dar. Die Abb. 5.4b stellt den Nachher-Zustand mit aktivierter Option zur Anzeige von Stereotypen als Kommentar dar.

5.3.2 PF2: SysML Ports

Dieses Problem teilt sich in den Fall für `StandardPorts` und den Fall für `FlowPorts` auf. Es wurde eine Erweiterung implementiert, die erkennt, ob Elemente sichtbar sind und nur bei Bedarf die unsichtbaren Elemente in Markodifferenzen kapselt. Diese Makrodifferenzen können dann bei Bedarf per Filter ausgeblendet werden. Bei `FlowPorts` tritt zusätzlich zur Erweiterung für PF2 die Erweiterung für PF1 in Aktion. Die Abb. 5.5a stellt den Vorher-Zustand für `StandardPorts` ohne Erweiterung dar. Die Abb. 5.5b stellt den Nachher-Zustand mit Erweiterung und aktivem Filter dar.

Die Abb. 5.6 stellt den Nachher-Zustand mit deaktiviertem Filter dar.

Die Abb. 5.7a stellt den Vorher-Zustand für `FlowPorts` ohne Erweiterung dar. Die Abb. 5.7b stellt den Nachher-Zustand mit Erweiterung und aktivem Filter dar.



(a) Vorher-Zustand (b) Nachher-Zustand

Abb. 5.5: PF2: (`StandardPorts`): Vorher/Nachher-Zustandsgegenüberstellung

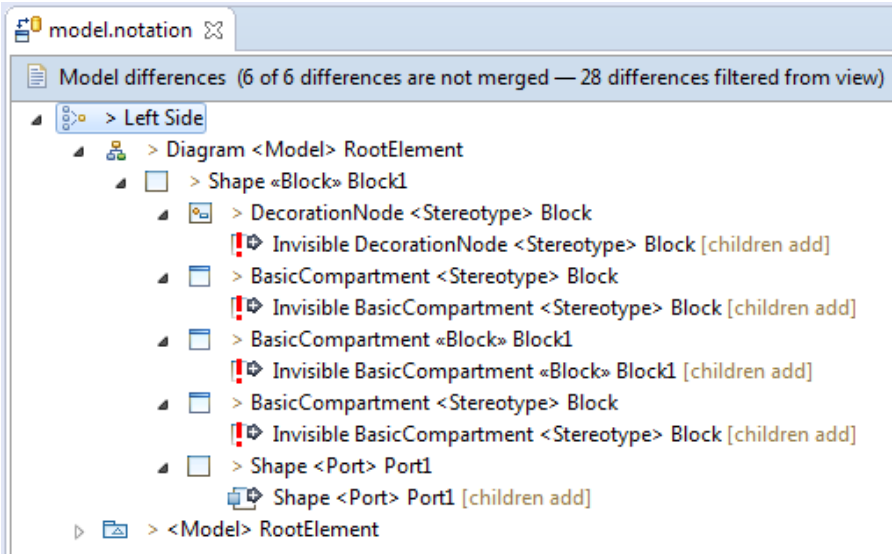
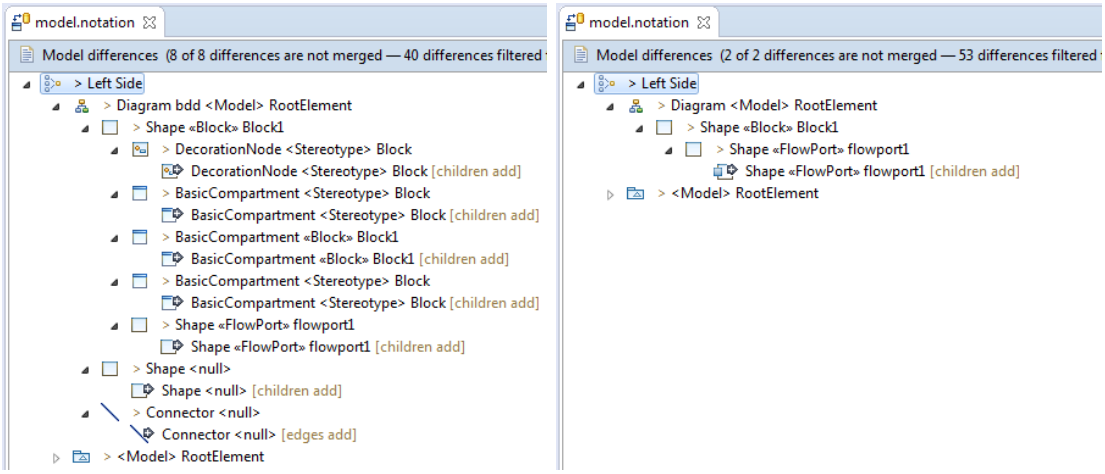


Abb. 5.6: PF2: (`StandardPorts`): Nachher-Zustandsvariation (Filter deaktiviert)



(a) Vorher-Zustand (b) Nachher-Zustand

Abb. 5.7: PF2: (`FlowPorts`): Vorher/Nachher-Zustandsgegenüberstellung

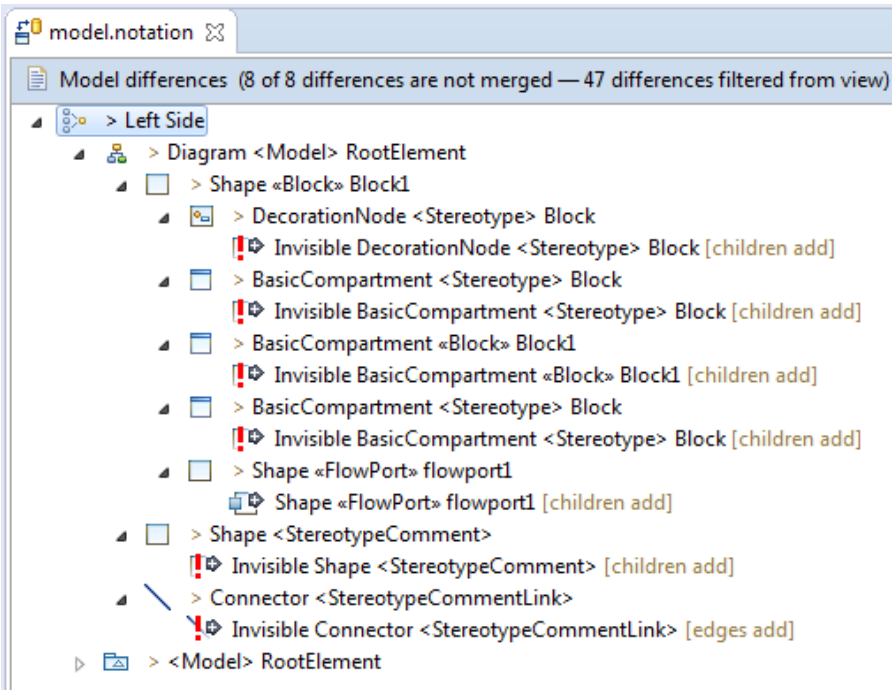


Abb. 5.8: PF2: (`FlowPorts`): Nachher-Zustandsvariation (Filter deaktiviert)

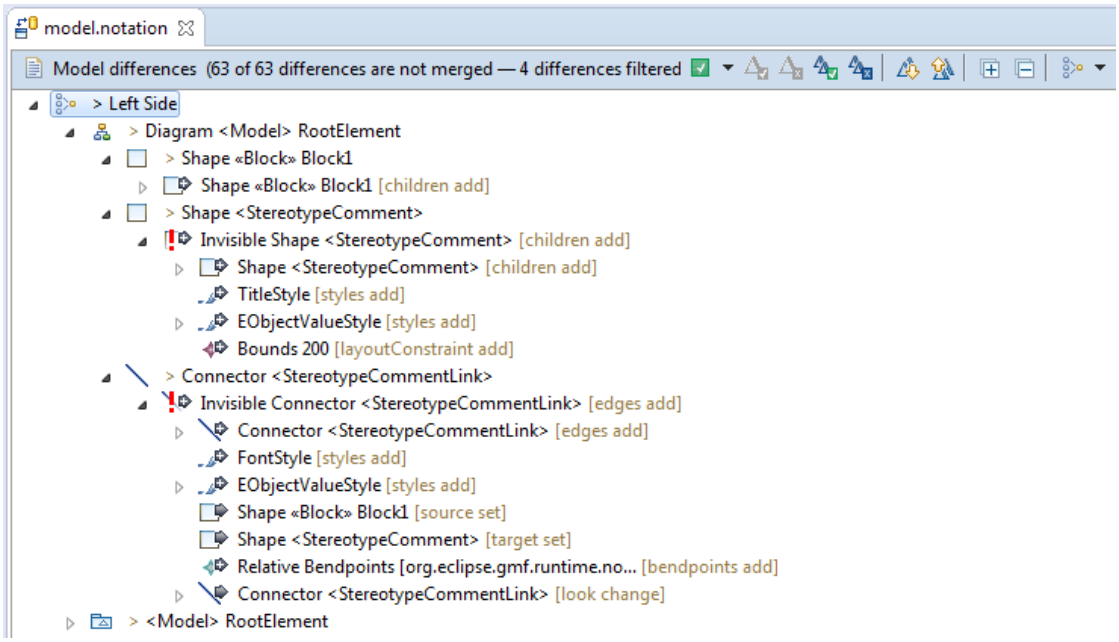


Abb. 5.9: PF3: Vorher-Zustand, mit doppelten Differenzen

Die Abb. 5.8 stellt den Nacher-Zustand mit deaktivierten Filtern dar, also ohne `SysML-AutoChange` Filter und ohne `StereotypeComment` Filter.

5.3.3 PF3: Doppelte Anzeige verfeinerter Differenzen

Dieses Problem zieht sich durch alle anderen Probleme. Immer dann wenn eine Differenz verfeinert wird, kommt es durch die Anzeige der primär verfeinernden Subdifferenz zu einer Verdoppelung der Differenz. Es wurde eine Erweiterung implementiert, die eine neue Gruppierungsoption bietet. Damit werden redundante Differenzen ausgeblendet. Diese Erweiterung ist enthalten in der Erweiterung für PF1. Zum Einsatz kommt die Gruppierungsoption erst durch manuelles Auswählen der Option in der Vergleichsanzeige. In den folgenden Abbildungen wird die Funktionalität an einem Beispiel aus dem bereits bekannten SysML Kontext demonstriert. Für die Abbildungen wurden der `Diagram Refined elements` und der `StereotypeComment` Filter deaktiviert. Die Abb. 5.9 stellt den Vorher-Zustand, also noch mit verdoppelten Differenzen dar. Die Abb. 5.10 stellt den Nachher-Zustand, also ohne doppelte Differenzen dar.

5.3.4 PF4: Baumstruktur verfeinernder Differenzen

Dieses Problem tritt, ähnlich dem PF3, bei nahezu jedem Vergleich auf. Zur Problemlösung wurde keine eigene Erweiterung implementiert, sondern das Verhalten der Erweiterungen für PF1 und PF2 entsprechend angepasst. Die implementierte Methode zur Behebung des Problems wird bei jeder Erzeugung einer Makrodifferenz aus einer der erstellten

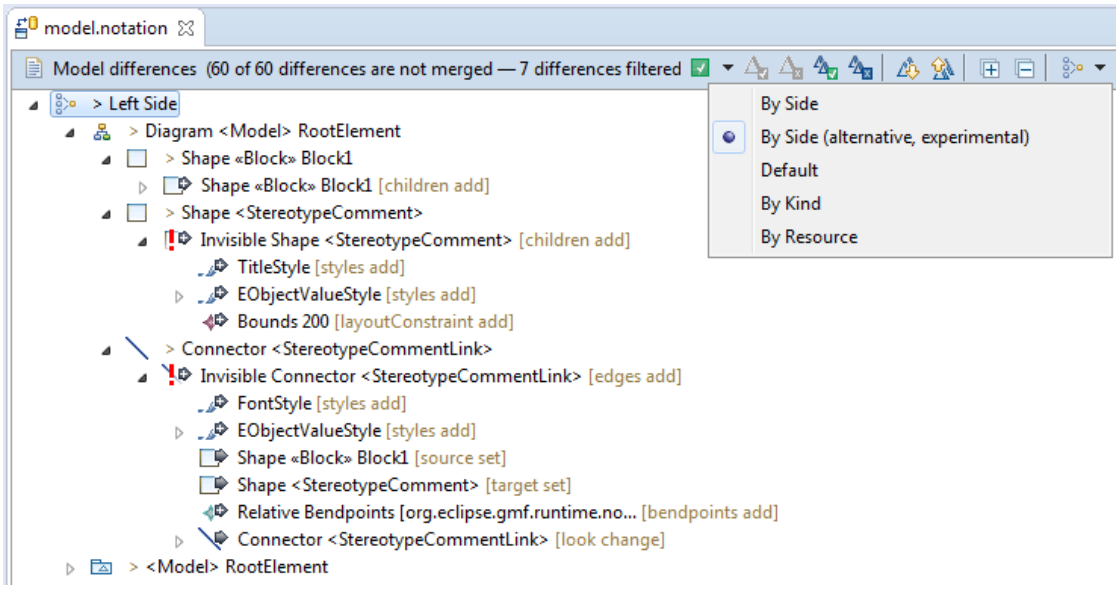


Abb. 5.10: PF3: Nachher-Zustand, ohne doppelte Differenzen

Erweiterungen aufgerufen. Eine Möglichkeit zur Deaktivierung per UI, wie etwa bei Filtern oder `PostProcessoren`, ist nicht gegeben. Um daher eine Abbildung zur Demonstration des Vorher-Zustands erzeugen zu können, musste die Methode in der jeweiligen `Factory` auskommentiert werden. Weiters wurden bei der Erstellung der Abbildungen der `Diagram Refined elements` Filter und der `StereotypeComment` Filter deaktiviert, sowie die alternative Gruppierungsoptions für PF3 aktiviert. Die Abb. 5.11 stellt den Vorher-Zustand dar, also ohne korrigierter Differenzstruktur. Die Abb. 5.12 stelle den Nachher-Zustand dar. Hier werden die Strukturen der Subdifferenzen, die vom Vergleichsmodell vorgegeben werden, bei der Erstellung der Makrodifferenzen beachtet und bei der Anzeige eingehalten.

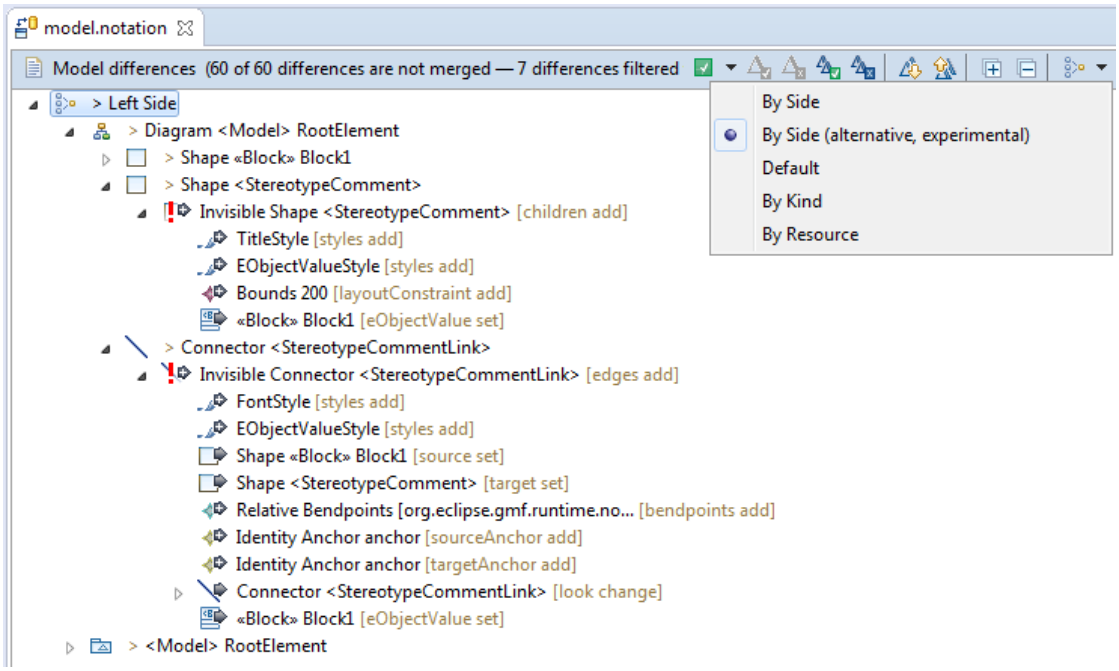


Abb. 5.11: PF4: Vorher-Zustand, unveränderte Differenzstruktur

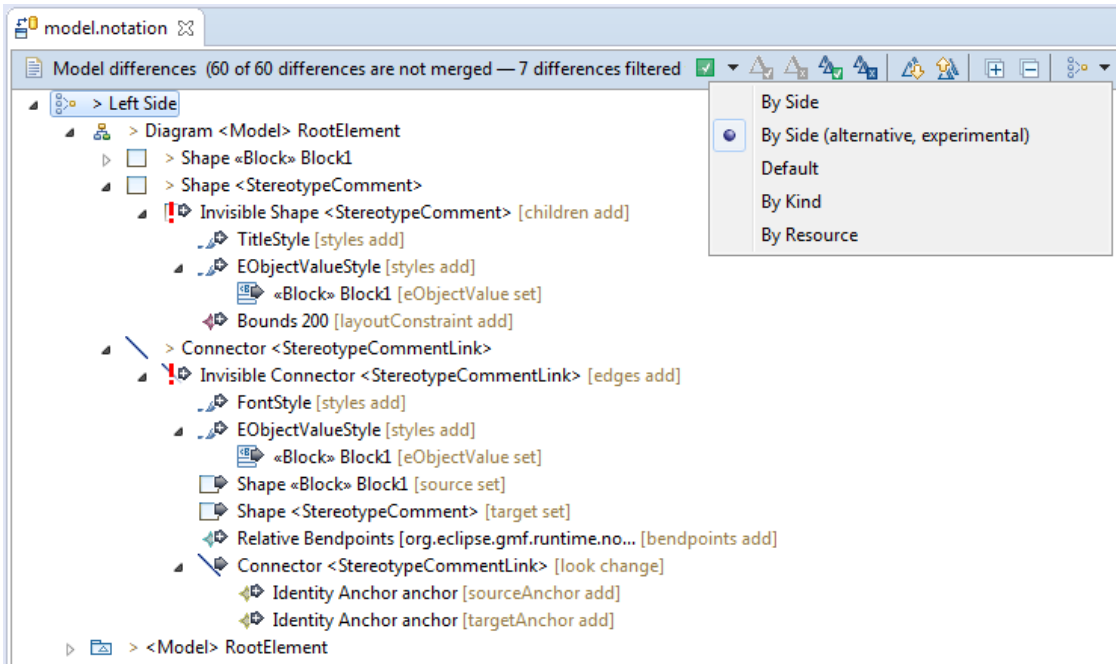


Abb. 5.12: PF4: Nachher-Zustand mit korrigierter Differenzstruktur

KAPITEL 6

Diskussion

Dieser Abschnitt soll der kritischen Betrachtung der gesamten Arbeit dienen, hauptsächlich jedoch die Ergebnisse der Analyse, sowie der Ausarbeitung mit den erklärten Erwartungen abgleichen. Der vorige Abschnitt hat bereits die Soll-Vorgaben mit den Ist-Zuständen der Vergleichsansicht abgeglichen. Die Abbildungen belegen weitestgehend die Erfüllung der erklärten Ziele.

Der Name der Arbeit deutet eine Spezialisierung auf SysML an, jedoch beschäftigt sich ein Gutteil der Arbeit mit allgemeinen, also nicht SysML spezifischen Problemen. Dazu muss folgendes in die Betrachtung miteinbezogen werden. Zu Beginn der Arbeit, nachdem die Analysen zu Problemen in SysML Diagrammen erstellt wurden, war die tatsächliche Natur einzelner ausgewählter Problemfälle nicht absehbar. Erst durch die Detailanalysen der gewählten Problemfälle wurde das spezifische Ausmaß, im speziellen von PF1, zur Gänze erkennbar. Der zweite ausgewählte Problemfall, SysML Ports, steht jedenfalls ganz im Zeichen des Titels der Arbeit. Eine generelle Anwendbarkeit ist hier nicht gegeben, im Gegenteil, die erstellte Erweiterung besitzt eine klare Ausrichtung auf SysML. Die für PF1 erstellte Erweiterung kann allerdings, da sie ein Problem behebt, das in untrennbarem Zusammenhang mit der Zuweisung von Stereotypen steht, in den Kontext von DSLs gerückt werden, indem sich SysML ebenfalls befindet. Die Problemfälle 3 und 4, die ebenfalls dem allgemeinen EMF Compare Kontext zuzuschreiben sind, haben sich erst im Laufe der Ausarbeitung ergeben, wurden also nicht aktiv aus der Problemmenge der Analyse ausgesucht. Die Bearbeitung hatte hier hauptsächlich pragmatisch technische Gründe, aber nicht zuletzt auch mit Interesse an der Umsetzung zu tun.

Ein weiterer Punkt in Bezug auf den Titel der Arbeit ist durch den Bearbeitungsfokus gegeben. Der Titel gibt Modellversionierung als Themenbereich vor, der Fokus liegt aber klar auf der Darstellung von Diagrammdifferenzen und damit auf einem Teilaspekt der Modellversionierung. Dieser Umstand ist, ebenso wie die oben beschriebene Abweichung von der SysML Spezialisierung, mit der unklaren Problemlage der Analysephase zu erklären. Zwar wurden anfangs exemplarisch weiterführende Analysen, z.B.

zur Modellzusammenführung durchgeführt - von hauptsächlichem Interesse war hier die Zusammenführung im Konfliktfall - wobei auch schnell Probleme gefunden wurden. Diese Probleme wurden aber, da sich alleine aus der Analyse der Modellvergleiche schon mehr als genug potentielle Problemstellungen ergaben, nicht weiter verfolgt und deshalb auch nicht in die Arbeit übernommen. Konkret bedeuten diese Überlegungen, dass die Arbeit zwar Modellversionierung im Titel trägt und sich auch mit Versionierung und ihren Teilaspekten beschäftigt, die gewählten und bearbeiteten Problemfälle jedoch eigentlich nur Probleme der Differenzdarstellung addressieren.

Mit Blick auf die Analyse zur Problemerhebung bei Modellvergleichen kann gesagt werden, dass die Auswahl der Operationen bzw. Änderungen oft nicht unbedingt ideal waren. Das Wissen rund um interessante Details im Zuge von Änderungen, z.B. an Blöcken oder Ports, war jedoch zum Zeitpunkt der Erstellung der Analyse noch nicht gegeben. Dieses Wissen wurde erst durch die Detailanalysen und Experimente vertieft. Analysen anhand von Änderungen, die im jeweiligen Elementkontext als wichtig erachtet wurden, z.B. Sichtbarmachung von Stereotypkommentaren oder `Compartments`, wurden dann während der Detailanalysen bzw. der Implementierungsphase nachträglich erstellt. Zusammenfassend kann zu diesem Punkt gesagt werden, dass die Analysen mit dem gewonnenen Wissen der Ausarbeitung in einigen Punkten anders aussehen würden.

Ein unschönes Detail findet sich in der Anzeige der Summe der gefilterten Differenzen. Für die Erfüllung von PF3 wurde eine eigene Option zur Differenzgruppierung erstellt, welche die primäre Subdifferenz einer Makrodifferenz ausblendet. Diese ausgeblendete Subdifferenz wird aber dennoch zur Gesamtzahl der Differenzen gezählt. Wurde nun ein `ReferenceChanceImpl` Objekt bei der Verfeinerung durch eine Erweiterung als primäre Subdifferenz einer Makrodifferenz referenziert, erhöht sich die Gesamtzahl der Differenzen um eins, nämlich durch die Makrodifferenz. Wurde nun in weiterer Folge diese Makrodifferenz wiederum in einer übergeordneten Makrodifferenz zusammengefasst, erhöht sich die Gesamtsummer nochmals um eins, insgesamt also um zwei. Zusammengefasst bedeutet dieser Umstand, dass durch die Erstellung von Makrodifferenzen immer auch die Gesamtzahl der Differenzen steigt. Das Entfernen von Makrodifferenzen aus dem Vergleichsmodell kam wegen eventuellen Abhängigkeiten bzw. möglichen Nebeneffekten vorerst nicht in Frage, diese Option müsste erst eingehend geprüft werden. Eine Möglichkeit zur Erweiterung der Vergleichsansicht per `ExtensionPoint` ist nicht gegeben. Die Anforderung bzw. Korrektur könnte somit nur direkt im entsprechenden Projekt umgesetzt werden. Die Korrektur der Zählweise auf Basis der bestehenden Vergleichsansicht, wurde nach Abwägung der Aufwände vorerst verworfen, wäre jedoch als Option für eine Weiterentwicklung denkbar. Noch eine Option wäre denkbar und zwar könnten Makrodifferenzen anders als bisher angezeigt werden, ohne Referenz auf das Diagrammelement, also ohne das Diagrammelement der primären Subdifferenz zu verwenden. Damit würde in der Darstellung keine Verdoppelung mehr vorkommen und die Zählweise als eigene Differenz wäre begründet. Diese Option müsste einer eingehenden Analyse unterzogen werden und verbleibt damit vorerst ebenfalls als Option zur Weiterentwicklung.

Weiters kann angemerkt werden, dass die Wahl des Lösungswegs für PF3 über eine neue Gruppierungsoption nur einen der möglichen Wege darstellt und dabei sicher nicht den besten oder elegantesten. Gruppierungsoptionen haben eigentlich den Sinn, spezielle Ansichten für den Benutzer anzubieten, damit z.B. im Konfliktfall die Änderungen der beteiligten Modellversionen getrennt betrachtet werden können. Der Lösungsweg über Gruppierungsoptionen ist also in der Hinsicht für einen Benutzer nicht von Vorteil, eher das Gegenteil ist der Fall. Der Weg über eine Erweiterung für den `ExtensionPoint differenceGroupExtender`[1] wäre eleganter und auch mit dem Benutzerzweck im Fokus die bessere Wahl gewesen. Allerdings ist zu erwähnen, dass durch die gewählte Form der Erweiterung jederzeit die Möglichkeit bestand, die Erweiterung per Auswahl der Gruppenoption zu aktivieren bzw. durch Abwahl zu deaktivieren. Dadurch war eine einfache und praktische Möglichkeit gegeben die Erweiterung in allen getesteten Vergleichssituationen zuzuschalten und damit ebenfalls zu testen. Wohingegen diese Möglichkeit bei der anderen Erweiterungsform nicht bestanden hätte. Der gewählte Lösungsansatz bot einen minimalinvasiven und praktischen Weg zur prototypischen und exemplarischen Lösung an und brachte auch noch den Vorteil mit sich, direkt mit Differenzen arbeiten zu können. Der erwähnte `ExtensionPoint` arbeitet mit `TreeNodes`, was im direkten Vergleich ungleich aufwändiger gewesen wäre.

Die Erzeugung von eigene Erweiterungen in Form von Plugins dient dieser Arbeit vorwiegend als Möglichkeit zur Abgrenzung zu bestehender Funktionalität, ein weiterer Grund war schlicht der Erfahrungsgewinn. Die Erweiterung für PF1 wäre wohl eher dem Kontext der bestehenden EMF Compare Erweiterung für Papyrus Diagramme zuzuordnen, zumindest was eine Zuordnung nach funktionalen Gesichtspunkten betrifft. Die Erweiterungen für PF3 und PF4 wurden in die Erweiterung für PF1 integriert, hauptsächlich aus praktischen Gründen. Die Erweiterung für PF3 wäre kontextuell wohl eher dem EMF Compare Projekt[2] zuzuordnen. Der Ansatz zur Behebung von PF4 kann nicht als eigene Erweiterung gesehen werden, hier kann nur die Einarbeitung bzw. Beachtung der erläuterten Umstände in der jeweiligen `ChangeFactory` Abhilfe schaffen.

Diskutiert werden muss auch die generelle Motivation zur Behebung von PF3 und PF4, bzw. ob ein Aufwand hier prinzipiell gerechtfertigt ist. Aus pragmatisch technischer Sicht wurde die Relevanz beider Problemfälle eingehend beschrieben. Der Nutzen einer Bearbeitung bzw. Behebung dürfte, zumindest aus dieser Sicht, jedenfalls nachvollziehbar sein. Die Anwendersicht bzw. der praktische Nutzen fällt in den überaus subjektiven Kontext der Benutzbarkeit[3], womit aber das Blickfeld der Arbeit verlassen werden würde.

Ein weiterer Punkt bezieht sich auf die Notwendigkeit eigener Symbole für neu erstellte Makrodifferenzen. Hier muss ebenfalls die Frage nach dem praktischen Nutzen gestellt werden, da die EMF Compare Diagrammerweiterung bereits Symbole für jede Diagrammform mitbringt. Für die vorliegende Arbeit erschließt sich der praktische Nutzen rein in

[1] `org.eclipse.emf.compare.rcp.ui.differenceGroupExtender`
[2] konkret dem Plugin `org.eclipse.emf.compare.ide.ui`
[3] engl.: Usability

der einfacheren Erkennbarkeit der neu hinzugefügten Symbole, sowie in der durch den Entwicklungsprozess gesammelten Erfahrung.

Der erstellte Quellcode orientiert sich in der Hauptsache an den funktionellen Vorgaben und im Allgemeinen an dem Zustand der EMF Compare Diagrammerweiterung. Die Erweiterungen sind als funktionelle Prototypen zu verstehen, ein Produktiveinsatz war dabei nicht im Fokus. Die erstellten Erweiterungen sind somit auch nur auf die beschriebenen Problemfälle zugeschnitten und stellen keine gesamtheitlichen Lösungen im Versionierungskontext dar. Nebenwirkungen und mögliche unerwünschte Auswirkungen beim Einsatz der erstellten Erweiterungen, z.B. beim Zusammenführen von Modellen, können aktuell nicht ausgeschlossen werden. Da das Hauptaugenmerk der gewählten Problemfälle auf der korrekten Anzeige der Diagrammdifferenzen lag, wurden weiterführenden Aspekte nicht analysiert und bei der Ausarbeitung daher auch nicht beachtet. Wie bereits zu Beginn des Abschnitts über die Analysephase erläutert, unterliegen die erstellten Erweiterungen gewissen Einschränkungen, was die verwendeten Versionen von Programmpaketen betrifft. Es wurde kein spezielles Augenmerk auf Interoperabilität oder Effizienz gelegt. Gerade in Hinsicht auf die Eclipse Version gäbe es Potenzial für Verbesserungen.

Weitere Arbeit könnte eine Anpassung an den derzeit aktuellen Versionskontext, die Überarbeitung des Quellcodes nach EMF Compare Projektvorgaben oder die Eingliederung der Funktionalität in bestehende Erweiterungen zum Ziel haben.

Auch für eine nach außen gewandte kritische Bemerkung soll Platz sein. Die Einarbeitung in Schlüssel- oder Kerntechnologien wurde immer wieder gebremst, durch den leider oft inkonsistenten Zustand der EMF Compare Dokumentation bzw. der Dokumentation einiger anderer benötigter Eclipse Projekte, wie z.B. GMF. Bestehende Dokumentation verlinkt häufig auf veraltete oder nicht mehr existente Inhalte, manche Dokumente beschreiben oft längst überholte Versionsstände. Ebenfalls wenig hilfreich sind leider relativ häufig anzutreffende Platzhalter in bestehender Dokumentation, so z.B. „PENDING“[4], oder „If the meaning of the [...] reference isn't clear, there really should be more of a description here...“[5]. Besonders der zweite Lückenfüller stimmt nachdenklich, da er sehr häufig vorkommt und zwar in den Javadocs generierter Projektklassen. Dort wiederum in speziellen Bereichen, die für manuelle Änderungen vorbehalten sind und bei neuerlicher Generierung nicht überschrieben werden [176]. Warum solche nichtssagende Platzhalter eingefügt werden, kann nur spekuliert werden. Nun handelt es sich hierbei um OpenSource Projekte und es steht jedem frei daran teilzunehmen und Änderungen einzubringen. Dazu müsste das Wissen aber vorhanden sein. Demjenigen, der die Dokumentation konsultiert, weil er das Wissen nicht hat, nutzt das leider nichts.

[4] z.B. im EMF Compare Developer Guide [6]

[5] allein sechs mal im EMF Compare Hauptprojekt `org.eclipse.emf.compare`

ANHANG A

EMF Compare Differenzmodell

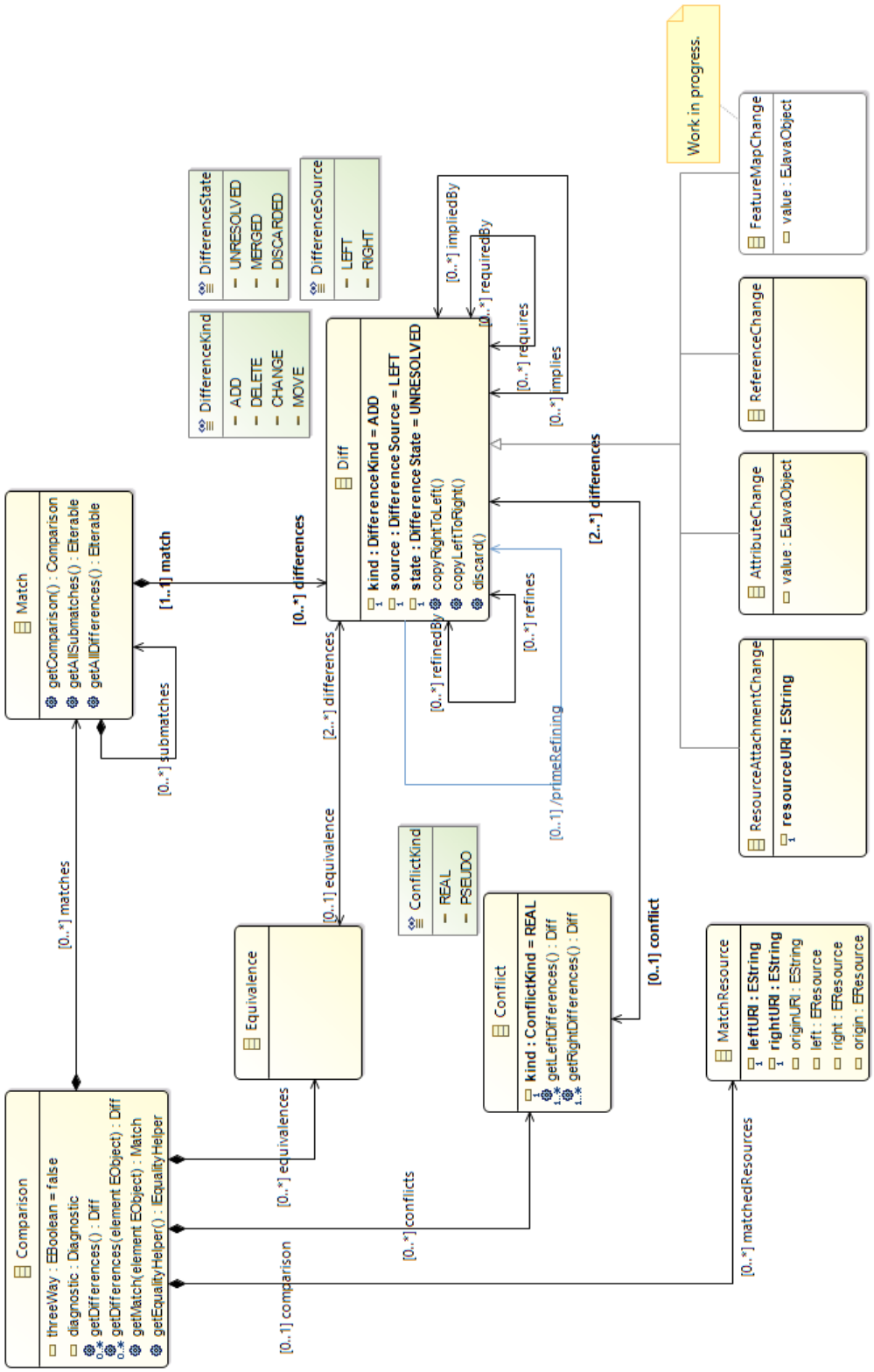


Abb. A.1: EMF Compare Vergleichsmodell [6]

ANHANG B

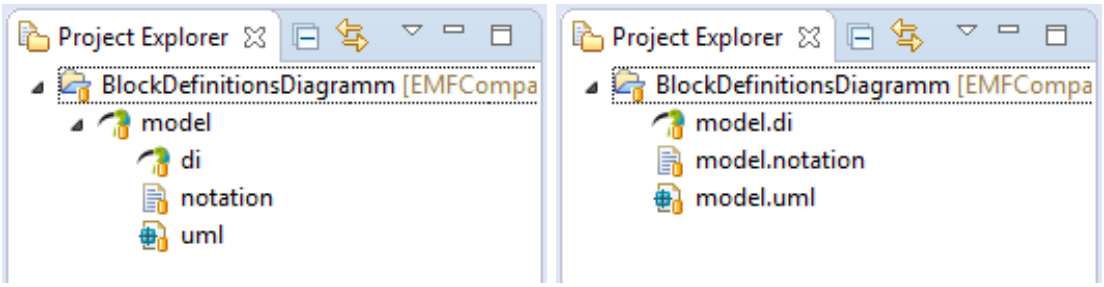
Anmerkungen zum Analysevorgehen

B.1 Einführung, Terminologie und Vorgehen

Dieser Anhang ist der Definition und Erläuterung verschiedener Begriffe im Umgang mit EMF Compare und der Vergleichsansicht vorbehalten. Diese Einführung soll zu einem besseren Verständnis beitragen und dabei helfen, Missverständnissen vorzubeugen. Aus Gründen der Nachvollziehbarkeit und Überprüfbarkeit werden auch Vorgehensweisen beschrieben, die bei der Durchführung der notwendigen Vergleiche zum Einsatz kamen. Die beschriebenen Vorgehensweisen wurden immer in genau der selben Abfolge von Schritten durchgeführt. Ein Abweichen von den beschriebenen Vorgehensweisen könnte unter Umständen zu anderen Ergebnissen führen. Es werden nicht alle englischen Fachbegriffe in deutscher Sprache wiedergegeben, sollte keine kontextuell passende oder nur eine holprige Übersetzung existieren, wird der englische Begriff verwendet.

Bei der Erstellung eines neuen Papyrus Projekts wurden keine Diagramme zur automatischen Erstellung ausgewählt, es wurde jedes Mal mit einem leeren Modell gestartet. Bei der Erstellung eines neuen Modells wird standardmäßig der Name `model` vorgeschlagen, in den folgenden, beispielhaften Erläuterungen wird dieser Name der Einfachheit halber beibehalten.

Unabhängig von der Anzahl der enthaltenen Diagramme, besteht ein Modell in Papyrus immer nur aus drei Dateien. In diesem Beispielszenario sind das die Dateien `model.di`, `model.notation` und `model.uml`. Die Datei `model.notation` enthält die Diagramminformationen, die Datei `model.uml` enthält die reinen Modellinformationen und die Datei `model.di` dient dem Speichern von Papyrus spezifischen Metadaten, also nicht direkt diagramm- oder modellbezogene Informationen , z.B. Konfigurationsdaten. Um eine Differenzierung zur eigentlichen Modelldatei `model.uml` zu schaffen, wird die



(a) mit Filter „Di view“ (b) ohne Filter

Abb. B.1: Papyrus Projektansicht

Gesamtheit des Datenmaterials eines Papyrusmodells in der Folge als Gesamtmodell bezeichnet.

Die tatsächliche Dateistruktur wird von Papyrus standardmäßig verborgen, angezeigt werden in der Papyrus Perspektive nur der Modellname und die eigentlichen Dateiendungen `di`, `notation` und `uml` in Form einer Verzeichnisstruktur[1], siehe Abb. B.1.

Wenn im Weiteren also von Modellvergleichen gesprochen wird, sind immer Vergleiche des Gesamtmodells gemeint, nicht jedoch Vergleiche einzelner Modelldateien, wie etwa `model.uml`. Weiters sind mit Modellvergleich immer 3-Wege Vergleiche bezeichnet, die notwendigen Schritte dafür werden weiter unten im Text beschrieben. Die Begründungen für die Reduzierung auf 3-Wege Vergleiche sind in den Beschränkungen zu finden, die sich durch 2-Wege Vergleiche ergeben, siehe dazu Abschnitt 2.4.2.

Ein 3-Wege Vergleich zwischen zwei Zweigen, dem aktiven Zweig im Workspace und einem Zweig aus dem Repository wird angestoßen, indem das Gesamtmodell ausgewählt bzw. markiert wird, mit einem Rechtsklick darauf das Kontextmenü geöffnet wird und in der Auswahl „Compare-with“ die Option „Branch, Tag or Reference...“ ausgewählt wird. Im erscheinenden Dialog wird dann der gewünschte Zweig für den Vergleich markiert und mit einem Klick auf den Button „Compare“ bestätigt.

Zentrales Anschauungsobjekt dieser Erläuterungen bzw. des ersten Teils der Analyse ist die Vergleichsansicht. Bezeichnet wird damit ein Teil der „Team Synchronize“ Perspektive von Eclipse, siehe Abb. B.2. Eclipse schlägt einen Wechsel zu dieses Perspektive beim ersten Vergleich automatisch vor, es ist dies die Standardperspektive für Vergleiche im Versionierungskontext. Soll diese Perspektive bei nachfolgenden Vergleichen ohne weitere Nachfrage automatisch geöffnet werden, so muss die Option zum Merken der Entscheidung aktiviert werden.

Modellen muss die Teilnahme an der Synchronisierung mit einem VCS explizit ermöglicht werden. Dazu muss vorab eine spezielle Option in den Einstellungen aktiviert werden.

[1] Dieses Gruppierung kann über die Anzeigeoption „Di view“ des Project Explorer Views geändert werden. Nach der Deaktivierung werden die tatsächlichen Dateien angezeigt.

In der hier beschriebenen Projektkonfiguration wird Git als VCS verwendet, die Option heißt hier „Allow models (e.g., Java, EMF) to participate in synchronizations“[2]. Diese Option ist nicht global für alle VCS aktivierbar, sondern muss für das jeweilige zum Einsatz kommende VCS separat aktiviert werden.

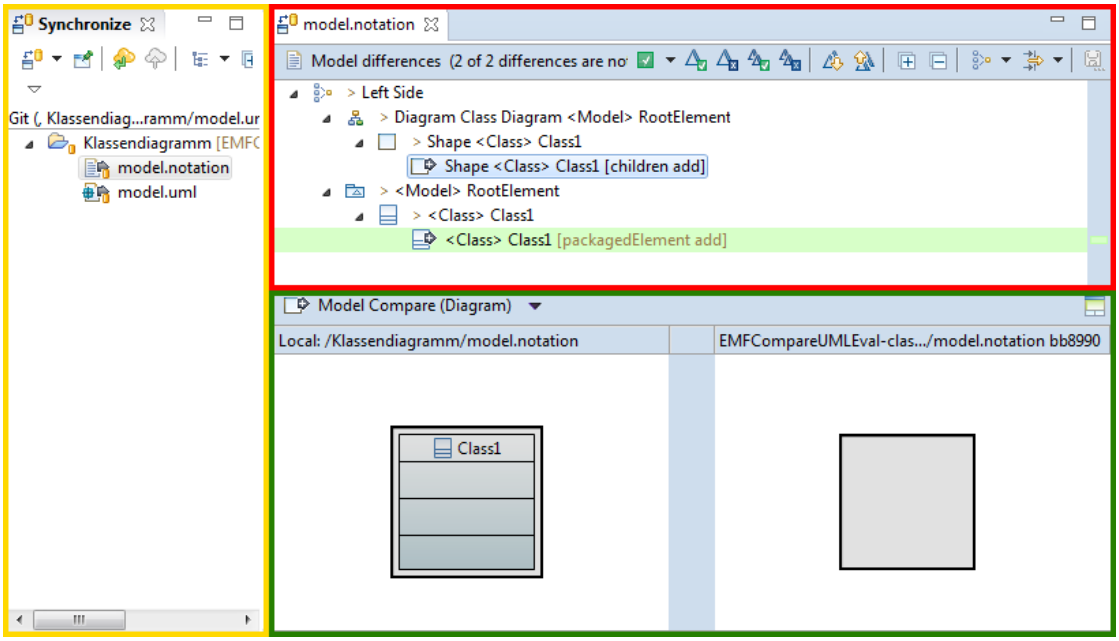


Abb. B.2: Eclipse „Team Synchronize“ Perspektive

Die „Team Synchronize“ Perspektive ist unterteilt in die Bereiche `StructureMergeViewer` (rote Umrandung) und `ContentMergeViewer` (grüne Umrandung). Links davon wird zur Navigation eine Baumansicht der veränderten Dateien angezeigt (gelbe Umrandung). Diese Navigationsansicht ist weniger interessant, da sie bei Modellvergleichen nur eine bis maximal drei Elemente enthält. Es sind dies die drei oben erwähnten Dateien aus denen jedes Papyrusmodell besteht. Interessant ist hier aber der Bruch in der Repräsentation der Modelldateien. In der Papyrus Perspektive werden die tatsächlichen Dateinamen normalerweise verborgen, hier sind sie immer in der realen Form wiederzufinden. Werden im Beispielszenario reine Modelländerungen vorgenommen, ohne das Diagramm zu verändern, ist hier nur die `model.uml` Datei zu finden, da sämtliche Modellinformationen darin abgelegt werden. Wird im Gegenteil nur das Diagramm verändert, ohne Änderungen am Modell, ist hier nur die `model.notation` Datei zu finden, da sämtliche Diagramminformationen ausschließlich hier abgelegt werden. Die Änderung

[2]Diese Option findet sich in den Grundeinstellungen (Menü Window > Preferences) in der Kategorie „Team“ und dort in der jeweiligen VCS Abteilung. Hier am Beispiel Git, befindet sich die Option in der Abteilung „Synchronize“.

des Namens einer Eigenschaft etwa, kann zwar im Diagramm angezeigt werden, geändert wird allerdings nur das Modell, also die `model.uml` Datei. Die Positionsänderung eines Diagrammelements wird im Gegenteil nur eine Änderung in der `model.notation` Datei bewirken. Allgemein wird in den Erläuterungen von gemischten Änderungen ausgegangen, also sowohl Diagramm-, als auch Modelländerungen.

Der `ContentMergeViewer` Bereich ist hier ebenfalls von untergeordneter Bedeutung. Hier werden Diagrammänderungen grafisch dargestellt oder geänderte Elemente in einer Text- bzw. Baumansicht angezeigt, jeweils als Gegenüberstellung von wahlweise zwei oder drei Modellversionen[3].

Hauptsächlich von Interesse ist der sogenannte `StrukturemergeViewer` Bereich. In der weiteren Folge wird dieser Bereich als Vergleichsansicht bezeichnet. Nach der Auswahl einer veränderten Datei per Doppelklick im Navigationsbereich werden in der Vergleichsansicht sowohl die Diagramm- als auch Modelländerungen als Differenzen der verglichenen Versionen in einer Baumstruktur dargestellt, siehe Abb. B.3. Die Anzeige ist immer in Diagramm- und Modelländerungen unterteilt. Dabei ist zu bemerken, dass bei gleichzeitigen Diagramm- und Modelländerungen, ungeachtet der Auswahl im Navigationsbereich immer alle Änderungen in der Baumstruktur der Vergleichsansicht angezeigt werden. EMF Compare zeichnet Beziehungen bzw. Abhängigkeiten zwischen Elementen ein. Das markierte Differenzelement, „Shape <Class> Class1 [children add]“, ist das im Diagramm sichtbare Symbol für das hinzugefügte Modellelement, „<Class> Class1 [packagedElement add]“, gut erkennbar an der grünen Hinterlegung.

In Abb. B.3 kommt die voreingestellte Gruppierungsoption „By Side“ zur Darstellung, die Änderungen werden dabei nach links- und rechtsseitig eingeteilt, je nachdem, wo die Änderung vorgenommen wurde. Weitere Möglichkeiten sind die Gruppierungsoption „Default“, die trotz ihres Namens nicht standardmäßig zum Einsatz kommt und keine Seitengruppierung vornimmt, sondern alle Änderungen gemischt anzeigt, sowie die Gruppierungsoption „By Kind“, die nach Art der Änderung gruppiert und die Gruppierung „By Resource“, die Änderungen nach Modelldatei gruppiert. Jede dieser Gruppierungsansichten wird nach dem schon öfter erwähnten EMF Compare Vergleichsmodell erstellt.

EMF Compare geht bei einseitigen Änderungen immer von der linken Seite aus, deshalb wird in Abb. B.3 das Element „Left Side“ als Wurzel angezeigt. In der nächsten Ebene findet sich die Aufteilung in Änderungen aus Diagramm bzw. Modell, diese Elemente finden ihre Entsprechung im Vergleichsmodell als Match. Jede der drei verglichenen Versionen enthält z.B. das selbe Diagramm. Der nächste Match unter dem Diagramm stellt ein Shape Element dar, aber noch keine Änderung bzw. Differenz. Erst in der nächsten, darunterliegenden Ebene ist die tatsächliche Differenz dargestellt, erkennbar an dem Zusatz „[children add]“ in der Bezeichnung. Nur Differenzen tragen diesen Zusatz. Bezeichnet wird damit die Metareferenz - in dem Fall „children“ - und die Art der Änderung, hier „add“. Es wurde also ein Shape Element als Kind oder Subelement des Diagramms hinzugefügt. Gut zu erkennen ist die Unterscheidung zwischen Modelldiffe-

[3] Es wird, wie bereits erwähnt, immer von 3-Weg Vergleichen ausgegangen.

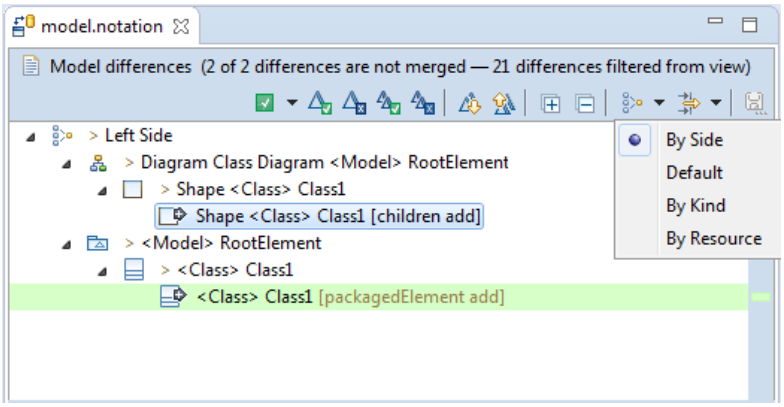


Abb. B.3: Vergleichsansicht mit aufgeklappten Gruppierungsoptionen

renzen und Diagrammdifferenzen. Für Abb. B.3 wurde lediglich eine neue UML Klasse erstellt, daraus resultieren bereits zwei Änderungen, das Hinzufügen der Klasse im Modell und das Hinzufügen der Klasse im Diagramm.

Wie bereits beschrieben, siehe Abschnitt 2.4.3, besteht eines der zentralen Probleme bei Modell- bzw. Diagrammvergleichen in der Differenz zwischen dem was der Benutzer in einem Diagramm verändert und dem was beim Vergleich der neuen Diagrammversion und den Vorgängerversionen als Unterschiede angezeigt wird. Diese Differenz entsteht konkret durch die Diagrammelemente, die zur Anzeige von Modellelementen verwendet und dargestellt werden.

Der Benutzer arbeitet bei der Erstellung von Modellen üblicherweise mit einem Diagrammeditor. Der Diagrammeditor stellt die Diagrammänderungen grafisch dar. Im Hintergrund werden die entsprechenden Änderungen am Modell vorgenommen. Der Diagrammeditor präsentiert dem Benutzer eine abstrahierte Darstellung des eigentlichen Modells in grafischer Form. Aktionen die der Benutzer vornimmt und aus einer einzigen, nicht mehr unterteilbaren Handlung bestehen, werden als atomare Aktionen oder Änderungen bezeichnet. Das einfache Hinzufügen eines Elements zum Diagramm, das Ändern eines Bezeichners, sowie das Verschieben oder Löschen eines Elements können als atomare Aktionen verstanden werden, da aus Benutzersicht zur Durchführung jeweils nur ein Schritt oder eine Aktion notwendig ist. Für den Benutzer führen atomare Aktionen zu atomaren Änderungen im Diagramm. Im Hintergrund jedoch und damit für den Benutzer nicht unmittelbar erfahrbar, zerfallen diese atomaren Änderungen in eine Vielzahl von Änderungen die sich aus dieser einen Aktion ergeben. Eine einzelne, atomare Aktion oder Änderung im Diagramm führt also zu vielen Änderungen oder Differenzen in der Vergleichsansicht.

Wird z.B. einem leeren UML-Diagramm eine neue Klasse hinzugefügt, so wird ein Element im zugrundeliegenden Modell erstellt, siehe Abb. B.4. Der Modellbaum enthält nach dem Einfügen ein „<Class>“ Element mit dem Namen „Class1“. Im Diagramm jedoch werden zur Anzeige der Klasse ungleich mehr Elemente benötigt und erstellt.

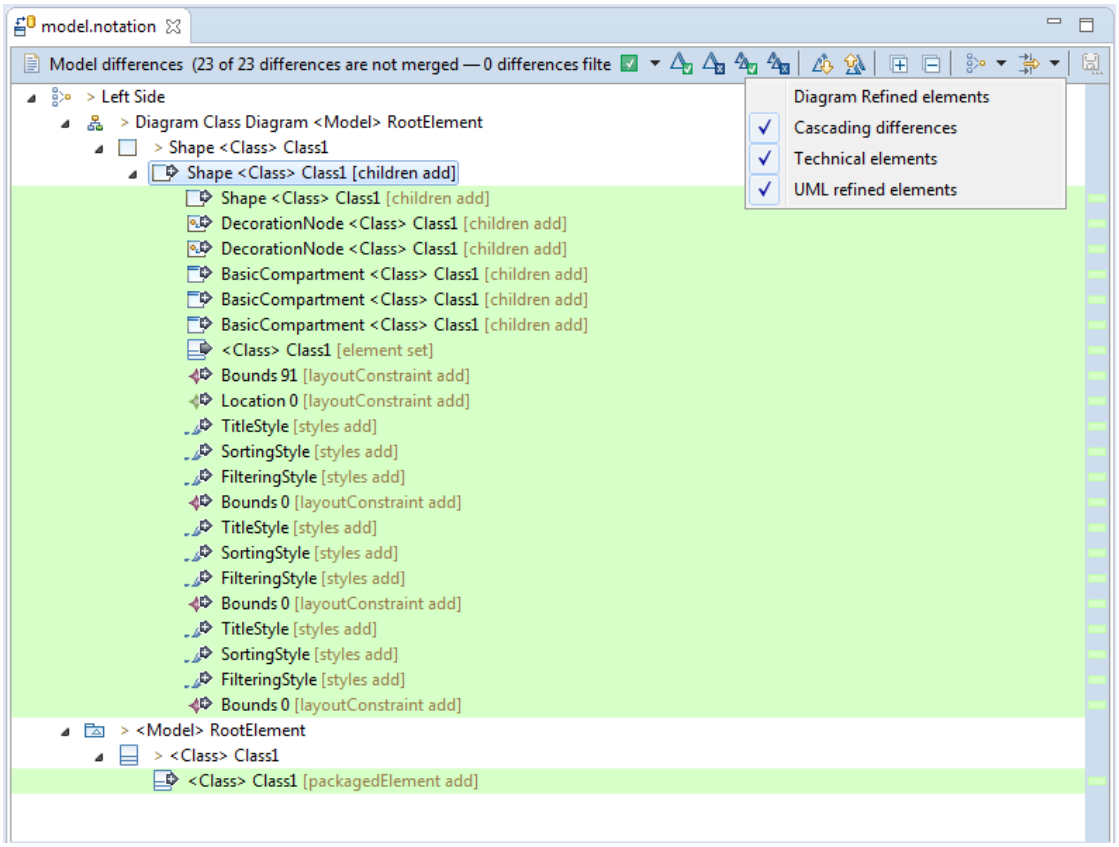


Abb. B.4: Vergleichsansicht einer neu hinzugefügten Klasse „Class1“

Der Diagrammbaum in der Vergleichsansicht enthält nach dem Einfügen der neuen Klasse 21 Elemente, die unterschiedliche Elementteile bzw. Änderungen darstellen. Die meisten Elemente werden im Diagramm benötigt, um eine UML-Klasse anzuzeigen. Unter Verwendung der Normaleinstellungen der Vergleichsanzeige werden diese Elemente jedoch nicht angezeigt, da sie über eine Filterfunktion ausgeblendet werden. Ein Filter erkennt gewisse Elemente bzw. Elementkonstellationen und unterbinden die Anzeige. Der Filter „Diagramm Refined Changes“ etwa filtert alle Differenzelemente aus, die von einem anderen, übergeordneten Differenzelement[4] verfeinert werden.

Verfeinerung bedeutet im Kontext des obigen Beispiels, dass anstatt der 21 Änderungen, die für die Anzeige des Diagrammelements „Class1“ benötigt werden und deshalb in der Vergleichsansicht auftauchen, nur eine Differenz angezeigt wird, siehe Abb. B.3. Diese Differenz steht stellvertretend für alle anderen Differenzen. Die notwendigen Änderungen werden durch die verfeinerte Änderung abstrahiert und repräsentiert, es stellt sozusagen einen Container für die Änderungen oder die Zusammenfassung aller Änderungen dar.

[4]engl.: high-level difference [6]

Die EMF Compare Dokumentation [6] spricht hier von einer makroskopischen Differenz[5]. Die untergeordneten Differenzen werden als Teildifferenzen[6] bezeichnet[7]. Die Subdifferenzen werden von der Makrodifferenz zusammengefasst oder gekapselt. Dieser Vorgang wird in der Fachliteratur auch als „semantische Aufwertung“[8] bezeichnet [91]. Durch die Zusammenfassung von Subdifferenzen, die für den Benutzer nicht unmittelbar nachvollziehbare Änderungen darstellen, zu einer stellvertretenden Makrodifferenz, wird die gesamte Bedeutung der Teilelemente in einem Element konzentriert. Eine derart vereinfachte Darstellung gewinnt an Übersichtlichkeit und Nachvollziehbarkeit und wertet das Ergebnis für den Benutzer insgesamt auf.

Als Zielvorstellung soll hier das pragmatische Ideal einer klaren, für den Benutzer einfach nachvollziehbaren 1:1 Relation dienen. Für eine Änderung im Diagramm soll eine Makrodifferenz in der Vergleichsanzeige dargestellt werden. Und zwar ungeachtet der eigentlichen Anzahl an Subdifferenzen bzw. notwendigen Änderungen im Hintergrund.

Die oben erwähnten Filterfunktionen können dabei vom Benutzer lediglich aktiviert bzw. deaktiviert werden. Die verfügbaren Filter sind in aller Regel automatisch aktiviert. Das wahlfreie Filtern nach beliebigen Elementen wird nicht angeboten, der Benutzer hat also keine Möglichkeit hier über die angebotenen Optionen hinaus einzugreifen. Nun wäre das Ausblenden sämtlicher unbekannter Elemente zwar möglich, allerdings wäre dabei nicht klar, was genau ausgeblendet würde. Somit könnten Informationen verborgen werden, die für den Benutzer von Interesse wären. Grundvoraussetzung für ein sinnvolles Vorgehen sind also die Suche, die Identifizierung und die entsprechende Verarbeitung bzw. Verfeinerung der diversen erstellten Elemente bzw. der daraus resultierenden Differenzen.

EMF Compare summiert alle Differenzen zu einer Gesamtzahl und differenziert dann zwischen angezeigten und gefilterten Differenzen, siehe Abbildunge B.3 für gefilterte Differenzen und im Vergleich dazu Abb. B.4 ohne gefilterte Differenzen. Jede Differenz die im Zuge des EMF Vergleichsprozesses erstellt wird, ob von EMF Compare oder einer Erweiterung, wird zu dieser Summe gezählt.

B.2 Beschreibung der Detailanalyse zu PF1: Unbenannte UDDs

Da mit den Mitteln der Vergleichsansicht nicht mehr über unbenannte UDDs zu erfahren war, wurde das schrittweise Debuggen von Teilen des Vergleichsprozesses notwendig.

Wie bereits in Abschnitt 2.5.4 beschrieben, bietet EMF Compare, verteilt über den gesamten mehrphasigen Vergleichsprozess, eine Reihe von Möglichkeiten an, um über Erweiterungen in den Vergleichsprozess einzugreifen. Als Einstiegspunkt besonders gut geeignet erschien hier ein `PostProcessor`, da beim Aufruf einer der angebotenen

[5] im Folgenden wird der Begriff Makrodifferenz verwendet
[6] engl.: unit difference
[7] im Folgenden wird die Bezeichnung Subdifferenz verwendet
[8] engl.: semantic lifting

Methoden, z.B. `postComparison`, jedes Mal das `Comparison` Objekt als Argument übergeben wird. In diesem Objekt findet sich das gesamte, im Vergleichsprozess bis zu diesem Punkt erstellte Vergleichsmodell, siehe dazu Abb. A.1 in Anhang A.

Besonders gut geeignet erschien der `CompareDiagramPostProcessor`[9] aus der `DiagramCompare` Erweiterung. Dieser `PostProcessor` ist zuständig für die Erkennung und Verfeinerung von Diagrammelementen. Die oben genannte Methode `postComparison` wird nach Abarbeitung aller Schritte des Vergleichsprozesses aufgerufen. Im Vergleichsmodell enthalten sind alle gefundenen Elementpaare, Differenzen und Konflikte die sich aus den verglichenen Diagrammen ergeben. Interessant waren hier vor allem die Differenzen, die zum Großteil von EMF Compare aus den gefundenen und verglichenen Paaren erstellt werden, und in weiterer Folge von den diversen Erweiterungen verbessert und ergänzt bzw. verfeinert werden. Um einen besseren Überblick über die Differenzen zu erlangen, wurde eine angepasste Version der Hilfsklasse `EMFComparePrettyPrinter`[10] eingesetzt.

Aufgabe der angepassten Hilfsklasse war die Ausgabe einer Baumansicht aller geänderten Elemente, mitsamt der erkannten Differenzen auf der Konsole, siehe Abb. B.5. Angestoßen wurde die Ausgabe jeweils über einen `PostProcessor`, der als erstes nach dem kompletten Durchlaufen aller Vergleichsphasen (`postComparison`) aufgerufen wurde und einen `PostProcessor`, der als letztes aufgerufen wurde, also nach allen anderen Erweiterungen. Die Ausgabe entsprach dem Objektbaum des Vergleichsmodells, einmal vor der Ausführung der diversen Erweiterungen und einmal danach. Dadurch konnten die Differenzen, die EMF Compare erzeugte, mit den Differenzen verglichen werden, die von den diversen Erweiterungen erzeugt wurden. Ein weiteres Hilfsmittel waren `ItemProvider`, die speziell angepasst wurden, um Objektdetails, nämlich Klassenname und Objekthash (Klassenname@Objekthash), in der Vergleichsansicht anzuzeigen.

Konkret wurde dafür die Subklasse `DebuggingCompareItemProviderAdapterFactorySpec` von der Klasse `CompareItemProviderAdapterFactorySpec`[11] abgeleitet und als `Extension`[12] registriert. Diese Factory erzeugt bei Bedarf einen `ItemProvider` der Subklasse `DebuggingReferenceChangeItemProviderSpec`, die wiederum von der Klasse `ReferenceChangeItemProviderSpec`[13] abgeleitet wurde. Aufgabe dieser abgeleiteten `ItemProvider` Klasse ist, die Methode `getValuetext` zu überschreiben, um zusätzlich die oben genannten Objektdetails auszugeben, sowie die Begrenzung der Zeilenlänge aufzuheben. Damit waren zwei unterschiedliche Ansichten auf die Informationen des Vergleichsmodells gegeben und konnten bis ins Detail analysiert und verglichen werden. Interessant dabei waren einerseits die Unterschiede der dargestellten Strukturen und andererseits die ausgegebenen Objektdetails, hauptsächlich die Klassennamen in Verbindung mit den jeweiligen Objekthashes. Damit konnten die Objekte über alle

[9] `org.eclipse.emf.compare.diagram.internal.CompareDiagramPostProcessor`
[10] `org.eclipse.emf.compare.utils.EMFComparePrettyPrinter`
[11] `org.eclipse.emf.compare.provider.spec.CompareItemProviderAdapterFactorySpec`
[12] `ExtensionPoint: org.eclipse.emf.edit.itemProviderAdapterFactories`
[13] `org.eclipse.emf.compare.provider.spec.ReferenceChangeItemProviderSpec`

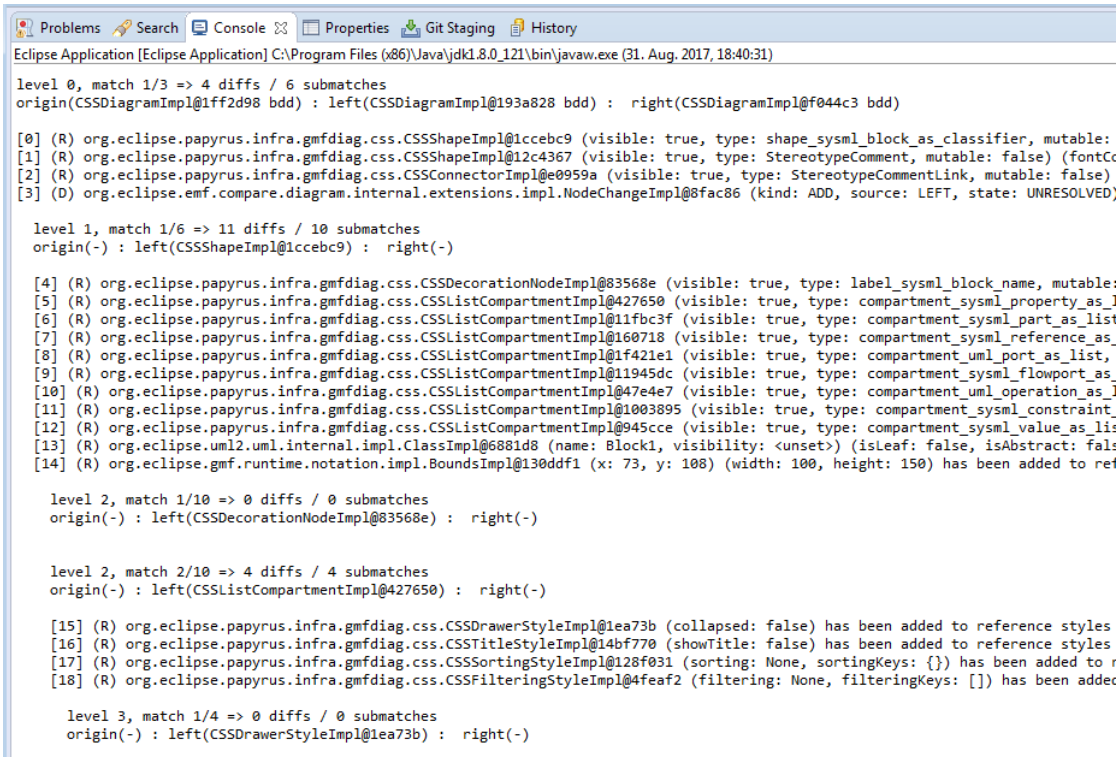


Abb. B.5: Konsolenausgabe der angepassten Hilfsklasse `EMFComparePrettyPrinter`

Ansichten zugeordnet und verfolgt werden. In Abb. B.6 ist eine derartig erweiterte Vergleichsansicht dargestellt, für diese Ansicht wurden die Filter „Diagram Refined elements“ und „Cascading differences“ deaktiviert. Durch Abgleichen der Objektbaumansicht mit der Vergleichsansicht, konnten die unbenannten UDDs identifiziert werden.

Das Differenzelement „Shape <null>“ entspricht einem Diagrammobjekt der Klasse `CSSShapeImpl`[14], das Attribut `type` enthält die Kennzeichnung `StereotypeComment`. Das Differenzelement „Connector <null>“ entspricht einem Diagrammobjekt `CSSConnectorImpl`[15], das Attribut `type` enthält die Kennzeichnung `StereotypeCommentLink`. Die korrelierenden Elemente waren damit identifiziert. Die Typbezeichnungen deuteten auf einen Zusammenhang mit zugewiesenen Stereotypen hin, im Falle eines SysML Blocks nämlich das Stereotyp „Block“.

[14] `org.eclipse.papyrus.infra.gmfdiag.css.CSSShapeImpl`
[15] `org.eclipse.papyrus.infra.gmfdiag.css.CSSConnectorImpl`

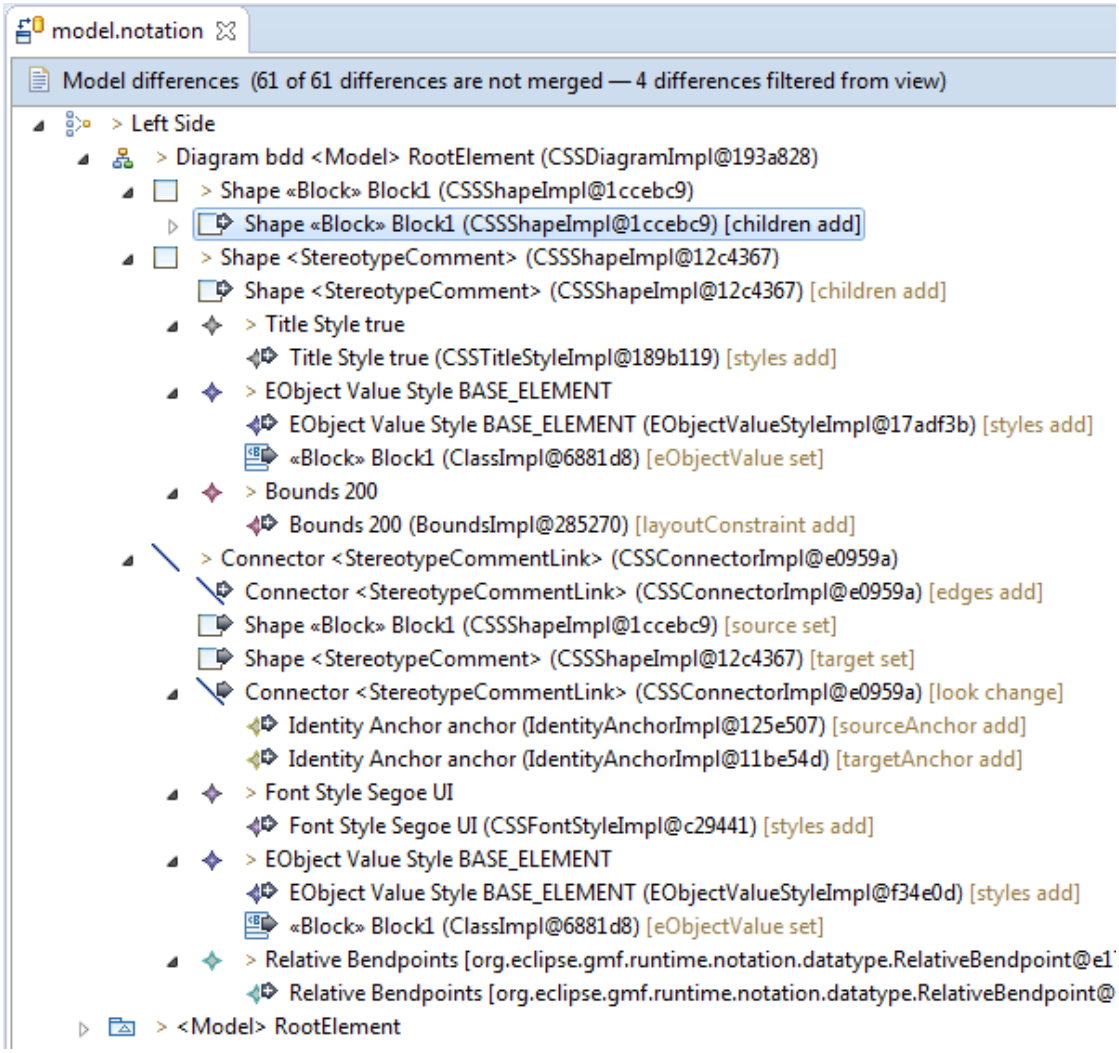


Abb. B.6: Vergleichsansicht mit erweiterter Elementbeschriftung

ANHANG C

Analyseprotokolle

C.1 Blockdefinitionsdiagramm (bdd)

Aktion	Branch	verglichen mit	erwartete Anzahl	angezeigte Anzahl	Diagramm-Änderungen	Modell-Änderungen
add BDD	0.1	master	2	13	1	12
add block	1.1	0.1	2	4	3	1
change block (Name)	1.2	1.1	1	1	0	1
remove block	1.3	1.2	2	4	3	1
add property	2.1	2.0	1	1	0	1
change property (Typ)	2.2	2.1	1	1	0	1
remove property	2.3	2.2	2	1	0	1
add operation	3.1	3.0	1	1	0	1
change operation (Name)	3.2	3.1	1	1	0	1
remove operation	3.3	3.2	1	1	0	1
add constraint	4.1	4.0	2	2	1	1
change constraint (Name)	4.2	4.1	1	1	0	1

remove constraint	4.3	4.2	2	2	1	1
add association	5.1	5.0	2	4	3	1
change association (Name)	5.2	5.1	1	1	0	1
remove association	5.3	5.2	2	2	1	1
add directed association	6.1	6.0	2	2	1	1
change directed association (Name)	6.2	6.1	1	1	0	1
remove directed asociation	6.3	6.2	2	2	1	1
add composition	7.1	7.0	2	2	1	1
change composition (Name)	7.2	7.1	1	1	0	1
remove composition	7.3	7.2	2	2	1	1
add port	8.1	8.0	2	6	5	1
change port (Name)	8.2	8.1	1	1	0	1
remove port	8.3	8.2	6	2	1	1
add generalization	9.1	9.0	2	2	1	1
change generalization (Position)	9.2	9.1	1	1	0	1
remove generalization	9.3	9.2	2	2	1	1
add actor	10.1	10.0	2	2	1	1
change actor (Name)	10.2	10.1	1	1	0	1
remove actor	10.3	10.2	2	2	1	1
add valueType	11.1	11.0	2	4	3	1
change valueType (Name)	11.2	11.1	1	1	0	1
remove valueType	11.3	11.2	2	4	3	1
add comment	12.1	12.0	2	2	1	1
add comment link	12.2	12.1	2	2	1	1
change comment (Wert)	12.3	12.2	1	1	0	1
remove comment link	12.4	12.3	2	2	1	1

remove comment	12.5	12.4	2	2	1	1
add block	13.1	13.0	2	4	3	1
move block	13.2	13.1	1	1	1	0
remove block	13.3	13.2	2	4	3	1
add block	14.1	14.0	2	4	3	1
resize block	14.2	14.1	2	2	2	0
remove block	14.3	14.2	2	4	3	1
change appearance (Stereotype-Kommentar: 1)	15.2	15.1	1	3	3	0
change appearance (Stereotype-Kommentar: 0)	15.3	15.2	1	2	2	0

C.2 Internes Blockdiagramm (ibd)

Aktion	Branch	verglichen mit	erwartete Anzahl	angezeigte Anzahl	Diagramm-Änderungen	Modell-Änderungen
add IBD	0.1	master	1	15	2	13
add part	1.1	0.1	2	5	3	2
change part (Name)	1.2	1.3	1	1	0	1
remove part	1.3	1.2	2	2	1	1
add port	2.1	2.0	2	6	5	1
change port (Name)	2.2	2.1	1	1	0	1
remove port	2.3	2.2	6	2	1	1
add part and refeence	3.0	0.1	4	6	2	4
add connector	3.1	3.0	2	2	1	1
change connector (Name)	3.2	3.1	1	1	0	1
remove connector	3.3	3.2	2	2	1	1

C.3 Zusicherungsdiagramm (par)

Aktion	Branch	verglichen mit	erwartete Anzahl	angezeigte Anzahl	Diagramm-Änderungen	Modell-Änderungen
add PAR	0.1	master	1	15	2	13
add constraint-Property	1.1	0.1	2	7	5	2
change constraint-Property (Name)	1.2	1.1	1	1	0	1
remove constraint-Property	1.3	1.2	7	4	3	1
add constraint-Property	2.0	0.1	2	5	3	2
add constraint-Property	2.1	2.0	2	6	5	1
change constraintParameter (Name)	2.2	2.1	1	1	0	1
remove constraint-Parameter	2.3	2.2	2	2	1	1
add value	3.1	0.1	1	4	2	2
change value (Name)	3.2	3.1	1	1	0	1
remove value	3.3	3.2	4	2	1	1
add constraint-Property, constraint-Parameter, value	4.0	0.1	6	8	4	4
add connector	4.1	4.0	1	4	3	1
change connector	4.2	4.1	1	1	0	1
remove connector	4.3	4.2	1	4	3	1

C.4 Anforderungsdiagramm (req)

Aktion	Branch	verglichen mit	erwartete Anzahl	angezeigte Anzahl	Diagramm-Änderungen	Modell-Änderungen
add REQ	0.1	master	1	13	1	12
add requirement	1.1	0.1	2	4	3	1
change requirement (Name)	1.2	1.1	1	1	0	1
remove requirement	1.3	1.1	2	4	3	1
add package	2.1	0.1	2	2	1	1
change package (Name)	2.2	2.1	1	1	0	1
remove package	2.3	2.2	2	2	1	1
add requirements	3.0	0.1	4	8	6	2
add abstraction	3.1	3.0	2	4	3	1
change abstraction (Name)	3.2	3.1	1	1	0	1
remove abstraction	3.3	3.2	2	4	3	1
add requirements	4.0	0.1	4	8	6	2
add connection	4.1	4.0	2	4	3	1
change connection (Form)	4.2	4.1	1	1	0	1
remove connection	4.3	4.2	2	1	1	0

C.5 Aktivitätsdiagramm (act)

Aktion	Branch	verglichen mit	erwartete Anzahl	angezeigte Anzahl	Diagramm-Änderungen	Modell-Änderungen
add ACT	0.1	master	1	15	2	13
add action	1.1	0.1	2	2	1	1
change action	1.2	1.1	1	1	0	1
remove action	1.3	1.2	2	3	2	1
add action	2.0	0.1	2	2	1	1
add pin	2.1	2.0	2	3	2	1
change pin (Name)	2.2	2.1	1	1	0	1
remove pin	2.3	2.2	2	2	1	1
add node	3.1	0.1	2	2	1	1
change node (Name)	3.2	3.1	1	1	0	1
remove node	3.3	3.2	2	3	2	1
add actions	4.0	0.1	4	4	2	2
add control flow	4.1	4.0	2	4	1	3
change control flow (Name)	4.2	4.1	1	1	0	1
remove control flow	4.3	4.2	2	5	2	3

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] *Duden | homonym | Rechtschreibung, Bedeutung, Definition, Synonyme, Herkunft.* http://www.duden.de/rechtschreibung/homonym, besucht: 2016-06-24 .

[2] *Duden | meta-, Meta-, vor Vokalen und vor h met-, Met- | Rechtschreibung, Bedeutung, Definition, Herkunft.* http://www.duden.de/rechtschreibung/meta_, besucht: 2016-08-01 .

[3] *Duden | Modell | Rechtschreibung, Bedeutung, Definition, Synonyme, Herkunft.* http://www.duden.de/rechtschreibung/Modell, besucht: 2016-06-23 .

[4] *Eclipse Modeling Project.* http://www.eclipse.org/modeling/emf/, besucht: 2016-11-22 .

[5] *Eclipse Platform Help : Platform Plug-in Developer Guide - Programmer's Guide - Platform architecture.* http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm&cp=2_0_1, besucht: 2016-11-05 .

[6] *EMF Compare Developer guide.* http://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html, besucht: 2016-11-05 .

[7] *Git branch | Atlassian Git Tutorial.* https://www.atlassian.com/git/tutorials/using-branches/git-branch, besucht: 2016-11-05 .

[8] *History of Systems Engineering.* https://www.incose.org/AboutSE/history-of-systems-engineering, besucht: 2016-10-22 .

[9] *Model Driven Acronyms | Models Everywhere.* https://modelseverywhere.wordpress.com/2010/10/31/model-driven-acronyms/, besucht: 2016-11-04 .

[10] *OMG SysML.* http://www.omgsysml.org/, besucht: 2016-11-05 .

[11] *Why Git is Better Than X.* https://web.archive.org/web/20120211173441/http://whygitisbetterthanx.com/#distributed, besucht: 2016-11-05 .

[12] *Eclipse Modeling Framework - Interview with Ed Merks*, 2010. https://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-100007.html, besucht: 2016-11-22 .

[13] Alanen, M. und Porres, I.: *Difference and Union of Models.* In: *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*, S. 2–17. Springer Berlin Heidelberg, 2003.

[14] Altmanninger, K.: *Models in Conflict - Towards a Semantically Enhanced Version Control System for Models.* In: *Models in Software Engineering, Workshops and Symposia at MoDELS 2007*, S. 293–304, 2007.

[15] Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K. und Wimmer, M.: *Why Model Versioning Research is Needed!? An Experience Report.* In: *Proceedings of the Joint MoDSE-MCCM 2009 Workshop*, 2009.

[16] Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W. und Wimmer, M.: *AMOR - Towards Adaptable Model Versioning.* In: *1st Int. Workshop on Model Co-Evolution and Consistency Management, in conjunction with Models'08*, 2008.

[17] Altmanninger, K., Seidl, M. und Wimmer, M.: *A survey on model versioning approaches.* International Journal of Web Information Systems, 5(3):271–304, 2009.

[18] Asenov, D., Guenat, B., Müller, P. und Otth, M.: *Precise Version Control of Trees with Line-Based Version Control Systems.* In: *Fundamental Approaches to Software Engineering*, S. 152–169. Springer Berlin Heidelberg, 2017.

[19] Atkinson, C. und Kühne, T.: *Rearchitecting the UML Infrastructure.* ACM Trans. Model. Comput. Simul., 12(4):290–321, 2002.

[20] Atkinson, C. und Kuhne, T.: *Model-driven development: a metamodeling foundation.* IEEE Software, 20(5):36–41, 2003.

[21] Baker, L., Clemente, P., Cohen, B., Permenter, L., Purves, B. und Salmon, P.: *Foundational concepts for model driven system design.* INCOSE Model Driven System Design Interest Group, 16, 2000.

[22] Barone, I., Lucia, A. D., Fasano, F., Rullo, E., Scanniello, G. und Tortora, G.: *COMOVER: Concurrent model versioning.* In: *2008 IEEE International Conference on Software Maintenance*, S. 462–463, 2008.

[23] Barrett, S., Chalin, P. und Butler, G.: *Model merging falls short of software engineering needs.* In: *2nd Workshop on Model-Driven Software Evolution*, 2008.

[24] Barrett, S. C., Butler, G. und Chalin, P.: *Mirador: A Synthesis of Model Matching Strategies.* In: *Proceedings of the 1st International Workshop on Model Comparison in Practice*, S. 2–10. ACM, 2010.

[25] Bernzen, R.: *Modell.* In: *Europäische Enzyklopädie zu Philosophie und Wissenschaften*, Bd. L - Q. Meiner, 1990.

[26] Bersoff, E. H., Henderson, V. D. und Siegel, S. G.: *Software Configuration Management.* SIGMETRICS Perform. Eval. Rev., 7(3-4):9–17, 1978.

[27] Brambilla, M., Cabot, J. und Wimmer, M.: *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 1. Aufl., 2012.

[28] Brand, M. van den, Protić, Z. und Verhoeff, T.: *Fine-grained Metamodel-assisted Model Comparison.* In: *Proceedings of the 1st International Workshop on Model Comparison in Practice*, S. 11–20. ACM, 2010.

[29] Brand, M. van den, Protić, Z. und Verhoeff, T.: *Generic Tool for Visualization of Model Differences.* In: *Proceedings of the 1st International Workshop on Model Comparison in Practice*, S. 66–75. ACM, 2010.

[30] Brand, M. van den, Protic, Z. Z. und Verhoeff, T. T.: *RCVDiff - a stand-alone tool for representation, calculation and visualization of model differences.* Association for Computing Machinery, Inc, 2011.

[31] Brocke, J. v.: *Referenzmodellierung: Gestaltung und Verteilung von Konstruktionsprozessen.* Dissertation, Universität Münster, 2003.

[32] Brooks Jr., F. P.: *No Silver Bullet - Essence and Accidents of Software Engineering.* IEEE Computer, 20(4):10–19, 1987.

[33] Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K. und Wimmer, M.: *An Introduction to Model Versioning.* In: *Formal Methods for Model-Driven Engineering*, S. 336–398. Springer Berlin Heidelberg, 2012.

[34] Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K. und Wimmer, M.: *The Past, Present, and Future of Model Versioning.* In: *Emerging Technologies for the Evolution and Maintenance of Software Models*, S. 410–443. IGI Global, 2012.

[35] Brosch, P., Kappel, G., Seidl, M., Wieland, K., Wimmer, M., Kargl, H. und Langer, P.: *Adaptable Model Versioning in Action.* In: *Modellierung 2010*, S. 221–236, 2010.

[36] Brosch, P., Langer, P., Seidl, M., Wimmer, M. und Kappel, G.: *Generic vs. Language-Specific Model Versioning - Adaptability to the Rescue.* Softwaretechnik-Trends, 32(4), 2012.

[37] Brouse, P. S.: *Configuration management in: A.* Systems Engineering and management for Sustainable Development-Volume I, S. 214, 2009.

[38] Brun, C. und Pierantonio, A.: *Model differences in the eclipse modeling framework.* UPGRADE, The European Journal for the Informatics Professional, 9(2):29–34, 2008.

[39] Bézivin, J.: *In search of a basic principle for model driven engineering.* Novatica – Special Issue on UML (Unified Modeling Language), 5(2):21–24, 2004.

[40] Bézivin, J.: *In search of a basic principle for model driven engineering.* Novatica Journal, Special Issue, 5(2):21–24, 2004.

[41] Bézivin, J.: *On the unification power of models.* Software & Systems Modeling, 4(2):171–188, 2005.

[42] Bézivin, J.: *Model Driven Engineering: An Emerging Technical Space.* In: *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, S. 36–64. Springer Berlin Heidelberg, 2006.

[43] Bézivin, J. und Gerbé, O.: *Towards a Precise Definition of the OMG/MDA Framework.* In: *16th IEEE International Conference on Automated Software Engineering*, S. 273–280, 2001.

[44] Bézivin, J. und Heckel, R.: *04101 Summary - Language Engineering for Model-driven Software Development.* In: *Language Engineering for Model-Driven Software Development, 29. February - 5. March 2004*, 2004.

[45] Bézivin, J., Jouault, F. und Valduriez, P.: *On the Need for Megamodels.* In: *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications,(2004)*, 2004.

[46] Cadavid, J., Combemale, B. und Baudry, B.: *Ten years of Meta-Object Facility: an Analysis of Metamodeling Practices.* Techn. Ber. RR-7882, INRIA, 2012.

[47] Chacon, S. und Straub, B.: *Pro Git - Book.* `https://git-scm.com/book/de/v1`, besucht: 2016-11-22 .

[48] Chen, P. P.: *The Entity-Relationship Model - Toward a Unified View of Data.* ACM Trans. Database Syst., 1(1):9–36, 1976.

[49] Cicchetti, A. und Ciccozzi, F.: *Towards a Novel Model Versioning Approach Based on the Separation Between Linguistic and Ontological Aspects.* In: *Proceedings of the Workshop on Models and Evolution co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems*, S. 60–69, 2013.

[50] Cicchetti, A., Ruscio, D. D. und Pierantonio, A.: *A Metamodel Independent Approach to Difference Representation.* Journal of Object Technology, 6(9):165–185, 2007.

[51] Conradi, R. und Westfechtel, B.: *Version Models for Software Configuration Management.* ACM Comput. Surv., 30(2):232–282, 1998.

[52] Dahl, O. J. und Nygaard, K.: *SIMULA: An ALGOL-based Simulation Language.* Commun. ACM, 9(9):671–678, 1966.

[53] De Lucia, A., Fasano, F., Scanniello, G. und Tortora, G.: *Concurrent Fine-Grained Versioning of UML Models.* In: *13th European Conference on Software Maintenance and Reengineering*, S. 89–98. IEEE, 2009.

[54] Delligatti, L.: *SysML Distilled: A Brief Guide to the Systems Modeling Language.* Addison-Wesley Professional, 2013.

[55] Estefan, J. A.: *Survey of model-based systems engineering (MBSE) methodologies.* Incose MBSE Focus Group, 2008.

[56] Favre, J. M.: *Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon1.* In: *Language Engineering for Model-Driven Software Development*, 2004.

[57] Favre, J. M.: *Megamodelling and Etymology.* In: *Transformation Techniques in Software Engineering*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik, 2006.

[58] Fettke, P. und Loos, P.: *Referenzmodellierungsforschung.* Wirtschaftsinformatik, 46(5):331–340, 2004.

[59] Fischer, J.: *Was haben Analogrechner und Simula-67 mit modernen Modellierungssprachen zu tun?* In: *Informatik: Aktuelle Themen im historischen Kontext*, S. 105–133. Springer Berlin Heidelberg, 2006.

[60] Fowler, M.: *Refactoring - Improving the Design of Existing Code.* Addison Wesley object technology series. Addison-Wesley, 1999.

[61] France, R. B., Ghosh, S., Dinh-Trong, T. und Solberg, A.: *Model-driven development using UML 2.0: promises and pitfalls.* Computer, 39(2):59–66, 2006.

[62] Friedenthal, S. und Burkhart, R.: *Extending UML from software to systems.* In: *INCOSE Symp, July*, 2003.

[63] Friedenthal, S., Moore, A. und Steiner, R.: *A Practical Guide to SysML, Third Edition: The Systems Modeling Language.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.

[64] Förtsch, S. und Westfechtel, B.: *Differencing and Merging of Software Diagrams - State of the Art and Challenges.* In: *Proceedings of the Second International Conference on Software and Data Technologies*, S. 90–99, 2007.

[65] Fuentes-Fernández, L. und Vallecillo-Moreno, A.: *An introduction to UML profiles.* UML and Model Engineering, 2, 2004.

[66] Gamma, E., Helm, R., Johnson, R. und Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley professional computing series. Addison-Wesley Boston , Mass., 1994.

[67] Gasevic, D., Djuric, D. und Devedzic, V.: *Model Driven Engineering and Ontology Development.* Springer Publishing Company, Incorporated, 2. Aufl., 2009.

[68] Girschick, M.: *Difference detection and visualization in UML class diagrams.* Technical University of Darmstadt Technical Report TUD-CS-2006-5, S. 1–15, 2006.

[69] Greenfield, J. und Short, K.: *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools.* In: *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, S. 16–27. ACM, 2003.

[70] Grossman, M., Aronson, J. E. und McCarthy, R. V.: *Does UML Make the Grade? Insights from the Software Development Community.* Inf. Softw. Technol., 47(6):383–397, 2005.

[71] Hailpern, B. und Tarr, P.: *Model-driven Development: The Good, the Bad, and the Ugly.* IBM Syst. J., 45(3):451–461, 2006.

[72] Harel, D. und Rumpe, B.: *Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff.* Techn. Ber., Weizmann Science Press of Israel, 2000.

[73] Harel, D. und Rumpe, B.: *Meaningful Modeling: What's the Semantics of "Semantics"?* IEEE Computer, 37(10):64–72, 2004.

[74] Hause, M.: *The SysML Modelling Language.* In: *Fifth European Systems Engineering Conference.* INCOSE, 2006.

[75] Hause, M.: *The sysml modelling language.* In: *Fifteenth European Systems Engineering Conference*, Bd. 9. Citeseer, 2006.

[76] Henderson-Sellers, B.: *UML – the Good, the Bad or the Ugly? Perspectives from a panel of experts.* Software & Systems Modeling, 4(1):4–13, 2005.

[77] Henderson-Sellers, B.: *Bridging Metamodels and Ontologies in Software Engineering.* J. Syst. Softw., 84(2):301–313, 2011.

[78] Henderson-Sellers, B. und Gonzalez-Perez, C.: *Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0.* In: *Model Driven Engineering Languages and Systems: 9th International Conference*, S. 16–26. Springer Berlin Heidelberg, 2006.

[79] Henderson-Sellers, B. und Unhelkar, B.: *Open Modeling with UML.* ACM Press/Addison-Wesley Publishing Co., 2000.

[80] Hesse, W.: *Modelle - Janusköpfe der Software-Entwicklung - oder: Mit Janus von der A- zur S-Klasse.* In: *Modellierung 2006*, S. 99–113, 2006.

[81] Hesse, W.: *More matters on (meta-)modelling: remarks on Thomas Kühne's "matters".* Software & Systems Modeling, 5(4):387–394, 2006.

[82] Hesse, W. und Mayr, C. H.: *Modellierung in der Softwaretechnik: eine Bestandsaufnahme.* Informatik-Spektrum, 31(5):377–393, 2008.

[83] Hevner, A. R., March, S. T., Park, J. und Ram, S.: *Design Science in Information Systems Research.* MIS Quarterly, 28(1):75–105, 2004.

[84] Hutchinson, J., Rouncefield, M. und Whittle, J.: *Model-driven engineering practices in industry.* In: *2011 33rd International Conference on Software Engineering (ICSE)*, S. 633–642, 2011.

[85] Jouault, F., Bézivin, J. und Barbero, M.: *Towards an advanced model-driven engineering toolbox.* Innovations in Systems and Software Engineering, 5(1):5–12, 2009.

[86] Kaschek, R.: *Was sind eigentlich Modelle?* EMISA Forum, 19(1):31–35, 1999.

[87] Kaschek, R.: *Schwachstellen einer Analyse des Modellbegriffs.* EMISA Forum, 20(1):11–15, 2000.

[88] Kaufmann, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W. und Schwinger, W.: *An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example.* In: *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, S. 271–285. Springer, 2009.

[89] Kehrer, T., Kelter, U., Ohrndorf, M. und Sollbach, T.: *Understanding model evolution through semantically lifting model differences with SiLift.* In: *28th IEEE International Conference on Software Maintenance*, S. 638–641, 2012.

[90] Kehrer, T., Kelter, U., Pietsch, P. und Schmidt, M.: *Adaptability of Model Comparison Tools.* In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, S. 306–309. ACM, 2012.

[91] Kehrer, T., Kelter, U. und Taentzer, G.: *A Rule-based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning.* In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, S. 163–172. IEEE Computer Society, 2011.

[92] Kelter, U., Schmidt, M. und Wenzel, S.: *Architekturen von Differenzwerkzeugen für Modelle.* In: *Software Engineering 2008. Fachtagung des GI-Fachbereichs Softwaretechnik*, S. 155–168, 2008.

[93] Kelter, U., Wehren, J. und Niere, J.: *A Generic Difference Algorithm for UML Models.* In: *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik*, S. 105–116, 2005.

[94] Kent, S.: *Model Driven Engineering.* In: *Proceedings of the Third International Conference on Integrated Formal Methods*, S. 286–298. Springer-Verlag London, UK, 2002.

[95] Khelladi, D. E., Bendraou, R. und Gervais, M. P.: *Towards a User-Guided Difference-Based Detection of Atomic Changes.* In: *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, S. 211–214, 2016.

[96] Kühne, T.: *Clarifying matters of (meta-) modeling: an author's reply.* Software & Systems Modeling, 5(4):395–401, 2006.

[97] Kühne, T.: *Matters of (Meta-) Modeling.* Software & Systems Modeling, 5(4):369–385, 2006, ISSN 1619-1374.

[98] Kobryn, C.: *Will UML 2.0 Be Agile or Awkward?* Commun. ACM, 45(1):107–110, 2002.

[99] Koch, S., Krell, M. und Krüger, D.: *Förderung von Modellkompetenz durch den Einsatz einer Blackbox.* In: *Erkenntnisweg Biologiedidaktik*, Bd. 14, S. 93–108. Freie Universität Berlin, Didaktik der Biologie, 2015.

[100] Koegel, M. und Helming, J.: *EMFStore: A Model Repository for EMF Models.* In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, S. 307–308. ACM, 2010.

[101] Koegel, M., Helming, J. und Seyboth, S.: *Operation-based Conflict Detection and Resolution.* In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, S. 43–48. IEEE Computer Society, 2009.

[102] Koegel, M. und Langer, P.: *Integrating Open-Source Modeling Projects: Collaborative Modeling with Papyrus and EMF Compare.* In: *Proceedings of the International Workshop on Open Source Software for Model Driven Engineering*, 2015.

[103] Kolovos, D. S.: *Establishing Correspondences between Models with the Epsilon Comparison Language.* In: *Model Driven Architecture - Foundations and Applications*, S. 146–157. Springer Berlin Heidelberg, 2009.

[104] Kolovos, D. S., Di Ruscio, D., Pierantonio, A. und Paige, R. F.: *Different models for model matching: An analysis of approaches to support model differencing.* In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, S. 1–6. IEEE Computer Society, 2009.

[105] Kolovos, D. S., Paige, R. F. und Polack, F. A.: *Model Comparison: A Foundation for Model Composition and Model Transformation Testing.* In: *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, S. 13–20. ACM, 2006.

[106] Kolovos, D. S., Rose, L. M., Paige, R. F., Guerra, E. M., Cuadrado, J. S., De Lara, J. M., Ráth, I., Varró, D., Sunyé, G. und Tisi, M.: *MONDO: Scalable Modelling and Model Management on the Cloud.* In: *STAF2015 Project Showcase*, 2015.

[107] Konemann, P.: *Model-independent Differences.* In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, S. 37–42. IEEE Computer Society, 2009.

[108] Koshima, A. A. und Englebert, V.: *Collaborative Editing of EMF/Ecore Metamodels and Models.* Sci. Comput. Program., 113(P1):3–28, 2015.

[109] Krahn, H., Rumpe, B. und Völkel, S.: *Integrated Definition of Abstract and Concrete Syntax for Textual Languages.* In: *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, S. 286–300. Springer Berlin Heidelberg, 2007.

[110] Kreisel, G.: *Modell.* In: *Handbuch wissenschaftstheoretischer Begriffe*, S. G – Q. Vandenhoeck & Ruprecht, Göttingen, 1980.

[111] Langer, P., Mayerhofer, T. und Kappel, G.: *Semantic Model Differencing Utilizing Behavioral Semantics Specifications.* In: *Model-Driven Engineering Languages and Systems*, S. 116–132. Springer International Publishing, 2014.

[112] Langer, P., Wimmer, M., Gray, J., Kappel, G. und Vallecillo, A.: *Language-Specific Model Versioning Based on Signifiers.* Journal of Object Technology, 11(3):4–1, 2012.

[113] Liddle, S. W.: *Model-Driven Software Development.* In: *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, S. 17–54. Springer Berlin Heidelberg, 2011.

[114] Lin, Y., Gray, J. und Jouault, F.: *DSMDiff: a differentiation tool for domain-specific models.* European Journal of Information Systems, 16(4):349–361, 2007.

[115] Lindkvist, C., Stasis, A. und Whyte, J.: *Configuration Management in Complex Engineering Projects.* Procedia CIRP, 11:173 – 176, 2013.

[116] Long, J.: *Relationships between common graphical representations in Systems Engineering.* Vitech white paper, Vitech Corporation, Vienna, VA, S. 70, 2002.

[117] Lucia, A. D., Fasano, F., Scanniello, G. und Tortora, G.: *Concurrent Fine-Grained Versioning of UML Models.* In: *2009 13th European Conference on Software Maintenance and Reengineering*, S. 89–98, 2009.

[118] Ludewig, J.: *Modelle im Software Engineering - eine Einführung und Kritik.* In: *Modellierung 2002, Modellierung in der Praxis - Modellierung für die Praxis, Arbeitstagung der GI*, S. 7–22, 2002.

[119] Ludewig, J.: *Models in software engineering – an introduction.* Software & Systems Modeling, 2(1):5–14, 2003.

[120] Mahr, B.: *Das Modell des Modellseins.* In: *Modelle.* Lang, Frankfurt, Wien, 2008.

[121] Mahr, B.: *Die Informatik und die Logik der Modelle.* Informatik-Spektrum, 32(3):228–249, 2009.

[122] Maoz, S. und Ringert, J. O.: *A framework for relating syntactic and semantic model differences.* In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, S. 24–33, 2016.

[123] Maoz, S., Ringert, J. O. und Rumpe, B.: *ADDiff: Semantic Differencing for Activity Diagrams.* In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, S. 179–189. ACM, 2011.

[124] Maoz, S., Ringert, J. O. und Rumpe, B.: *CDDiff: Semantic Differencing for Class Diagrams.* In: *ECOOP 2011 – Object-Oriented Programming*, S. 230–254. Springer Berlin Heidelberg, 2011.

[125] Maoz, S., Ringert, J. O. und Rumpe, B.: *A Manifesto for Semantic Model Differencing.* In: *Models in Software Engineering*, S. 194–203. Springer Berlin Heidelberg, 2011.

[126] Marcus Alanen und Ivan Porres: *Version Control of Software Models.* In: *Advances in UML and XML-Based Software Evolution*, S. 47–70. IGI Global, 2005.

[127] Mayerhofer, T., Langer, P., Wimmer, M. und Kappel, G.: *xMOF: Executable DSMLs Based on fUML.* In: *Software Language Engineering*, S. 56–75. Springer International Publishing, 2013.

[128] Mehra, A., Grundy, J. und Hosking, J.: *A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design.* In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, S. 204–213. ACM, 2005.

[129] Mellor, S. J., Clark, A. N. und Futagami, T.: *Model-driven development - Guest editor's introduction.* IEEE Software, 20(5):14–18, 2003.

[130] Mens, T.: *A state-of-the-art survey on software merging.* IEEE Transactions on Software Engineering, 28(5):449–462, 2002.

[131] Meyer, B.: *Object-oriented software construction.* Prentice-Hall, Upper Saddle River, NJ, USA, 2. Aufl., 1997.

[132] Mittelstraß, J.: *Anmerkungen zum Modellbegriff.* Debatte; 2 - Modelle des Denkens : Streitgespräch in der Wissenschaftlichen Sitzung der Versammlung der Berlin-Brandenburgischen Akademie der Wissenschaften am 12. Dezember 2003, 2:65–67, 2005.

[133] Müller, Roland: *Zur Geschichte des Modelldenkens und des Modellbegriffs.* In: *Modelle - Konstruktion der Wirklichkeit.* Fink, München, 1983.

[134] Muller, P. A., Fondement, F., Baudry, B. und Combemale, B.: *Modeling modeling modeling.* Software and Systems Modeling, 11(3):347–359, 2012.

[135] Murta, L., Corrêa, C., Prudêncio, J. G. und Werner, C.: *Towards odyssey-VCS 2: Improvements over a UML-based Version Control System.* In: *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*, S. 25–30. ACM, 2008.

[136] Murta, L., Corrêa, C., Prudêncio, J. G. und Werner, C.: *Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System.* In: *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*, S. 25–30. ACM, 2008.

[137] Nguyen, T. N., Munson, E. V., Boyland, J. T. und Thao, C.: *An Infrastructure for Development of Object-oriented, Multi-level Configuration Management Services.* In: *Proceedings of the 27th International Conference on Software Engineering*, S. 215–224. ACM, 2005.

[138] Ober, I. und Prinz, A.: *What do we need metamodels for?* In: *Nordic Workshop on UML and Software Modelling*, S. 8–28. Agder University College - Faculty of Engineering and Science, 2006.

[139] Oda, T. und Saeki, M.: *Generative Technique of Version Control Systems for Software Diagrams.* In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*, S. 515–524. IEEE Computer Society, 2005.

[140] Oestereich, Bernd, .: *Analyse und Design mit der UML 2.5. Objektorientierte Softwareentwicklung.* Oldenbourg, München, 10., aktualisierte u. erw. Aufl., 2012.

[141] Ohst, D., Welle, M. und Kelter, U.: *Differences between versions of UML diagrams.* In: *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference*, S. 227–236, 2003.

[142] Ohst, D., Welle, M. und Kelter, U.: *Differences Between Versions of UML Diagrams.* In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, S. 227–236. ACM, 2003.

[143] Oliveira, H., Murta, L. und Werner, C.: *Odyssey-VCS: A Flexible Version Control System for UML Model Elements.* In: *Proceedings of the 12th International Workshop on Software Configuration Management*, S. 1–16. ACM, 2005.

[144] Oliveira, H., Murta, L. und Werner, C.: *Odyssey-VCS: A Flexible Version Control System for UML Model Elements.* In: *Proceedings of the 12th International Workshop on Software Configuration Management*, S. 1–16. ACM, 2005.

[145] OMG: *MDA Guide Version 1.0.1*, 2003. `http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf`, besucht: 2016-10-14 .

[146] OMG: *OMG Meta Object Facility (MOF) Core Spezification Version 2.5.1*, 2016. `https://www.omg.org/spec/MOF/2.5.1/PDF`, besucht: 2016-11-22 .

[147] OMG: *OMG System Modeling Language Version 1.5*, 2017. `https://www.omg.org/spec/SysML/1.5/PDF`, besucht: 2017-06-15 .

[148] OMG: *OMG Unified Modeling Language (OMG UML) Version 2.5.1*, 2017. `https://www.omg.org/spec/UML/2.5.1/PDF`, besucht: 2017-12-18 .

[149] Peffers, K., Tuunanen, T., Rothenberger, M. und Chatterjee, S.: *A Design Science Research Methodology for Information Systems Research.* Journal of Management Information Systems, 24(3):45–77, 2008.

[150] Peters, W.: *Zur Theorie der Modellierung von Natur und Umwelt.* Dissertation, Technische Universität Berlin, 2003.

[151] Reisig, W.: *50 Jahre modellbasierter Entwurf: Vom Modellieren mit Programmen zum Programmieren mit Modellen.* In: *Informatik: Aktuelle Themen im historischen Kontext*, S. 275–314. Springer Berlin Heidelberg, 2006.

[152] Reiter, T., Altmanninger, K., Bergmayr, A., Schwinger, W. und Kotsis, G.: *Models in Conflict–Detection of Semantic Conflicts in Model-based Development.* In: *Proceedings of the 3rd int. Workshop on Model-driven Enterprise Information Ssytems*, 2007.

[153] Rivera, J. E. und Vallecillo, A.: *Representing and Operating with Model Differences.* In: *Objects, Components, Models and Patterns*, S. 141–160. Springer Berlin Heidelberg, 2008.

[154] Rochkind, M. J.: *The source code control system.* IEEE Transactions on Software Engineering, SE-1(4):364–370, 1975.

[155] Rodriguez-Priego, E., García-Izquierdo, F. J. und Rubio, n. L.: *Modeling Issues: a Survival Guide for a Non-expert Modeler.* In: *Model Driven Engineering Languages and Systems: 13th International Conference*, S. 361–375. Springer Berlin Heidelberg, 2010.

[156] Rothenberg, Jeff: *The Nature of Modeling.* In: *Artificial Intelligence, Simulation & Modeling.* John Wiley & Sons, Inc., 1989.

[157] Rozen, R. van und Storm, T. van der: *Origin Tracking + Text Differencing = Textual Model Differencing.* In: *Theory and Practice of Model Transformations*, S. 18–33. Springer International Publishing, 2015.

[158] Schmidt, D. C.: *Guest Editor's Introduction: Model-Driven Engineering.* IEEE Computer, 39(2):25–31, 2006.

[159] Schmidt, M.: *SiDiff: generische, auf Ähnlichkeiten basierende Berechnung von Modelldifferenzen.* Softwaretechnik-Trends, 27(2), 2007.

[160] Schmidt, M. und Gloetzner, T.: *Constructing Difference Tools for Models Using the SiDiff Framework.* In: *Companion of the 30th International Conference on Software Engineering*, S. 947–948. ACM, 2008.

[161] Schneider, C., Zündorf, A. und Niere, J.: *CoObRA-a small step for development tools to collaborative environments.* In: *Workshop on Directions in Software Engineering Environments*, 2004.

[162] Schütte, R.: *Grundsätze ordnungsmäßiger Referenzmodellierung; Konstruktion konfigurations- und anpassungsorientierter Modelle.* Neue betriebswirtschaftliche Forschung ; 233. Gabler, Wiesbaden, 1998. Zugl.: Münster, Univ., Diss., 1997.

[163] Schütte, R.: *Zum Realitätsbezug von Informationsmodellen.* EMISA Forum, 19(2):26–36, 1999.

[164] Schütte, R.: *Realitätsbezug von Informationsmodellen - Eine Erwiderung auf Kritik.* EMISA Forum, 20(2):14–21, 2000.

[165] Seidl, M., Scholz, M., Huemer, C. und Kappel, G.: *UML@Classroom.* Undergraduate topics in computer science. Springer Berlin Heidelberg, 2015.

[166] Selic, B.: *The Pragmatics of Model-Driven Development.* IEEE Software, 20(5):19–25, 2003.

[167] Selic, B.: *A Systematic Approach to Domain-Specific Language Design Using UML.* In: *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, S. 2–9, 2007.

[168] Selonen, P.: *A Review of UML Model Comparison Techniques.* In: *Proceedings of the 5th Nordic Workshop on Model Driven Engineering*, S. 37–51, 2007.

[169] Selonen, P. und Kettunen, M.: *Metamodel-Based Inference of Inter-Model Correspondence.* In: *11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems*, S. 71–80, 2007.

[170] Sen, S., Moha, N., Baudry, B. und Jézéquel, J. M.: *Meta-model Pruning.* In: *Model Driven Engineering Languages and Systems: 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, S. 32–46. Springer Berlin Heidelberg, 2009.

[171] Solberg, A., France, R. und Reddy, R.: *Navigating the metamuddle.* In: *Proceedings of the 4th Workshop in Software Model Engineering*, 2005.

[172] Stachowiak, H.: *Allgemeine Modelltheorie.* Springer-Verlag, Wien, 1973.

[173] Stachowiak, H.: *Erkenntnisstufen zum Neopragmatismus und zur Modelltheorie.* In: *Modelle - Konstruktion der Wirklichkeit.* Fink, München, 1983.

[174] Stahl, T., Völter, M., Bettin, J., Haase, A. und Helsen, S.: *Model-Driven Software Development - Technology, Engineering, Management.* Pitman, 2006.

[175] Stahl, T., Voelter, M. und Czarnecki, K.: *Replace! Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[176] Steinberg, D., Budinsky, F., Paternostro, M. und Merks, E.: *EMF: Eclipse Modeling Framework, 2nd Edition.* Eclipse Series. Addison-Wesley Professional, 2008.

[177] Stephan, M. und Cordy, J. R.: *A Survey of Methods and Applications of Model Comparison.* Techn. Ber. 2011-582 Rev. 3, School of Computing, Queen's University, 2012.

[178] Stephan, M. und Cordy, J. R.: *A Survey of Model Comparison Approaches and Applications.* In: *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, S. 265–277, 2013.

[179] Strahringer, S.: *Zum Begriff des Metamodells.* Technische Hochschule Darmstadt, Institut für Betriebswirtschaftslehre, 1996.

[180] Strahringer, S.: *Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips.* In: *Modellierung '98, Proceedings des GI-Workshops in Münster*, 1998.

[181] The Collaborative Modeling Initiative: *Collaborative Modeling with Eclipse*. http://www.collaborative-modeling.org, besucht: 2016-03-18 .

[182] Thomas, O.: *Das Modellverständnis in der Wirtschaftsinformatik: Historie, Literaturanalyse und Begriffsexplikation*. Nr. 184 in *IWi-Hefte*. Institut für Wirtschaftsinformatik im Deutschen Forschungszentrum für Künstliche Intelligenz, Saarbrücken, 2005.

[183] Thomas, O.: *Das Referenzmodellverständnis in der Wirtschaftsinformatik : Historie, Literaturanalyse und Begriffsexplikation*. Techn. Ber., Saarländische Universitäts- und Landesbibliothek, 2006.

[184] Tichy, W. F.: *Rcs — a system for version control*. Software: Practice and Experience, 15(7):637–654, 1985.

[185] Tichy, W. F.: *Tools for Software Configuration Management*. In: *Proceedings of the International Workshop on Software Version and Configuration Control*, S. 1–20, 1988.

[186] Tichy, W. F.: *Software Configuration Management*. In: *Encyclopedia of Computer Science*, S. 1601–1604. John Wiley and Sons Ltd., Chichester, UK, 2003.

[187] Toulmé, A.: *Presentation of EMF compare utility*. In: *paper presented at the Eclipse Modeling Symposium*, 2006.

[188] Treude, C., Berlik, S., Wenzel, S. und Kelter, U.: *Difference Computation of Large Models*. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, S. 295–304. ACM, 2007.

[189] Universität Siegen: *Bibliography on Comparison and Versioning of Software Models*. http://pi.informatik.uni-siegen.de/CVSM/, besucht: 2016-10-28 .

[190] Völter, M.: *MD* Best Practices*. Journal of Object Technology, 8(6):79–102, 2009.

[191] Voelter, M., Visser, E., Helander, M., Benz, S., Engelmann, B., Dietrich, C. und Wachsmuth, G.: *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Publishing Platform, 2013.

[192] Walden, D. D., Roedler, G. J., Forsberg, K., Hamelin, R. D. und Shortell, T. M.: *Systems engineering handbook: a guide for system life cycle processes and activities*. John Wiley & Sons Inc, 4. Aufl., 2015.

[193] Wedekind, H., Görz, G., Kötter, R. und Inhetveen, R.: *Modellierung, Simulation, Visualisierung: Zu aktuellen Aufgaben der Informatik*. Informatik-Spektrum, 21(5):265–272, 1998.

[194] Weilkiens, T.: *OMG SysML 1.1 Notationsübersicht*, 2008. http://model-based-systems-engineering.com/wp-content/uploads/2012/08/sysmod-sysml-1.1-notationsübersicht-oose.pdf, besucht: 2016-03-14 .

[195] Weilkiens, T.: *Systems Engineering mit SysML/UML*. dpunkt.verlag GmbH, 3. Aufl., 2014.

[196] Wendler, R.: *Das Spiel mit Modellen*. In: *Visuelle Modelle*. Fink, München, 2008.

[197] Wenzel, S. und Kelter, U.: *Model-driven design pattern detection using difference calculation*. In: *Workshop on Pattern Detection for Reverse Engineering*, 2006.

[198] Whittle, J., Hutchinson, J. und Rouncefield, M.: *The State of Practice in Model-Driven Engineering*. IEEE Software, 31(3):79–85, 2014.

[199] Wieland, K., Fitzpatrick, G., Kappel, G., Seidl, M. und Wimmer, M.: *Towards an Understanding of Requirements for Model Versioning Support*. International Journal of People-Oriented Programming, 1(2):1–23, 2011.

[200] Wile, D. S.: *Abstract Syntax from Concrete Syntax*. In: *Proceedings of the 19th International Conference on Software Engineering*, S. 472–480. ACM, 1997.

[201] Xing, Z.: *Genericdiff: A general framework for model comparison*. Techn. Ber., National University of Singapore, 2010.

[202] Xing, Z.: *Model Comparison with GenericDiff*. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, S. 135–138. ACM, 2010.

[203] Xing, Z. und Stroulia, E.: *UMLDiff: An Algorithm for Object-oriented Design Differencing*. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, S. 54–65. ACM, 2005.

[204] Xing, Z. und Stroulia, E.: *Refactoring Detection based on UMLDiff Change-Facts Queries*. In: *2006 13th Working Conference on Reverse Engineering*, S. 263–274, 2006.