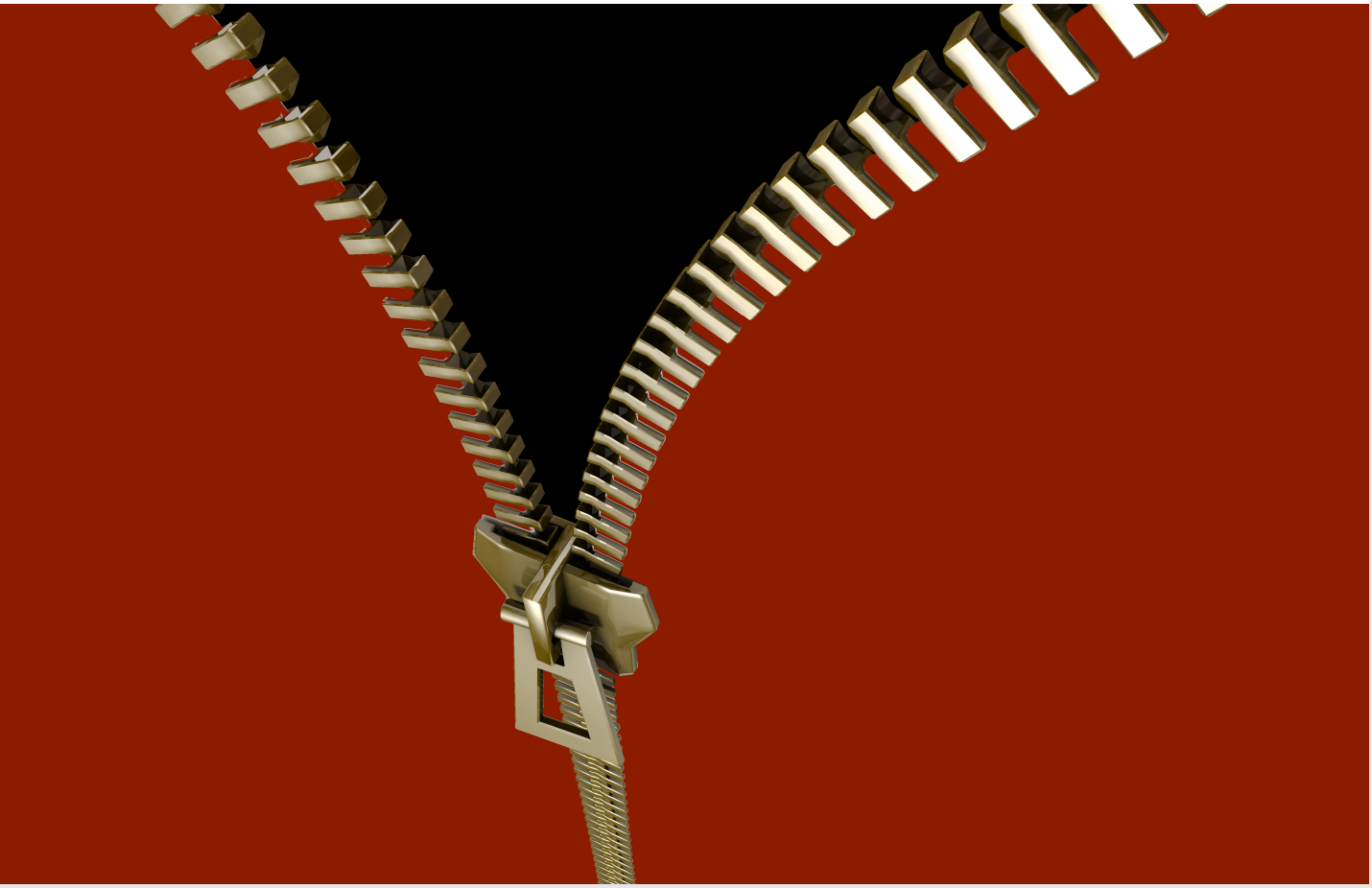
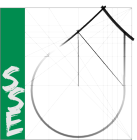
TU Clausthal



Christian Bartelt

Kollaborative Modellierung im Software Engineering

SSE-Dissertation 4



Software
Systems
Engineering

Institut für Informatik
Lehrstuhl von Prof. Dr. Andreas Rausch

Institut für Informatik der
Technischen Universität Clausthal

Kollaborative Modellierung im Software Engineering

Christian Bartelt

Vollständiger Abdruck der vom Institut für Informatik der Technischen Universität Clausthal zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:
Univ.-Prof. Dr. Niels Pinkwart
Technische Universität Clausthal

Prüfer der Dissertation:
1. Univ.-Prof. Dr. Andreas Rausch
Technische Universität Clausthal

2. Univ.-Prof. Dr. Manfred Broy
Technische Universität München

Die Dissertation wurde am 18.1.2011 bei der Technischen Universität Clausthal eingereicht und durch das Institut für Informatik am 6.4.2011 angenommen.

Kurzbeschreibung

Software Engineering ist Teamarbeit. Schon im Vorfeld der Programmierung modelliert und verfeinert ein Team aus Softwareingenieuren und Fachseite gemeinschaftlich (kollaborativ) die zu realisierende Software. Zur Beschreibung von Software haben sich weitgehend standardisierte Modellierungssprachen wie die UML aber auch eine Reihe domänenspezifischer Sprachen etabliert. Um die ambitionierten Zeitvorgaben heutiger Softwareprojekte einhalten zu können, arbeiten die Akteure oft hochgradig parallel an Softwaremodellen. Die effiziente und gleichzeitige korrekte Integration parallel und asynchron durchgeführter Modellbearbeitungen spielt dabei für den Projekterfolg eine immer entscheidendere Rolle.

Im Gegensatz zur Programmcodeverwaltung, bei der die effiziente und korrekte Integration von parallel bearbeiteten Codeversionen durch Werkzeuge hinreichend gut unterstützt wird, existiert für die Modellverwaltung noch keine adäquate Unterstützung. Aus diesem Grund werden in der Entwurfsphase kontinuierlich parallele Bearbeitungsstände verschiedener Teammitglieder fast ausschließlich manuell zu einem gemeinsamen Modell integriert. Diese manuelle Arbeitsweise ist wegen der Größe und Komplexität der zu verwaltenden Modelle einerseits sehr arbeits- und deswegen zeitaufwendig und andererseits fehleranfällig.

Durch eine geeignete Werkzeugunterstützung kann die kollaborative Arbeitsweise in der Softwaremodellierung stark unterstützt werden. Mit der vorliegenden Dissertation wird ein rechnergestütztes Verfahren zur Integration parallel erstellter Modellversionen konzipiert. In der vorgestellten Lösung lassen sich zwei Schwerpunkte ausmachen. Zum einen wird die Integration asynchron durchgeführter Modelländerungen auf Basis syntaktischer Elementarbausteine automatisiert, wodurch ein manuelles Einpflegen von Modelländerungen entfallen kann. Zum anderen wird die so integrierte Modellsyntax hinsichtlich ihrer Korrektheit validiert. Im Rahmen dieser Prüfung wird untersucht, ob das Modell die syntaktischen Korrektheitsanforderungen des vorgegebenen Metamodells erfüllt und ob weitere, für Modelle allgemeingültige Eigenschaften gelten. Eine dadurch mögliche automatische Lokalisierung von inkonsistenten parallelen Änderungen erspart den Modellierern eigene Analyseaufwände im Rahmen der letztendlich manuellen Korrektur inkonsistenter Änderungen.

Das konzipierte Verfahren beschleunigt die kollaborativen Modellierungsaufgaben, indem die bisher aufwendige manuelle Integration parallel entwickelter Modellversionen weitgehend automatisiert wird. Es ermöglicht den Aufbau einer werkzeuggestützten Modelldaten-verwaltung für Entwicklerteams, die Modelle beliebiger Modellierungssprachen verarbeiten kann.

Danksagung

Das wissenschaftliche Umfeld, in dem diese Dissertation entstanden ist, hat entscheidend zu ihrem Gelingen beigetragen. Es ermöglichte durch eine offene und konstruktive Diskussionskultur auf einem hohen fachlichen Niveau die Entwicklung kreativer Ideen und belastbarer Lösungen. Mein besonderer Dank gilt den Personen, die dieses Umfeld gestaltet haben. Die Veröffentlichung meiner Dissertation möchte ich deswegen als Anlass nutzen, mich bei Prof. Dr. Andreas Rausch, allen aktuellen und ehemaligen Mitarbeitern des Lehrstuhls Software Systems Engineering zu bedanken.

Für die konstruktive Auseinandersetzung mit meiner Arbeit, für die hilfreichen Kommentare und für die Erstellung des Zweitgutachtens gilt mein Dank Prof. Dr. Manfred Broy.

Insbesondere bedanke ich mich bei Prof. Dr. Rausch und Dr. Edward Fischer für die intensiven und mit viel Engagement geführten Fachgespräche, die sich aus der detaillierten Auseinandersetzung mit meinem Forschungsthema und den Vorabversionen der Dissertation ergaben. Die fachlichen Diskussionen, die immer das Ziel verfolgten, den Dingen auf den Grund zu gehen, haben mir viel Freude bereitet und werden dauerhaft mein Bild von anspruchsvoller Forschungsarbeit positiv prägen.

Daniel Dyballa möchte ich danken für die erfolgreiche Implementierung des entstandenen Softwareprototyps.

Nicht zuletzt gilt mein Dank meiner Familie, meinen Freunden und Kollegen, die mich durch Humor, Motivation und Verständnis unterstützt haben und so einen unverzichtbaren Ausgleich zur Promotionsarbeit ermöglicht haben.

Inhaltsverzeichnis

I Die Dissertation.....19

1 Ausgangssituation.....19

1.1 Kooperationen in Unternehmensnetzwerken –
Trend zu standortübergreifender, parallelisierter Softwareentwicklung.....20

1.2 Modellbasierte Entwicklung –
Trend zu präziserer Modellbildung in frühen Softwareentwicklungsphasen. . .20

1.3 Kombination von Trends –
Parallel, asynchrone Kollaboration in der Softwaremodellierung.....21

1.4 Stand der verteilt, parallelen Bearbeitung von Softwareartefakten.....23

2 Problembeschreibung.....23

3 Ziele der Arbeit.....26

4 Vision – Kollaborative Modellierungsumgebung.....28

4.1 Rahmenbedingungen.....29

4.2 Einbettung in Forschungs- und Anwendungslandschaft.....29

5 Beitrag, Bedeutung und Innovationscharakter der Lösungskonzeption 30

6 Verwandte Arbeiten.....31

6.1 Software Configuration Management (SCM) –
Konzepte und Technologien.....32

6.2 Versionskontrolle von Modellen.....32

6.3 Konsistenzanalyse paralleler Modelländerungen.....33

7 Überblick und Struktur der Arbeit.....34

II Grundlagen....37

1 Kollaboration im Software Engineering....37

1.1 Aktueller Stand der kollaborativen Softwareentwicklung....39

1.2 Software Configuration Management (SCM)....39

2 Versionsverwaltung im Software Engineering....40

2.1 Basisterminologie....40

2.2 Inhaltliche Synchronisierung von Arbeitsergebnissen im Team....43

2.2.1 Lock-Modify-Write....43
2.2.2 Copy-Modify-Merge....43
2.2.3 Merge-Verfahren in der Versionsverwaltung....45
2.2.4 3-Wege-Merge-Prinzip....46
2.2.5 Zustandsbasierte Merge-Verfahren....48
2.2.6 Operationsbasierte Merge-Verfahren....48
2.2.7 3-Wege-Merge für Textdaten – Text-Merge-Verfahren....50

2.3 Aktueller Stand der SCM-Werkzeugunterstützung....51

2.3.1 Zentrale Versionsverwaltung....51
2.3.2 Verteilte Versionsverwaltung....52

3 Modellbasierte Softwareentwicklung....52

3.1 Modellbasierte Entwicklung von Software Systemen....53

3.1.1 Modellierungssprachen in der Softwareentwicklung....53
3.1.2 Universelle Standards versus domänenspezifische Modellierung....55

3.2 Automatische Validierung der Korrektheit von Software durch statische Analyse ihrer formalen Syntax....60

3.2.1 Verifikation versus Validierung....60
3.2.2 Statische Analysen zur Validierung von Programmen und Softwaremodellen....60

3.3 Spezifikation von Modellierungssprachen im Software Engineering....62

3.3.1 Statische Analyse abstrakter Syntaxgraphen von Modellen auf Basis von MOF-Metamodellspezifikationen....64
3.3.2 Schichten-Architektur der MOF....65
3.3.3 Validierung der statischen Semantik von Modellen durch OCL-Constraints im Metamodell....68

III Synchronisierung kollaborativ entwickelter Softwaremodelle.....69

1 Synchronisierung von Teamarbeit in der Quellcodeverwaltung – eine Erfolgsgeschichte.....70

1.1 Teamkoordination mittels pessimistischer Versionsverwaltung.....70

1.2 Teamkoordination mittels optimistischer Versionsverwaltung.....71

2 Analyse des Merge-Verfahrens für Quellcode aus Anwendersicht.....73

2.1 Integration von Quellcodeänderungen mit Text-Merge-Verfahren.....73

2.2 Merge-bedingte Validierung in integrierten Quellcodeversionen.....74

3 Kollaborative Softwaremodellierung in der Analysephase – ein Illustrationsbeispiel.....79

3.1 Ein Metamodell für die Anforderungsanalyse.....79

3.2 Kollaborative Modellierung einer Kontenverwaltung.....81

4 Ein Merge-Verfahren für Softwaremodelle.....87

4.1 Scheitern der Integration asynchroner Modelländerungen mit Text-Merge-Verfahren.....87

4.2 Intuitive Integration logischer Modelländerungen.....90

4.3 Konzept eines morphologischen Merge-Verfahrens für Modelle auf Basis syntaktischer Zerlegung.....93

4.4 Konzept der automatischen Validierung integrierter Modellversionen durch statische Analyse.....96

4.4.1 Merge-bedingte Verletzung der MOF-Syntaxspezifikation von Modellen.....96

4.4.2 Merge-bedingte Verletzung der statischen Semantik von Modellen außerhalb der MOF-Syntaxspezifikation.....100

4.4.3 Merge-bedingte Anomalien in Modellen.....104

4.5 Fazit: Die werkzeuggestützte Versionsverwaltung von Modellen erfordert neue Lösungsansätze.....105

IV Korrekte Integration asynchroner Modelländerungen.....107

1 Ein morphologisches Merge-Verfahren für Modelle.....108

1.1 Identifizierung generischer Änderungen zwischen Modellversionen.....108

1.1.1 Morphologische Zerlegung von Modelldaten.....109
1.1.2 Schema der morphologischen Modellrepräsentation.....110
1.1.3 Transformation von Modelldaten im XMI-Format in eine morphologische Modellrepräsentation.....117
1.1.4 2-Wege-Diff für MMR-Versionen.....120

1.2 3-Wege-Merge-Prinzip für MMR-Versionen.....122

1.2.1 Ein Schema zur Repräsentation der asynchronen Evolution von Dokumenten.....122
1.2.2 Integration asynchroner durchgeführter Änderungen – Semantik eines 3-Wege-Merge-Verfahrens.....125

2 Automatisierung statischer Analysen zur merge-bedingten Modellvalidierung.....129

2.1 Überblick: Merge-bedingte Validierung von Modellen in MMR.....130

2.2 Beschreibungslogiken, OWL und die Anfragesprache SPARQL.....134

2.2.1 Die Beschreibungslogik SHOIN.....135
2.2.2 Inferenzservices.....136
2.2.3 Offene- versus Geschlossene-Welt-Semantik.....137
2.2.4 Syntax für Beschreibungslogiken – Die Web Ontology Language (OWL).…137
2.2.5 Inferenzwerkzeuge für Beschreibungslogiken.....138
2.2.6 SPARQL und Konjunktive Anfragen zur Informationsanalyse in Wissensbasen.....138
2.2.7 Aktuelle Ansätze zum Einsatz von Beschreibungslogiken zur Validierung von Modellen.....140

2.3 Alternative 1: Statische Analyse durch ABox-Reasoning mit OWL im Rahmen der Modellvalidierung.....141

2.3.1 Transformation von Metamodellen und Modellen nach OWL.....141
2.3.2 Statische Analyse durch Konsistenzprüfung von OWL-Wissensbasen.……150
2.3.3 Einschränkungen der statischen Modellanalyse durch Konsistenzprüfung im Rahmen des ABox-Reasonings.....152

2.4 Alternative 2: Statische Analyse durch SPARQL-Anfragenauswertung im Rahmen der Modellvalidierung.....153

2.5 Statische Analyse zur Anomalieerkennung durch TBox-Reasoning 156

2.6 Auflösung inkonsistenter Änderungen / Wiederherstellung syntaktischer Korrektheit....160

V Implementierung und Konzeptbewertung....163

1 Team.Mode – Automatisierung der Versionsverwaltung von Softwaremodellen....164

1.1 Arbeiten mit Team.Mode-Projekten....164

1.2 Kollaboratives Modellieren mit Team.Mode....169

2 Werkzeuggestütze Analyse Merge-bedingter Integrationsprobleme....173

3 Konzeptbewertung und Fazit....176

VI Zusammenfassung und Perspektiven....179

1 Zusammenfassung der Ergebnisse....179

2 Anwendungsgebiet und Nutzen....180

3 Lösungsalternativen und Erweiterungspotenzial....181

3.1 Lösungsalternativen....181

3.1.1 Verstärkung synchroner Modellierungsarbeit....181
3.1.2 Operationsbasierte Synchronisation von Modellversionen....181
3.1.3 Einsatz von Sperrmechanismen....182

3.2 Erweiterungsmöglichkeiten....182

3.2.1 Erweiterungspotenzial bei der Synchronisierung der visuellen Modelldarstellung....182
3.2.2 Erweiterungspotenzial bei der visuellen Darstellung inkonsistenter Änderungen....183
3.2.3 Erweiterungspotenzial bei der Konsistenzanalyse paralleler Änderungen....183
3.2.4 Unterstützung der Co-Evolution von Metamodellen und Modellen....183
3.2.5 Einbettung in ganzheitliche Entwicklungsumgebungen....183

4 Perspektiven – Entwicklung von Modellen im Team....184

Bibliografie....187

Abkürzungsverzeichnis

ABox	(engl. assertional box) Faktenwissen in einer beschreibungslogischen Wissensbasis (vgl. Abschnitt IV.2.2)
BSI-Model	Modell der BankSoftwareInfrastructure, Beispielmodell (vgl. Abschnitt III.3.2)
CWA	(engl. closed world assumtion) Geschlossene-Welt-Annahme in logischen Systemen(vgl. Abschnitt IV.2.2)
DL	(engl. description logic) Beschreibungslogik (vgl. Abschnitt IV.2.2)
DSL	(engl. domain specific language) domänenspezifische Sprache (vgl. Abschnitt II.3.3)
ECore	Technische Umsetzung der Meta-Metamodellierungssprache EMOF (vgl. Abschnitt II.3.3)
EMF	Eclipse Modeling Framework (vgl. Abschnitt II.3.3)
ID	Identifizierer
m	Indizes für Modell
$\hat{m}$	Indizes für Metamodell
MDU	Morphologische Dateneinheit (vgl. Abschnitt IV.1.1.2)
MMR	Morphologische Modellrepräsentation(vgl. Abschnitt IV.1.1.2)
MOF	(engl. Meta Object Facility) Meta-Metamodellierungssprache (vgl. Abschnitt II.3.3)
OCL	(engl. Object Constraint Language) Constraint-Sprache (vgl. Abschnitt II.3.3)
OWL	(engl. Web Ontology Language)
RE-Metamodel	Requirements Engineering Metamodell (vgl. Abschnitt III.3.1)
SCM	(engl. software configuration management) Softwarekonfigurationsverwaltung (vgl. Abschnitt II.1.2)
SPARQL	SPARQL Protocol RDF Query Language (vgl. Abschnitt IV.2.2)
TBox	(engl. terminological box) Domänenwissen in einer beschreibungslogischen Wissensbasis (vgl. Abschnitt IV.2.2)
UML	(engl. Unified Modeling Language) Standard zu Software- und Systemmodellierung (vgl. Abschnitt II.3.3)
VO, VA, VB, VM	Abk. für relevante Versionen eines 3-Wege-Merges: VO – Ausgangsversion, VA – Version von Alice, VB – Version von Bob, VM – gemergte Version aus VA und VB
XMI	(engl. XML Model Interchange) Austauschdatenformat für Modelle im Software Engineering (vgl. Abschnitt II.3.3)

I Die Dissertation

Die kollaborative Modellierung von Softwaresystemen in verteilten Teams gewinnt in der Analyse- und Entwurfsphase des Software Engineerings eine immer größere Bedeutung. Allerdings steckt in diesem Bereich die Entwicklung einer geeigneten methodischen und technischen Unterstützung der verteilt, parallelen Arbeitsweise im Vergleich zur Programmierung noch in den Kinderschuhen. In den folgenden Abschnitten dieses Kapitels werden wesentliche Herausforderungen der kollaborativen Softwaremodellierung erörtert, bevor einige zentrale Problemstellungen herausgegriffen und im Detail betrachtet werden. Aus diesen werden in Abschnitt 3 konkrete Ziele für die Dissertation formuliert. Zum Abschluss dient Abschnitt 7 als inhaltlicher Wegweiser der Dissertation, indem deren Struktur erläutert und eine inhaltliche Zusammenfassung aller Teile gegeben wird.

1 Ausgangssituation

Softwaresysteme bestimmen inzwischen direkt oder indirekt fast alle Bereiche unseres Lebens. Dabei nimmt die jetzt schon enorme Komplexität dieser Systeme aufgrund steigender Vernetzung und Integration einerseits und ihrer zunehmenden Größe und Dynamik andererseits stetig zu. Gleichzeitig steigen die Ansprüche an Langlebigkeit und Fehlerfreiheit, da der Einsatz von Softwaresystemen verstärkt in immer sicherheitskritische Bereiche vordringt.

Diese Situation erfordert ein hohes Qualitätsbewusstsein der Softwareingenieure bei der Entwicklung und Wartung dieser Systeme. Die Beherrschung der enormen Komplexität von Softwaresystemen in Entwicklung und Wartung unter verstärkter Berücksichtigung von Verlässlichkeitsanforderungen stellt in den nächsten Jahren eine der größten Herausforderungen im Gebiet des Software Engineerings dar.

Um den genannten Herausforderungen zu begegnen, kann man schon in den frühen Entwicklungsphasen – Anforderungsanalyse und Softwareentwurf – einen Trend zu immer präziserer Modellbildung erkennen.

Zusätzlich erschwert wird eine zeit- und qualitätsbewusste Softwaremodellierung durch die verstärkt standortübergreifende Verteilung der entwickelnden Softwareingenieure und Anforderungssteller (Bartelt u. a. 2009).

1.1 Kooperationen in Unternehmensnetzwerken – Trend zu standortübergreifender, parallelisierter Softwareentwicklung

Aufgrund der fortschreitenden Globalisierung von Wirtschaft und Gesellschaft kann man insbesondere in der IT-Branche eine stetig steigende Internationalisierung von Unternehmen, Unternehmensnetzwerken und Kundenbeziehungen beobachten.

Diese Entwicklung beeinflusst auch die Organisation der Softwareentwicklung nachhaltig. Wegen der regional und weltweit differenzierten Verteilung von spezifischen Entwicklungskompetenzen und wirtschaftlichen Entwicklungsressourcen aber auch aufgrund internationaler Entwicklungskooperationen und Kundenorientierung werden Softwaresysteme häufig in geografisch verteilten Teams entwickelt.

Die erfolgreiche Berücksichtigung der hohen Erwartungen an Entwicklungszeit und Qualität der Ergebnisse wird dabei stark durch die Art der Koordinierung solcher Teams beeinflusst. Hierbei spielt immer mehr eine kontinuierliche und angemessene inhaltliche Integration von asynchron und parallel erstellten Softwareprodukten eine entscheidende Rolle.

Im Forschungsfeld des Software Engineerings wird weltweit an Lösungen gearbeitet, um die Herausforderungen durch die sich ändernden Anforderungen an Softwaresysteme und neue Organisationsstrukturen in der Softwareentwicklung zu bewältigen. Die geänderten Organisationsstrukturen in Softwareentwicklungsprojekten erfordern ein hohes Maß an zeitlich paralleler, kollaborativer Entwicklungsarbeit innerhalb verteilter Teams insbesondere auch in den frühen Entwicklungsphasen.

1.2 Modellbasierte Entwicklung – Trend zu präziserer Modellbildung in frühen Softwareentwicklungsphasen

Gerade in den kreativen Entwurfsphasen spielen intuitiv verständliche, meist grafische Modellierungstechniken für die Kommunikation der Softwareingenieure eine wichtige Rolle. So wurden syntaktisch formale Modellierungssprachen entwickelt, die eine Erstellung von Softwaremodellen aus einer ausgewogenen Mischung von grafischen Symbolen und textuellen Ausdrücken erlaubt.

Auf Grundlage perspektivenbezogener Sichten können Teams von Softwareingenieuren Disziplin übergreifend ihre Entwürfe im Modell festhalten, um so eine gemeinsame Kommunikationsbasis zur Abstimmung einzelner Systemteile zu erhalten. Darüber hinaus werden intuitive, grafische Modelle zur Dokumentation von Softwaresystemen eingesetzt, wodurch die fortlaufende Weiterentwicklung sowie die Wartung und Pflege stark vereinfacht wird.

Bis heute sind in der Praxis der Softwareentwicklung Beschreibungen informell als Texte oder Grafiken dokumentiert. Erste Ansätze zur Definition standardisierter, grafischer Modellierungssprachen Anfang der 90iger Jahren beschränkten sich auf die Festlegung einer

grafischen Universalsyntax zur Software- und Systemmodellierung – der *Unified Modeling Language* (*UML*) (Booch, Rumbaugh und Jacobson 1998).

Seit Ende der 90iger Jahre wurden die Initiativen zur Formalisierung der UML-Syntax kontinuierlich verstärkt. Inzwischen veröffentlichte die *Object Management Group (OMG)* die Version 2.2 dieser universellen Modellierungssprache zur System- und Softwaremodellierung.

Allerdings kann eine universelle Modellierungssprache wie die UML nur einen Teil des Bedarfes nach geeigneten Beschreibungstechniken im Software Engineering abdecken. So werden in spezifischen Domänen wie beispielsweise in der Geschäfts- und Produktionsprozessautomatisierung sowie in der Verfahrenstechnik weitere, speziell auf die jeweiligen Bedürfnisse angepasste, Beschreibungstechniken verwendet.

Die syntaktische Spezifikation dieser domänenspezifischen Modellierungssprachen (*Domain Specific Language – DSL*) wird entweder mithilfe entsprechender Profile zur „leichtgewichtigen" Anpassung der durch die UML standardisierten Syntax oder auf Basis ganz eigener Sprachspezifikationen festgelegt. Zur Unterstützung und Standardisierung des Entwurfes domänenspezifischer Modellierungssprachen bietet die OMG seit 2002 den Sprachstandard *Meta Object Facility* (*MOF*) an, mit dem die Spezifikation von Metamodellen beliebiger Modellierungssprachen vereinfacht und vereinheitlicht wird.

Mithilfe eines MOF-basierten Metamodells lässt sich die abstrakte Syntax einer Modellierungssprache formal und maschinenverarbeitbar beschreiben. Die Formalisierung durch ein Metamodell ermöglicht somit eine automatische Validierung von Modellen durch statische Analyse ihrer syntaktischen Repräsentation. Ziel dabei ist die Prüfung der Korrektheit des Modells hinsichtlich der Sprachspezifikation des Metamodells.

Durch eine entsprechende Werkzeugunterstützung kann eine aufwendige und fehleranfällige manuelle Prüfung der syntaktischen Korrektheit eines Modells vermieden bzw. stark verringert werden. Gerade bei der automatischen Synthese eines integrierten Modells aus mehreren parallelen Bearbeitungszuständen kann die syntaktische Korrektheit eines Modells umbemerkt verletzt werden.

Neben dem Aufwand einer manuellen Analyse der Modellsyntax nach einer solchen Integration können Fehler im automatisch integrierten Modell leicht übersehen werden. Sowohl die Effizienz der Entwicklung als auch die Qualität der Softwaremodelle profitieren somit automatischen Analysemöglichkeiten.

1.3 Kombination von Trends – Parallel, asynchrone Kollaboration in der Softwaremodellierung

Wie erwähnt, werden große, komplexe Softwaresysteme heute in global verteilten Teams entwickelt. Zur Beschleunigung der Entwicklungszeit werden Analyse- und Entwurfsaktivitäten auf geografisch verteilte Teams aufgeteilt und parallel bearbeitet. Abbildung I.1 stellt diese Situation überblicksartig dar.

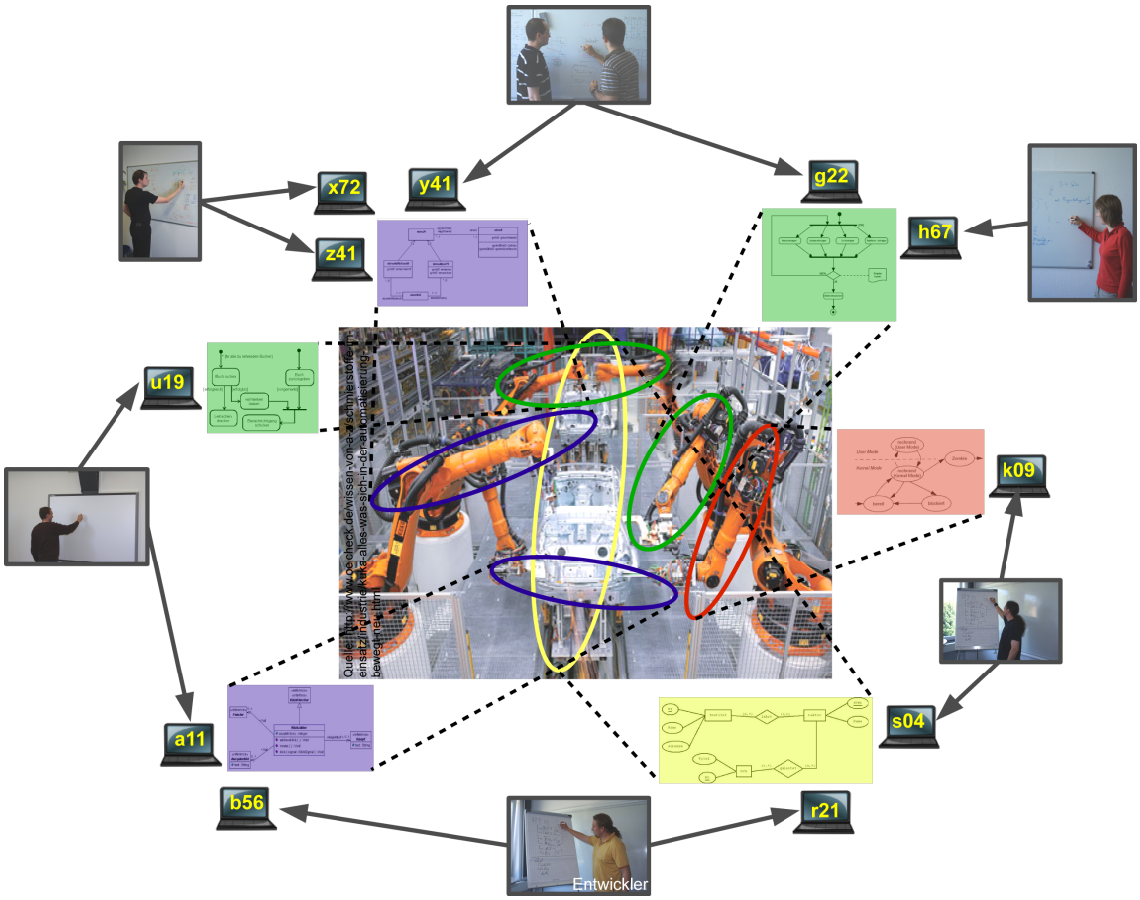


Abbildung I.1. Überblick Ausgangssituation

Obwohl die Arbeit der Modellierer eines Entwicklungsteams zum Teil einen unterschiedlichen inhaltlichen Fokus besitzt, so arbeiten doch alle gleichzeitig am gleichen Modell. Parallel und oft asynchron erstellen sie kontinuierlich neue Arbeitsversionen des Modells, i. d. R. unterstützt durch eine Versionsverwaltung.

Verfahren und Werkzeuge, die Entwicklerteams in der Verwaltung ihrer Softwareprodukte unterstützten, werden dem Sammelbegriff des *Software Configuration Managements* (*SCM*) untergeordnet. Asynchron erstellte Inhalte einzelner Modellversionen werden in der Praxis durch die beteiligten Teams kontinuierlich inhaltlich synchronisiert, um Missverständnisse bezüglich der Zielsetzung und Widersprüchlichkeiten zwischen Modellteilen frühzeitig zu vermeiden.

Ziel ist es, vor Beginn der Programmierung einen vollständigen und korrekt integrierten Softwaresystementwurf fertigzustellen, da während der Programmierung identifizierte Fehler im zugrundeliegenden Entwurf enorme Aufwände bei deren dann notwendigen Behebung nach sich ziehen können.

1.4 Stand der verteilt, parallelen Bearbeitung von Softwareartefakten

Aktuelle und in der Praxis gut erprobte SCM-Werkzeuge unterstützen schon heute die Integration von Codeversionen während der asynchronen Softwareprogrammierung. Dabei unterscheidet man zwischen pessimistischen und optimistischen Verfahren.

Bei pessimistischen Verfahren wird einem Bearbeiter das exklusive Recht eingeräumt, ein Dokument zu ändern. In der Praxis handelt es sich dabei um einzelne Dateien, Verzeichnisse oder Datenbankeinträge. Aufgrund der Bearbeitungseinschränkungen wird das Risiko verringert, dass mehrere Bearbeiter gleiche oder stark voneinander abhängige Teile parallel auf verschiedene Weise ändern und so inhaltliche Abhängigkeiten zwischen einzelnen Informationsartefakten verletzen.

Auf Basis negativer Erfahrungen mit pessimistischer Versionierung wegen fehlender Bearbeitungsflexibilität wurden optimistische SCM-Verfahren eingeführt. Diese haben bis heute eine starke Verbreitung gefunden und werden sehr erfolgreich in global verteilten Projekten der Open-Source-Programmierung eingesetzt.

Beim Einsatz optimistischer Versionierung wird auf das Sperren bestimmter Inhalte verzichtet, sodass insbesondere auch eine parallele Bearbeitung einzelner Dateien ermöglicht wird. Besondere Vorteile bieten diese Freiheiten bei der parallelen Bearbeitung von digitalen Texten, beispielsweise von Quellcodedateien bei der Programmierung.

Für einfachen Text und Codedateien werden durch aktuelle SCM-Systeme automatische Mechanismen zur Synchronisierung parallel geänderter Inhalte angeboten. Diese unterstützen die Bearbeiter bei der Zusammenführung ihrer unabhängigen Änderungen, indem unabhängige, parallele Änderungen von Textzeilen zusammen in eine neue Nachfolgeversion eingepflegt werden. Im Gegensatz zu der Evolution von Programmcode gibt es zur feingranularen Verwaltung kollaborativ entwickelter Softwaremodelle bisher keine vergleichbar praktikablen Verfahren.

2 Problembeschreibung

Im letzten Abschnitt wurde sowohl das Umfeld, indem heutige Softwareentwicklungsprojekte abgewickelt werden, beleuchtet als auch vorhandene Lösungsansätze in zwei wesentlichen Bereichen – modellbasierte Entwicklung und Software Configuration Management (SCM) – erörtert.

Dieser Abschnitt geht auf die Differenz zwischen Anspruch und Wirklichkeit in der Softwareentwicklung ein. Dabei steht der Bereich des modellbasierten Softwareentwurfs im Fokus. Für diesen Bereich werden konkrete Problemstellungen benannt, deren Lösung eine noch immer ernst zu nehmende Herausforderung darstellt.

Dazu werden zunächst die beschriebenen Rahmenbedingungen und die zur Verfügung stehenden Lösungskonzepte für den (kollaborativen) Softwareentwurf gegenübergestellt. Aus diesem Vergleich wird ein grundlegender und bisher ungelöster Problembereich herausgestellt und erläutert.

Die im letzten Abschnitt genannten Rahmenbedingungen für Softwareprojekte lassen auf die folgende Dimensionen abbilden:

i. Steigende Komplexität und Größe der zu entwickelnden Systeme

ii. Steigende Anforderungen an die Verlässlichkeit der zu entwickelnden Systeme

iii. Parallel, asynchrone Arbeitsweise von Softwareingenieuren und Kunden

iv. Steigende Anforderungen an die Effizienz des Softwareentwicklungsprozesses

Um in diesem Entwicklungsumfeld die Entwurfsphase von Softwareprojekten erfolgreich gestalten zu können, werden in Wissenschaft und Industrie eine Reihe von Methoden und Technologien erforscht und erprobt. Hierbei sind einerseits Konzepte zur modellbasierten Softwareentwicklung und andererseits Konzepte zur Unterstützung der verteilt und parallelen Entwicklungsarbeit von Software-Engineering-Teams zu nennen.

In Forschung und Praxis der modellbasierten Softwareentwicklung spielen Beschreibungstechniken eine große Rolle, die einerseits verschiedene *Abstraktionsebenen* berücksichtigen, um die steigende Komplexität und Größe (vgl. i.) von Softwaresystemen zu berücksichtigen. Andererseits wird auch schon in den frühen Entwurfsphasen eine *Formalisierung der Softwarebeschreibung* (bspw. auf Basis formal fundierter Modellierungssprachen) angestrebt, um ein Höchstmaß an Verlässlichkeit (vgl. ii.) zu garantieren.

Mit standardisierten (Meta-) Modellierungssprachen wie UML, MOF, SysML, Architekturbeschreibungssprachen oder mit DSLs stehen (semi-)formale Beschreibungsmittel bereit um Softwareentwürfe auf unterschiedlichen Abstraktionsebenen möglichst präzise darzustellen.

Besonders die *Integration paralleler Entwicklungsstände* von Softwareprodukten, die aufgrund von Verteilung und Parallelisierung der Entwurfsarbeit entstehen, bildet eine ambitionierte Herausforderung in der Softwareentwicklung. Dieser hat sich das Teilgebiet der Versionsverwaltung im Rahmen des SCM verschrieben (MacKay 1995; Okubo u. a. 1998). Dabei hat sich bis heute weitgehend die Philosophie der optimistischen Versionsverwaltung durchgesetzt.

Durch den stetigen Ausbau der *Werkzeugunterstützung* wird sowohl im Bereich der modellbasierten Softwareentwicklung als auch im Bereich des SCM die Effizienz des Softwareentwicklungsprozesses gesteigert. Hierbei sind im Bereich der Softwaremodellierung sogenannte Case-Tools wie Rational Rose Modeler (IBM 2010a), MagicDraw (NoMagic Inc. 2010a), Borland Together (Borland 2010a) oder Poseidon (Gentleware 2010) weit verbreitet.

Auch in der automatischen Versionsverwaltung hat sich in der Softwareentwicklung eine vielfältige Werkzeugpalette etabliert. Sie reicht von reinen Versionsverwaltungssystemen wie CVS (GNU Project 2010b), Subversion (Tigris.org 2010) oder Bitkeeper (BitMover Inc. 2010) bis hin zu umfangreichen Kollaborationsplattformen wie IBM Jazz Team Concert (IBM 2010b) oder Microsoft TFS (Microsoft 2010).

Bis dato konnten die beiden Welten – Softwaremodellierung und Software Configuration Management (SCM) – leider nicht im Rahmen einer praktikablen Werkzeugunterstützung integriert werden. Aktuelle SCM-Werkzeuge konzentrieren sich fast ausschließlich auf die

Quellcodeverwaltung und sind im Bereich der Programmierung unverzichtbar geworden. Arbeiten hingegen mehrere Bearbeiter parallel an dem gleichen Modell, so können sie keine adäquate optimistische Versionsverwaltung der Modelldaten nutzen.

Hierin zeigt sich ein Hauptproblem, das im Bereich der kollaborativen Softwaremodellierung bisher ungelöst ist:

Die optimistische Versionsverwaltung von Softwaremodellen wird bisher nicht hinreichend durch Werkzeuge unterstützt.

Dabei sind die Anforderungen an eine solche optimistische Versionsverwaltung für Modelle im wissenschaftlichen und industriellen Diskurs durchaus formuliert worden und ähnlich gelagert wie die Bedürfnisse bei der Versionsverwaltung von Quellcode (Brunet u. a. 2006; Bendix und Emanuelsson 2009a): Parallel durchgeführte Modelländerungen sollen mit einem sogenannten optimistischen Merge-Verfahren zu einer Modellversion integriert werden.

Da automatisch integrierte Modellversionen möglicherweise inkorrekt in Bezug auf das Metamodell der Modellierungssprache sein können, bedarf es einer anschließenden Validierung. Sofern die Modellierungssprache eine formale und maschinenverarbeitbare Syntaxspezifikation vorgibt, kann die erforderliche statische Analyse ebenfalls automatisiert werden.

Automatisierung der Integration parallel durchgeführter Modellbearbeitungen. Zur Verringerung der Entwicklungszeit versucht man, durch eine geeignete Arbeitsorganisation, die Entwicklungstätigkeit von Softwareingenieuren so weit wie möglich zu parallelisieren. In der Realisierungsphase werden zur Synchronisierung parallel bearbeiteter Programmcodeversionen oft optimistische Verfahren aus dem Software Configuration Management (SCM) eingesetzt. Diese Verfahren implementieren zur Zusammenführung von parallelen Änderungen ein 3-Wege-Merge-Konzept (Mens 2002). Syntaktische Beziehungen zwischen den in einer Textdatei gespeicherten Daten, die insbesondere in Softwaremodellen stark vernetzt sind, werden bei diesem Verfahren nicht berücksichtigt. Wie in Kapitel III ausführlich beschrieben wird, lassen sich die aktuell vorhandenen SCM-Verfahren nicht praktikabel zur Integration paralleler Modellbearbeitungen anwenden. Der Grund sind massive Verletzungen des Datenformates zur Modellspeicherung (i. d. R. XML oder proprietäre Binärformate) durch die verwendeten Text-Merge-Ansätze. Wie gezeigt werden wird, eignen sich diese Ansätze grundsätzlich nicht, um kollaborativ bearbeitete Modelle zu verwalten. Durch das Auftreten syntaktischer Fehler lassen sich Modelldateien nicht mehr durch entsprechende Editoren korrekt parsen und damit nicht mehr visuell darstellen. Eine manuelle Identifizierung, Analyse und Behebung dieser verfahrensbedingten Probleme wäre somit nicht anhand der konkreten, den Modellierern bekannten (meist grafischen) Syntax möglich. Dies ist bei hinreichend großen und komplexen Modellrepräsentationen unrealistisch aufwendig. In der Praxis schränkt man deswegen oft parallele Bearbeitungsprozesse von Softwaremodellen stark ein, verzichtet durch eine sequenzielle Arbeitsweise sogar ganz darauf oder integriert mit erheblichen Mehraufwänden parallel Modellbearbeitungen manuell. Obwohl sich durch geeignete Verfahren zum Zusammenführen parallel bearbeiteter Versionen, welche die formale Syntax von Modellen besser berücksichtigen, die Parallelisierbarkeit der Entwurfsarbeit wesentlich steigern lässt, wurden in Praxis bisher nur wenige sehr spezialisierte Lösungsansätze experimentell eingesetzt.

Automatisierung der Erkennung inkonsistenter Modelländerungen durch statische Analysen. Durch die parallele, inhaltlich unabhängige Bearbeitung von Modellierungsständen ist das Risiko einer Verletzung der syntaktischen Korrektheit integrierter Modelle besonders hoch. Hierbei können die durch die Modellierungssprache (i. d. R. durch ein sog. Metamodell) vorgegebenen, teils sehr komplexen Korrektheitsbedingungen an die Modellsyntax bei der syntaktischen Integration paralleler Modelländerungen z. T. nicht eingehalten werden. So existiert oft eine Vielzahl von Abhängigkeiten zwischen syntaktischen Bausteinen parallel bearbeiteter Modellteile, die leicht durch die asynchrone Arbeitsweise unberücksichtigt bleiben. Zur frühzeitigen Erkennung und Vermeidung von Fehlern, die auf solche Inkonsistenzen zwischen parallelen Änderungen zurückzuführen sind, werden Modellbearbeitungsstände regelmäßig inhaltlich synchronisiert. Ohne technische Unterstützung ist diese Synchronisierung aber sehr aufwendig. Die verstärkten Formalisierungsbestrebungen bei der Syntaxspezifikation von Modellierungssprachen ermöglichen allerdings oft den Einsatz automatischer Verfahren zur statischen Analyse der Modellsyntax. Dabei werden Modelle hinsichtlich der im Metamodell spezifizierten Syntax und statischen Semantik geprüft. Eine aufwendige und fehleranfällige manuelle Korrektheitsanalyse von integrierten Modellversionen kann damit weitestgehend entfallen. Modellierungswerkzeuge integrieren heutzutage häufig Validierungsmechanismen, um Modellierer durch die angesprochenen automatischen Prüfungen zu unterstützen. Allerdings konzentrieren sich diese Prüfungen *nicht* auf die Analyse von Fehlern, die bei der Integration paralleler Bearbeitungsstände von Modellen entstehen können.

3 Ziele der Arbeit

Der letzte Abschnitt hat zwei Problembereiche in der kollaborativen Modellierung von Softwaresystemen aufgezeigt – Fehlende Automatisierung in der Integration paralleler Modelländerungen und in der Erkennung inkonsistenter Modelländerungen. Aus diesen werden im Folgenden zwei zentrale Ziele der in den nachfolgenden Kapiteln dokumentierten Arbeit abgeleitet.

Softwareingenieuren stehen für die Erstellung grafischer Modelle zahlreiche Modellierungswerkzeuge und für die Versionsverwaltung etablierte SCM-Systeme zur Verfügung. Wie im letzten Abschnitt beschrieben, berücksichtigen aktuelle optimistische SCM-Verfahren nicht die spezifische syntaktische Struktur von Modellen, was zum Scheitern einer effektiven Integration paralleler Modellbearbeitungen führt.

Obwohl sich auf Basis der formalen Syntax vieler Modellierungssprachen die Integration von Modellversionen automatisieren ließe, existieren dafür zurzeit keine praktikablen SCM-Werkzeuge. Um diese Lücke zu schließen, ist die Konzipierung eines automatischen Verfahrens zur *automatischen Integration paralleler Modellbearbeitungen auf Basis syntaktischer Modellbausteine* ein Ziel der Arbeit. Dabei sollen etablierte Ansätze wie das anerkannte 3-Wege-Merge-Prinzip auf die syntaktische Änderungsintegration übertragen werden.

Das zweite Ziel ist die Konzeption einer *automatischen Unterstützung zur Validierung der Modellsyntax durch statische Analyse*, die mit dem erwähnten Verfahren syntaktisch integriert wurden. Diese Analyse bildet die Grundlage, um inkonsistente Änderungen in parallelen Bearbeitungsständen eines Modells zu erkennen.

Die angesprochene statische Analyse der Korrektheit auf Basis der Modellsyntax schließt dabei Prüfungen ein, die folgende Fragen beantworten:

- Erfüllt die Modellsyntax die durch das Metamodell der Modellierungssprache vorgegebenen syntaktischen Korrektheitsbedingungen?
  - Ist der abstrakte Syntaxgraph des zu untersuchenden Modells korrekt im Sinne des Metamodells? (in Analogie zu Programmiersprachen schließt das die Prüfung der kontextfreien Sprachspezifikation – dem sog. Syntaxbaum – und bestimmter kontextsensitiver Nebenbedingungen, die häufig unter dem Begriff statische Semantik zusammengefasst werden, ein)
  - Sind die im Metamodell zusätzlich spezifizierten Bedingungen (Constraints) im abstrakten Syntaxgraphen des zu untersuchenden Modells erfüllt? (in Analogie zu Programmiersprachen handelt es sich hierbei um weitere Nebenbedingungen, die dem Begriff der statischen Semantik untergeordnet werden)
- Wird durch die Modellsyntax die allgemeine Abbildungseigenschaft, die grundsätzlich an Modelle im Sinne ihrer Definition (Stachowiak 1973) gestellt wird, ausgedrückt? (in Analogie zu Programmiersprachen handelt sich hierbei um zusätzliche (statische) Prüfungen (Johnson 1977), die im Rahmen der semantischen Analyse bzw. Optimierung eines Compilers (Kontroll- bzw. Datenflussanalyse) durchgeführt werden bspw. der Erkennung von nicht ausführbaren Code, i. F. wird hierbei der Begriff semantische Anomalie verwendet)

Durch die erwähnten Syntaxanalysen können Fehler in Modellversionen festgestellt werden, deren Syntax aus einer automatischen Synthese paralleler Bearbeitungen resultiert. Die Ermittlung der parallelen Bearbeitungsschritte, die ursächlich für die gefundenen Fehler sind, ist anschließend trivial.

Die automatische Erkennung der Fehlersituation und der entsprechenden inkonsistenten Änderungen ist aber außerordentlich nützlich für eine möglichst effiziente Korrektur. Die Korrektur selbst muss manuell erfolgen. Ziel der Arbeit ist es damit ausdrücklich nicht, inkonsistente Änderungssituationen automatisch aufzulösen, sondern die Modellierer durch die automatische Fehlererkennung weitestgehend bei der Korrektur zu unterstützen.

Wie in den letzten Abschnitten motiviert wurde, befindet sich eine Vielzahl an Modellierungssprachen im Einsatz. Zur Gewährleistung einer flexiblen Anwendbarkeit der erforschten Ansätze zur Modelldatenverwaltung muss deren Konzeption von spezifischen Modellierungssprachen abstrahieren. Das bedeutet, dass es möglich sein soll, Modelle beliebiger Modellierungssprachen unter Angabe eines maschinenverarbeitbaren Metamodells zu verwalten. Bevor im nächsten Abschnitt die Vision dieser Arbeit für die Praxis des Software Engineering vorgestellt wird, soll zunächst auf die beiden Ziele im Detail eingegangen werden.

Automatische Integration paralleler Modellbearbeitungen auf Basis syntaktischer Modellbausteine. Die meisten in der Praxis eingesetzten Modellierungswerkzeuge persistieren Modelle und die zugehörigen Diagramme in einzelnen Dateien oft als XML (Topcased 2010; NoMagic Inc. 2010a; IBM 2010a). Die zur Versionsverwaltung dieser Daten eingesetzten SCM-Systeme zerlegen die ihnen zur Verwaltung zugewiesenen Daten in einzelne versionierbare Datenartefakte. Diese Zerlegung orientiert sich an der Ablagestruktur der zu verwaltenden Daten auf dem Rechner und nicht an ihrer durch eine Sprache vorgegebenen syntaktischen Struktur. Am weitesten verbreitet ist bei heutigen SCM-Systemen die

Versionierung von Dateien, Verzeichnissen bzw. Zeilen in Textdateien. Bei dieser Art der Integration paralleler Modellbearbeitungen scheitert sehr häufig schon die lexikalische Analyse des Ergebnisses und macht eine weitergehende Untersuchung der Modellsyntax unmöglich. Zur Überwindung dieses Problems wird bei der der angestrebten Lösungskonzeption das Ziel verfolgt, Modelldaten anhand ihrer spezifischen syntaktischen Struktur in sog. morphologische Dateneinheiten zu zerlegen. Morphologische Dateneinheiten sind syntaktisch abgeschlossene Einheiten (syntaktische Bausteine), deren Informationsgehalt atomar maschinell verarbeitet wird. Als Indikator für die Identifizierung einzelner morphologischer Modellinformationen sollen dabei die in einem Metamodell einer Modellierungssprache formal beschriebenen syntaktischen Entitätstypen, ihre Attribute und ihre Relationen dienen. Nachdem eine derartige Zerlegung durchgeführt wurde, lassen sich syntaktische Änderungen durch den Vergleich von Modellen erkennen. Dieser Vergleich ist notwendig um das etablierte 3-Wege-Merge-Prinzip für ein zu entwerfendes SCM-Verfahren anzuwenden. Auf Grundlage dieses Prinzips muss ein Verfahren zur Erzeugung einer gemeinsamen Modellversion (sog. *gemergete Version* – Abk. VM), in der alle parallelen Änderungen eingepflegt wurden, geschaffen werden.

Automatische Unterstützung zur statischen Analyse Korrektheit integrierter Modelle. Aufgrund der parallelen Bearbeitung von Modellen können, wie oben geschildert, lexikalische Fehler (→ fehlerhafte Datenrepräsentation atomarer Modellelemente sog. Token) oder syntaktische Fehler (→ korrekte Datenrepräsentation atomarer Modellelemente aber ihre Verknüpfung verletzt die Sprachspezifikation) auftreten. Durch das oben motivierte Integrationsverfahren werden lexikalische Fehler bei Merge eines Modells im Gegensatz zur Anwendung etablierter SCM-Verfahren vermieden. Syntaktische Fehler können im Rahmen der Integration aber weiterhin auftreten. Existiert ein formales Metamodell, der Modellierungssprache, so lassen sich syntaktische Fehler automatisch erkennen und analysieren. Werden Verstöße gegen das Metamodell oder nicht instanziierbare Modellelemente erkannt, ist in der Regel eine automatische Lokalisierung und Analyse dieser Fehler möglich. Ferner lassen sich analog zu statischen Codeanalysen (wie Daten- bzw. Kontrollflussanalyse) Anomalien in Modellen erkennen die auf Fehler hindeuten. Die Analysen unterstützen die effiziente weil zielgerichtete Problembehebung während der Integration durch eine entsprechende Automatisierung der Fehlererkennung. Zur Unterstützung der Modellintegration wird somit als zweites Ziel der Entwurf eines Ansatzes zur maschinellen statischen Modellanalyse (Syntax incl. statischer Semantik und weiter Anomalien) von automatisch zusammengeführten Softwaremodellen verfolgt. Eine Automatisierung der Fehlerbehebung wird explizit nicht angestrebt.

4 Vision – Kollaborative Modellierungsumgebung

Mit der Entwicklung tragfähiger Lösungen, die die im letzten Abschnitt formulierten Ziele adressieren, sollen wesentliche Bausteine zur Umsetzung der Vision einer kollaborativen Modellierungsumgebung bereitgestellt werden. Der Fokus bei der Entwicklung dieser Umgebung soll dabei auf dem effizienten und korrekten Entwurf von Softwaresystemen im Team liegen.

Dieser Abschnitt dient dem inhaltlichen Überblick der in den folgenden Kapiteln vorgestellten Lösungskonzeption. Dazu werden eingangs die durch die Software-Engineering-Praxis gegebenen Rahmenbedingungen überblicksartig erläutert. Anschließend werden die für die Lösungsansätze notwendigen Grundlagen vorgestellt, bevor auf notwendige Voraussetzungen zur Anwendung einzelner Lösungsansätze eingegangen wird.

Abschließend werden Beitrag, Bedeutung und Innovationscharakter der vorgestellten Lösungskonzeption ausgeführt.

4.1 Rahmenbedingungen

Um eine breite Einsetzbarkeit zu ermöglichen, werden zwei wesentliche Rahmenbedingungen an die zu entwickelnden Verfahren gestellt – Modellierungssprachunabhängigkeit und Werkzeugunabhängigkeit.

Modellierungssprachunabhängigkeit: In der Einleitung wurde bereits angesprochen, dass für die konzeptionelle Modellierung in den frühen Software-Entwicklungsphasen eine Reihe von modellbasierten Beschreibungssprachen verbreitet sind. Für eine möglichst breite Einsetzbarkeit muss die zu entwerfende Teammodellierungsumgebung unabhängig von einer spezifischen Modellierungssprache einsetzbar sein. Das bedeutet, dass alle nachfolgend entworfenen Ansätze zur Verarbeitung und Prüfung von Modellversionen keine spezifische Modellierungssprache der Modelle voraussetzen sollen. Vielmehr sollen die modellierungssprachspezifischen Informationen (z. B. Metamodelle), die zur Verarbeitung und Prüfung von Modellen während der Integration notwendig sind, den entworfenen Modellintegrationsverfahren explizit als Eingabeparameter zur Verfügung gestellt werden.

Werkzeugunabhängigkeit: Im Gegensatz zu Programmcode, der als reiner Text vorliegt und mit beliebigen Texteditoren bearbeitet werden kann, sind zur Darstellung und Verarbeitung von Modellen spezifische Modellierungswerkzeuge notwendig. Die i. F. vorgestellten Verfahren zur Verarbeitung von Modellversionen in kollaborativen Entwicklungsprozessen sollen weitestgehend unabhängig von entsprechenden Modellierungswerkzeugen entworfen werden. Da sich die Standards XMI (Object Management Group (OMG) 2010a) zur Persistierung von Modellen und das EMF-Datenmodell (Eclipse Foundation 2010) zum Modellaustausch bei den meisten Herstellern von Modellierungssoftware durchgesetzt haben, bilden diese Standards die Grundlage für eine werkzeugunabhängige Weiterverarbeitung von Modelldaten in den zu entwickelnden Verfahren.

4.2 Einbettung in Forschungs- und Anwendungslandschaft

Bei der kollaborativen Codeentwicklung haben sich optimistische Merge-Verfahren zur Integration paralleler Programmierarbeit weitgehend durchgesetzt und sind in allen etablierten Versionsmanagementwerkzeugen implementiert. Die Integration von parallel bearbeiteten Modellversionen wird allerdings nur sehr unsystematisch und rudimentär durch Werkzeuge unterstützt. Aus diesem Grund unterliegt in der Praxis die parallele Teamarbeit bei der Modellierung entweder umfangreichen Restriktionen bzgl. der Bearbeitungsautorität, ist gänzlich sequenziell oder erfordert hohe manuelle Abstimmungs- bzw. Integrationsaufwände (Prudêncio, Murta und Werner 2009).

Eine Übertragung und Implementierung etablierter optimistischer Merge-Verfahren von Textdokumenten auf Modelldokumente würde diese unbefriedigende Situation in vielen Fällen verbessern und wird durch die Industrie vielfach eingefordert (Bendix und Emanuelsson 2008; Bendix und Emanuelsson 2009b; Bendix und Emanuelsson 2009a; Brunet u. a. 2006). Der in der Realisierungsphase erfolgreich erprobte Prozess – parallele Bearbeitung → Versions-zusammenführung → Validierung – dient somit als Muster für den zu konzipierenden kollaborativen Modellierungsprozess der Analyse- und Entwurfsphase.

Neben optimistischen Merge-Verfahren werden weitere Ansätze aus dem Bereich des modellbasierten Software-Engineerings als Grundlagen genutzt. Insbesondere sind hierbei

Ansätze zur Definition von Modellierungssprachen zu nennen. Die Meta Object Facility (MOF) (Object Management Group (OMG) 2010b) wurde dazu als sog. Meta-Metamodellierungssprache standardisiert. Das bedeutet, auf Basis dieser Modellierungssprache ist es möglich eine Sprachspezifikation als Metamodell für eine beliebige (domänenspezifische) Modellierungssprache zu erzeugen.

Die MOF bietet zur Metamodellspezifikation umfangreiche Möglichkeiten (Typmodell, Attributierung von Entitäten und Relationen, etc.), um komplexe Strukturen effizient und übersichtlich als Graph zu beschreiben. Da sich der MOF-Standard weitgehend zur Spezifikation von universellen (z. B. UML) und domänenspezifischen Modellierungssprachen in der modellbasierten Softwareentwicklung durchgesetzt hat und durch die meisten Werkzeuge auch implementiert wurde, dient dieser Sprachstandard als eine weitere Grundlage der Lösungskonzeption.

Die geforderte automatisierte Korrektheitsprüfung von Modellen einer per MOF spezifizierten Sprache benötigt eine maschinenauswertbare Beschreibung der Semantik des Meta-Metamodells. Der in den folgenden Kapiteln vorgestellte Ansatz stützt sich zur Semantikbeschreibung der MOF auf eine Transformation von per MOF spezifizierten Metamodellen und den zu prüfenden Modellen in eine Menge beschreibungslogischer Ausdrücke (Baader u. a. 2003) einer Wissensbasis bzw. formale Anfragen (sog. Queries) an diese.

Für Beschreibungslogiken existieren hochoptimierte Werkzeuge zum automatischen Schlussfolgern sog. Reasoner und zur Anfrageauswertung. Die genannten Grundlagen sind zur Konzeption eines Ansatzes zur automatischen Validierung von Modellen geeignet und werden entsprechend genutzt.

5 Beitrag, Bedeutung und Innovationscharakter der Lösungskonzeption

Nachdem die Ziele der vorliegenden Arbeit erörtert wurden, wird in diesem Abschnitt auf den konkreten Beitrag, die Bedeutung für die Praxis und den Innovationscharakter der vorliegenden Arbeit eingegangen. Dabei wird auch erläutert, welche Ziele mit der vorgestellten Lösungskonzeption nicht verfolgt werden.

Der Beitrag der Arbeit gliedert sich in zwei Themenbereiche. Im Rahmen dieser werden die vier unten aufgeführten Ansätze entwickelt. Zusammen mit vorhandenen Grundlagen zur modellbasierten Softwareentwicklung bilden die vorgestellten Ansätze eine Lösung für die in Abschnitt 3 formulierten Ziele.

- Automatische und metamodell-unabhängige Versionsverwaltung von Modellen
  - Ansatz zur Zerlegung der Modellsyntax in morphologische Einheiten
  - Ansatz zur Integration syntaktischer Modelländerungen parallel erstellter Modellversionen

- Automatische und modellierungssprachspezifische Validierung von Modellen zur Erkennung inkonsistenter, paralleler Modelländerungen
  - Ansatz zur beschreibungslogischen Repräsentation von Modellsyntax und MOF-basierten Metamodellen
  - Ansatz zur merge-bedingten Validierung der Modellsyntax auf Grundlage von durch die MOF spezifizierten Modellierungssprachen

Eine Lösung für die in Abschnitt 3 formulierten Ziele hat eine weitreichende Bedeutung für die kollaborative Modellierung von Softwaresystemen. Die bisher aufwendige und fehlerträchtige Integration parallel weiterentwickelter Modellversionen kann bzgl. der Belange Effizienz und Qualität durch ein rechnergestütztes Verfahren stark verbessert werden. Diese Verbesserung beruht auf einer Automatisierung bisher sehr arbeitsintensiver Arbeitsschritte zur Verwaltung verschiedenster Modelle in den frühen Phasen der Softwareentwicklung.

Obwohl das Problembewusstsein bezüglich der oben in Abschnitt 2 aufgeführten Punkte im wissenschaftlichen und industriellen Umfeld durchaus vorhanden ist, existieren noch keine flexiblen und tragfähigen Lösungen für die in dort benannten Ziele. Forschungsarbeiten zu ähnlichen oder verwandten Themen werden im folgenden Abschnitt im Detail erläutert und gegeneinander abgegrenzt.

Die erforschten Ansätze ergänzen die aktuell vorhandenen Grundlagen im Bereich der modellbasierten Softwareentwicklung, sodass auf dieser Grundlage eine Werkzeugunterstützung zur Versionsverwaltung im Rahmen einer kollaborativen Modellierungsumgebung entwickelt werden kann.

Dabei liegt der Fokus auf der technischen Konzeption automatischer Verfahren. Insbesondere liegt der Schwerpunkt nicht auf dem Entwurf eines kollaborativen Modellierungsprozesses bzw. einer Arbeitsorganisation von Modellierungsteams. Darüber hinaus konzentrieren sich die entwickelten Verfahren auf die Verwaltung der abstrakten Syntax, nicht aber auf die Verwaltung der grafischen Repräsentation von Modellen oder der identifizierten Fehler.

Ebenfalls werden die zur Modellvalidierung verwendeten, automatischen Schlussfolgerungsverfahren im Rahmen der Verwendung von Beschreibungslogiken nicht modifiziert und weiterentwickelt. Diese werden lediglich als konzeptionelle und technische Grundlage für die automatische Validierung innerhalb der Modellverwaltung nutzbar gemacht.

Diese Modellanalyse beschränkt sich dabei auf die metamodell-spezifische, syntaktische Korrektheitsprüfung und weiterer statischer Analysen zur Erkennung semantischer Anomalien. Dabei dienen die Analysen lediglich der Fehlererkennung, eine Automatisierung der Fehlerauflösung wird nicht angestrebt.

6 Verwandte Arbeiten

Die ersten Arbeiten mit Bezug zu den oben formulierten Zielen gehen zurück auf das Versionsmanagement von Dateien, digitalen Textdokumenten und Quellcode im Allgemeinen. Die wichtigsten dieser Arbeiten sollen im folgenden Unterabschnitt erwähnt sein. Eine ausführlichere Erläuterung der für diese Arbeit relevanten Grundlagen wird Kapitel II gegeben.

Abschnitt 6.2 gibt einen Überblick über die Forschungsaktivitäten zur Unterstützung der kollaborativen Modellierung bzw. des Versionsmanagements von Modellen. Dabei werden die Zielsetzungen der angeführten Arbeiten in Bezug zur Zielsetzung.

6.1 Software Configuration Management (SCM) – Konzepte und Technologien

Viele Ansätze im Gebiet des Software Configuration Management (SCM) wurden unmittelbar als Softwarewerkzeuge implementiert (Estublier u. a. 2005). Im Rückblick lagen die Schwerpunkte dieser Entwicklung im Bereich des Versionsmanagements von Softwareartefakten auf Dateisystem- bzw. Textdateiebene.

Die wohl bedeutendste Evolutionsschiene wird dabei wohl durch die Entwicklung der Versionskontrollsysteme (engl. *version control system – VCS*) RCS (Tichy 1985; GNU Project 2010a) über CVS (Grune 1986; GNU Project 2010b) bis hin zu Subversion (Collins-Sussman 2002; Tigris.org 2010) gebildet. Diese Werkzeuge realisieren eine zentrale Versionsverwaltung (i. d. R. als Client-/Serverarchitektur) von Verzeichnissen, Dateien und Textartefakten. Daneben hat sich eine Gruppe weiterer verteilter SCM-Ansätze und Werkzeuge insbesondere im Umfeld der Entwicklung des Linux-Kernels etabliert (Clatworthy 2007; Canonical Ltd. 2010; Git Authors 2010; BitMover Inc. 2010).

In der heutigen Programmierpraxis werden die genannten Ansätze bzw. Werkzeuge teils in umfangreiche Entwicklungsumgebungen integriert (IBM 2010b; Microsoft 2010).

Obwohl die genannten VCS-Werkzeuge bei der Integration parallel entwickelter Programmcodeversionen nicht die formale Syntax der jeweiligen Sprache berücksichtigen, konnten sie einen entscheidenden Beitrag zur effizienten Versionsverwaltung in großen parallel entwickelnden Programmiererteams leisten (Frühauf und Zeller 1999).

Seit den 90-er Jahren wird in der Forschungsgemeinde aber auch die Berücksichtigung der formalen Syntax während des Merge-Vorgangs von Softwareversionen diskutiert (Buffenbarger 1995; Mens 1999; Mens 2002; Hunt und Tichy 2002; Niu, Easterbrook und Sabetzadeh 2005). Ziel dessen ist, eine höhere Leistungsfähigkeit bei der Verwaltung von syntaktisch formal beschriebenen und parallel fortentwickelten Softwareartefakten zu ermöglichen.

Mit der verstärkten Verbreitung der modellbasierten Softwareentwicklung und dem damit einhergehenden Bedürfnis die formale Syntax von Softwaremodellen während der Entwicklung zu verwalten, gewannen Überlegungen in Richtung Software Merging wieder an Aktualität (Mens 2000).

6.2 Versionskontrolle von Modellen

Da etablierte Versionsverwaltungsverfahren die parallele Entwicklungsarbeit an Modellen nicht hinreichend genug unterstützen, wird seit einigen Jahren im wissenschaftlichen Umfeld und bei den Werkzeugherstellern nach neuen Lösungsansätzen gesucht (Altmanninger, Seidl und Wimmer 2009). Für spezifische Modellierungssprachen wie die UML werden durch die Werkzeughersteller Case-Tool-spezifische Lösungen zur Teamunterstützung angeboten (NoMagic Inc. 2010b; MID GmbH 2010; Letkeman 2005). Dabei kommen sowohl pessimistische als auch optimistische Synchronisierungsstrategien zum Einsatz.

Ansätze aus dem wissenschaftlichen Umfeld wie (Meisinger, Rausch und Sihling 2006) unterstützen die parallele Bearbeitung von XML-basierten Modellen durch Erweiterungen bzw.

Formatierungen der XML-Modellrepräsentation, die es ermöglichen, Modelle mit klassischen Werkzeugen wie CVS oder Subversion zu verwalten.

Die meisten Forschungsansätze konzentrieren sich allerdings auf die Verwaltung (Vergleich - Diff und Integration parallel erstellter Versionen - Merge) der spezifischen syntaktischen Struktur von Modellen (Alanen und Porres 2003; Mens und Theo D'Hondt 2000). Unterteilen lassen sich die Diff- und Merge-Strategien einerseits in modellierungssprachspezifische und sprachunabhängige und andererseits in zustands- und operationsbasierte Verfahren.

Modellierungssprachspezifische Verfahren konzentrieren sich auf die Verwaltung von Modellen einer speziellen Sprachsyntax. Dabei fokussieren die meisten Arbeiten in dem Gebiet auf die Verarbeitung von UML-Modellen (Murta u. a. 2007; Kelter, Wehren und Niere 2005; Kelter, Schürr und Westfechtel 2007; Christian Schneider und Zündorf 2007; Christian Schneider, Zündorf und Niere 2004; Ohst, Welle und Kelter 2003).

Modellierungssprachunabhängige Verfahren sollen ein adäquates Modellmanagement für Modelle beliebiger Modellierungssprachen unterstützen (Fleurey u. a. 2007). Viele Ansätze beruhen dabei auf der Verarbeitung von MOF- respektive EMF-spezifischen Metamodellen als Modellierungssprachspezifikation (Bartelt 2008a; Bartelt 2008b; Koegel und Helming 2010; Westfechtel 2010; Espinazo-Pagán und García-Molina 2010; Rick Salay u. a. 2007).

Unter anderem zur besseren Erkennung von inkonsistenten Änderungen (Ignat und Norrie 2004; Koegel, Hermannsdoerfer, Yang, Helming und David 2010) in parallel entwickelten Entwicklungsständen von Modellen wurden neben zustandsbasierten Modellvergleichs bzw. -Merge-Verfahren wie (Schmidt und Gloetzner 2008; Bartelt, Molter und Schumann 2009; Altmanninger 2008) operationsbasierte Verarbeitungsansätze entwickelt (Koegel, Helming und Seyboth 2009; Herrmannsdoerfer und Koegel 2010; Bendix, Koegel und Martini 2010; Westfechtel 2010; Tung Thanh Nguyen u. a. 2010; Lippe und van Oosterom 1992). Auf Forschungsaktivitäten im Bereich der Korrektheitsanalyse von Modellen und die Konsistenzanalyse paralleler Änderungen soll im nächsten Abschnitt eingegangen werden.

6.3 Konsistenzanalyse paralleler Modelländerungen

Werden Softwareartefakte wie Modelle in Teamarbeit geändert, so können inkonsistente Änderungen in parallel durchgeführten Weiterentwicklungen einer Version auftreten. Die automatische Erkennung dieser Änderungen und die Analyse der Ursachen ist der wesentliche Bestandteil in der Korrektur von Inkonsistenzen während der Merge-Phase in Versionskontrollsystemen (Spanoudakis und Zisman 2001).

Aktuelle Forschungsarbeiten machen sich zur Erkennung von inkonsistenten Änderungen zwei Umstände zunutze. Zum einen kann die formale Syntax einer Version nach der optimistischen Integration paralleler Änderungen hinsichtlich ihrer *syntaktischen (In-)Korrektheit* zur jeweiligen Sprachspezifikation untersucht werden. Liegt zum anderen die Folge von Anwendungen formal definierter Änderungsoperationen dediziert vor (bspw. bei operationsbasierten Merge-Ansätzen) und sind „Unverträglichkeiten“ zwischen den Änderungsoperationen hinreichend spezifiziert, so lassen sich inkonsistente Änderungen anhand der unverträglichen gleichzeitigen (parallelen) Anwendung bestimmter *unverträglicher Änderungsoperationen* ausmachen.

Konsistenzanalyseansätze auf Basis einer Untersuchung der syntaktischen (In-)Korrektheit unterscheiden sich dabei in sprachspezifische und generische Ansätze (Michel und Galal-Edeen

2009). Dabei steht bei sprachspezifischen Ansätzen die UML besonders im Zentrum der Betrachtungen (Elaasar und Briand 2004; Usman u. a. 2008).

Ansätze, die auf Basis einer Analyse der (Un-)Verträglichkeit von Änderungsoperationen die Konsistenz von parallelen Änderungen untersuchen, gehen zurück bis auf syntaxspezifische Ansätze zur Evolution von Programmcode (Mens 1999). Dabei unterstützen einige Ansätze die Identifizierung der Kette angewendeter Änderungsoperationen (engl. *Traces/Batches*) (Mens, Straeten und Maja D'Hondt 2005; Dantas, Murta und Werner 2005; Mens 2000; Schmidt u. a. 2009), andere setzen hingegen ein werkzeuggestütztes protokollieren von Änderungen (Herrmannsdoerfer und Koegel 2010) voraus und konzentrieren sich auf die Konzipierung von Merge und Konfliktanalyse (Koegel, Naughton u. a. 2010; Blanc u. a. 2008).

7 Überblick und Struktur der Arbeit

In diesem Kapitel wurden einige wesentliche Defizite in der Werkzeugunterstützung der kollaborativen Softwaremodellierung aufgezeigt und daraus Zielsetzungen zu deren Überwindung abgeleitet. Weiterhin wurden die konzeptionellen Herausforderungen zur Erreichung der formulierten Ziele identifiziert. In den folgenden Kapiteln werden die beschriebenen Herauforderungen in Teilprobleme zerlegt, für die dann Lösungen beschrieben werden. Diesbezüglich enthält die folgende Auflistung eine Gliederung der Dissertation stellt einen Überblick über alle weiteren Kapitel der Dissertation bereit:

Kapitel II - Grundlagen. Dieser Teil erörtert alle für die Lösungskonzeption relevanten und verwandten Grundlagen aus dem Bereich kollaborative Entwicklung mit den Schwerpunkten Versionsverwaltung und modellbasierte Softwareentwicklung.

Das Software Configuration Management (SCM) ist heutzutage insbesondere in der Programmierung elementarer Bestandteil von Softwareprojekten. Die Grundlagen in diesem Gebiet sind ausgereift und erprobt. Die Terminologie und die Fähigkeit zur Interaktion mit entsprechenden Werkzeugen gehören zum Fachwissen von Softwareingenieuren. Aufgrund dieser Voraussetzungen knüpft die Konzeption zur Versionsverwaltung von Modellen an die Grundlagen in diesem Bereich an. Dazu wird in Kapitel II die in den folgenden Kapiteln verwendete Terminologie erläutert und grundlegende Versionierungskonzepte, Methoden und Werkzeuge zusammenfassend dargestellt.

Im Bereich der modellbasierten Softwareentwicklung konzentrieren sich die Erörterungen auf aktuelle Modellierungstechniken auf Basis universeller und domänenspezifischer Modellierungssprachen sowie Techniken zur formalen syntaktischen Spezifikation von Modellierungssprachen durch Metamodelle auf Basis der MOF.

Kapitel III - Synchronisierung kollaborativ entwickelter Softwaremodelle. Die technische Unterstützung in der kollaborativen Programmierung wird allgemein als gut befunden. SCM-Werkzeuge wie bspw. Subversion und Bitkeeper stellen das in vielen erfolgreichen Softwareprojekten unter Beweis. Was ist das Geheimnis dieses Erfolges und wieso lässt er sich nicht ohne Weiteres auf die kollaborative Modellierung übertragen? Dieser Frage widmet sich Kapitel *III*. Dazu werden die Aspekte des Merge-Verfahrens für Quellcode, das durch populäre SCM-Werkzeuge implementiert wird, genauer beleuchtet. Anhand eines eingeführten Illustrationsbeispiels werden die Unterschiede in der Verwaltung von Quellcode und Modellen erklärt, die eine adäquate Versionsverwaltung von Modellen so schwierig machen. Ferner

werden für die hervorgehobenen Probleme Lösungen skizziert, die im nachfolgenden Kapitel im Detail ausgearbeitet werden.

Kapitel IV - Korrekte Integration asynchroner Modelländerungen. In diesem Kapitel werden die in Abschnitt 5 motivierten Beiträge im Detail erläutert. Dabei gliedert sich das Kapitel in die beiden inhaltlichen Schwerpunkte zum morphologischen Merge-Verfahren für Modelle und zur statischen Fehleranalyse von integrierten Bearbeitungsständen von Softwaremodellen. Im ersten Abschnitt dieses Kapitels wird ein Konzept zur Versionsverwaltung von Modellen auf Basis ihrer syntaktischen Struktur vorgestellt. Dabei steht die Zusammenführung der abstrakten Syntax von parallel fortentwickelten Modellen im Zentrum der Betrachtungen. Da wie oben eingehend motiviert auch eine syntaktische Zusammenführung parallel geänderter Modelle ihre sprachspezifische Korrektheit gefährdet, soll im zweiten Teil des Kapitels ein Konzept zur Unterstützung der Validierung integrierter Modellversionen durch Automatisierung vorgestellt werden.

Kapitel V - Implementierung und Konzeptbewertung. Kapitel *V* geht auf die Anwendung der in der Dissertation entworfenen Lösung ein. Dabei steht die entstandene Softwareimplementierung im Vordergrund, die auf Grundlage der in Kapitel *IV* vorgestellten Konzeption entwickelt wurde. Diese wurde in die bestehende Softwarelandschaft des Eclipse-Modeling-Projektes integriert und lässt sich im Zusammenspiel mit verschiedensten Modellierungswerkzeugen nutzen. Im Rahmen der Erläuterungen zur Umsetzung der Konzepte im Werkzeug werden die Vorteile der rechnergestützten Versionsverwaltungsverfahren für Modelle zum Status quo hervorgehoben.

Kapitel VI - Zusammenfassung und Perspektiven. Im letzten Teil werden die entwickelten Forschungsergebnisse im Kontext ihrer Bedeutung für das Software Engineering überblicksartig zusammengefasst. Abschließend wird ein Ausblick hinsichtlich potenzieller Weiterentwicklungen und Ergänzungen der beschriebenen Ansätze gegeben.

II Grundlagen

Im Bereich der kollaborativen Softwareentwicklung existiert ebenso wie im Forschungsgebiet der modellbasierten Softwareentwicklung eine Vielzahl an Grundlagen, auf die sich eine Lösung für die in Kapitel I motivierte Problemstellung stützen kann. Zu diesen Grundlagen gehören Forschungsergebnisse aus den Bereichen kollaboratives Software Engineering, Software Configuration Management (SCM) und modellbasierte Softwareentwicklung.

In den folgenden Abschnitten werden die für die kollaborative Softwaremodellierung relevanten Themen erläutert. Zunächst wird dabei auf die Kollaboration im Software Engineering eingegangen, bevor anschließend das Thema Versionsverwaltung herausgegriffen und detaillierter erörtert wird. Durch die Bereitstellung von Methoden und Werkzeugen zur Synchronisierung von Teamarbeit leisten diese Forschungsgebiete seit vielen Jahren erfolgreich einen Beitrag zur Koordinierung der Softwareentwicklung.

Da syntaktisch formal repräsentierte Softwaremodelle betrachtet werden, widmet sich der letzte Abschnitt dieses Kapitels den relevanten Grundlagen aus dem Bereich der Definition von Modellierungssprachen in der modellbasierten Softwareentwicklung. Insbesondere steht hierbei die formale syntaktische Spezifikation von Modellierungssprachen durch Metamodelle im Mittelpunkt.

1 Kollaboration im Software Engineering

Der Begriff Kollaboration ist ursprünglich im deutschen und französischen Sprachraum negativ geprägt. So beschreibt das Deutsche Wörterbuch (Wahrig 2005) den Begriff als „das Kollaborieren, Zusammenarbeit mit dem Feind oder der Besatzungsmacht [frz. Collaboration 'Mitarbeit']“. Der Begriff geht zurück auf das lateinische Wort *collaborare*, das mit zusammenarbeiten übersetzt wird.

In der Informatik ist die Bedeutung des Begriffes nicht negativ belegt. Im Gebiet des Software Engineering wird der Begriff im Sinne der englischen Bedeutung des Wortes

Collaboration verwendet, was wertneutral die Zusammenarbeit von Personengruppen im Allgemeinen bezeichnet.

Die Software-Engineering-Forschung zum Thema Kollaboration ist stark verzahnt mit dem interdisziplinären Forschungsgebiet des *Computer Supported Cooperative Work* (Abk. *CSCW*) (Baecker 1992). Das Gebiet des CSCW hat eine lange Tradition in der Systematisierung vieler Begriffe im Kontext kollaborativer Zusammenarbeit von Personengruppen. Etabliert hat sich hier das sog. 3K-Modell (Teufel, Sauter und Mühlherr 1995), das drei Ebenen der Kollaboration unterscheidet – Kommunikation, Koordination und Kooperation. Es ist in in Abbildung II.1 dargestellt.

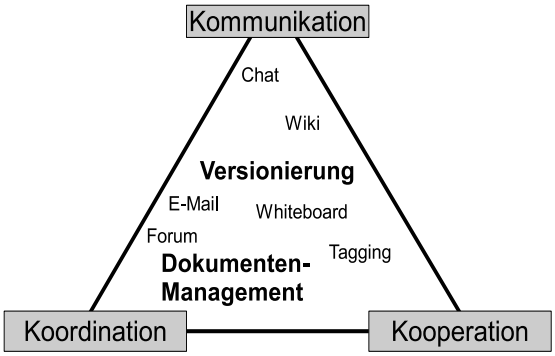


Abbildung II.1. 3K-Modell

Der Begriff Kommunikation bezeichnet dabei den Austausch von Informationen zwischen Personen einer Gruppe. Kommunikation ist die Grundlage zur Betrachtung von Koordination und Kooperation.

Unter dem Begriff Koordination werden Aktivitäten zur Arbeitsplanung von Teams zusammengefasst. Diese umfassen die Zerlegung von Aufgaben in Teilaufgaben, die Zuordnung von Aufgaben zu Teammitgliedern, die zeitliche Einplanung von Aufgaben und die Organisation der Zusammenführung von Ergebnissen. Sowohl in der Kommunikation als auch in der Koordination spielen die Begriffe *synchron* und *asynchron* eine wichtige Rolle. Bei synchroner Kommunikation als auch bei synchron eingeplanten Projektaktivitäten (inter-)agieren die Akteure gleichzeitig und untereinander abgestimmt. Im asynchronen Fall ist der Austausch von Informationen bzw. die gemeinsame parallele Arbeit an Produkten nicht jederzeit abgestimmt. Die Arbeitsweise muss deswegen in angemessenen Zeitabständen immer wieder synchronisiert werden.

Die Kooperation im Sinne des 3K-Modells charakterisiert den strategischen Rahmen der Zusammenarbeit. Dabei steht der Gesichtspunkt der Ziel-, Interessen- und Planungsübereinstimmung zusammenarbeitender Akteure im Vordergrund. Der Begriff Kollaboration wird i. F. im Kontext der Entwicklung von Dokumenten im Team verwendet. Unter diesem Begriff werden alle den drei Ebenen des 3K-Modells zugehörigen Aktivitäten verstanden.

Zur Steuerung der Kollaboration von Ingenieuren im Software Engineering spielen Softwarewerkzeuge eine große Rolle. Dahin gehend werden eine Reihe von Ansätzen aus dem CSCW-Bereich im Rahmen des Software Engineerings schon erfolgreich in der industriellen Praxis eingesetzt. Einen großen Raum nimmt dabei die Werkzeugunterstützung zur Teamarbeit

an Entwicklungsprodukten wie bspw. Quellcode ein. Im Fokus des folgenden Abschnittes steht die Analyse des aktuellen Standes der kollaborativen Softwareentwicklung.

1.1 Aktueller Stand der kollaborativen Softwareentwicklung

Softwareentwicklungsprojekte sind heutzutage standortübergreifend wenn nicht sogar global organisiert (Bartelt u. a. 2009). Die Domänen-übergreifende Vernetzung und das starke Vordringen von Software in viele Produktbereiche zwingt IT-Unternehmen, projektbezogene Kooperationsnetzwerke mit anderen Unternehmen einzugehen. Trotz der verteilungsbedingten Herausforderungen in der Koordination solcher Softwareprojekte steigen die Anforderungen an ihre Zeit- und Kosteneffizienz sowie an die Qualität der zu erstellenden Softwareprodukte.

Die sich verändernde Ausgangslage erfordert verstärkt verbesserte Lösungsansätze zur Gestaltung der kollaborativen Projektarbeit hinsichtlich der standortübergreifenden Kommunikation, Koordination und Kooperation. Um die Entwicklungszeit großer Softwaresysteme so gering wie möglich zu halten, arbeiten die Projektbeteiligten möglichst parallel. Dabei ist insbesondere die Erstellung von Softwareprodukten in den Spezifikations-, Entwurfs- und Realisierungsphasen Aufgabe von kollaborativ arbeitenden Ingenieurteams.

Um die Entwicklungsarbeit solcher Teams zu koordinieren, bedarf es eines angemessenen Software Configuration Managements (SCM), das die Verzahnung paralleler Aktivitäten möglichst automatisiert unterstützt. Im Kontext des Software Engineerings befasst sich diese Disziplin mit der Erforschung von geeigneten Lösungsansätzen zur Verwaltung von Softwareprodukten gerade auch in kollaborativen Entwicklungsprozessen.

1.2 Software Configuration Management (SCM)

Das *Software Configuration Management (SCM)* ist eine Ausprägung des Versions- und Konfigurationsmanagements in Bezug auf Software. SCM wird in aktuellen Softwareprojekten zur Koordinierung der Softwareentwicklung eingesetzt. Die zentralen Aufgaben des SCM sind:

- Versionsverwaltung von Softwareprodukten.
  Die Versionsverwaltung unterstützt die Koordinierung eines Softwareprojektes durch die Bereitstellung eines teamweiten Zugriffs auf Dokumente, die Archivierung bestimmter Arbeitsstände, die Protokollierung von Änderungen an Dokumenten und die Synchronisierung asynchroner Dokumentbearbeitungen in Entwicklerteams. Softwarewerkzeuge aus diesem Bereich unterstützen heute die Versionsverwaltung von Verzeichnissen und Dateien.

- Variantenverwaltung von Softwareprodukten.
  Die gleichzeitige Entwicklung, Pflege und Weiterentwicklung verschieden ausgeprägter Softwareprodukte (Varianten) sind die zentralen Herausforderungen dieses Teilgebietes. Hierbei werden Methoden und Techniken erforscht, die einerseits der Entkopplung von variablen und gemeinsamen Softwareteilen dienen und andererseits die korrekte Kombination variabler Features in Varianten ermöglichen.

- Änderungsverwaltung von Softwareprodukten.
  Eine Rückkopplung von Anwendern bzgl. Fehler, gewünschter Veränderungen oder Erweiterungen ist wichtig, um die Akzeptanz und Langlebigkeit von Softwareprodukten zu gewährleisten. Zur Koordination der Weiterentwicklung von Softwareprodukten von der Anwenderrückmeldung bis zur Umsetzung im Code bedarf es einer systematischen

Änderungsverwaltung. Diese ermöglicht die Dokumentation von Entwicklungsentscheidungen und die Integration von Entwicklungs- und Projektplanungsaktivitäten.

- Qualitätssicherung von Softwareprodukten.
  Für eine erfolgreiche Projektabwicklung muss die Qualität der Entwicklungsstände gemeinschaftlich erarbeiteter Softwareprodukte kontinuierlich überwacht und gesichert werden. Dazu muss die Konsistenz untereinander und die Korrektheit der Softwareprodukte garantiert werden. Zur Ermöglichung einer effizienten Fehlererkennung und -analyse helfen Softwarewerkzeuge. Syntaktische Prüfmechanismen findet man bspw. in allen populären Compilern und Debuggern für Programmiersprachen. Darüber hinaus erlauben automatische Testumgebungen und manuelle Reviews eine weitergehende semantische Prüfung von Softwareprodukten. Analysemechanismen zur Qualität von Softwareprodukten, die in eine SCM-Umgebung integriert sind, finden sich aber nur ansatzweise in wenigen Werkzeugen (IBM 2010b) wieder.

Der Anspruch des SCM, Softwareprojekte zu unterstützen, ist mit diesem umfangreichen Leistungsspektrum sehr hoch. Eine Analyse, inwieweit SCM-Ansätze im Produktivumfeld zu Einsatz kommen, fällt allerdings ernüchternd aus. Obwohl die Notwendigkeit eines systematischen Versions- und Konfigurationsmanagements im Software Engineering schon früh erkannt wurde (Estublier 2000), fand nur ein unbefriedigend kleiner Teil der Forschungsergebnisse breite Anwendung in der industriellen Softwareentwicklung.

Einzig in Bereichen, in denen eine praktikable Werkzeugunterstützung zur Verfügung gestellt wurde, zeigt sich ein positives Bild. Zu nennen sind hierbei das Versionsmanagement von Quellcode (vgl. Abschnitt 2) und die Änderungsverwaltung (engl. *Bugtracking*) (Scott und Nisse 2001).

2 Versionsverwaltung im Software Engineering

„Rom ist nicht an einem Tag erbaut worden!“ – Sinngemäß gilt dieses Sprichwort auch für die Entwicklung von Softwareprodukten, angefangen von den Anforderungsdokumenten, über Softwaremodelle bis hin zum Quellcode. All diese Produkte unterliegen schon im Laufe der Entwicklung und auch später in der Wartung einem kontinuierlichen Veränderungsprozess.

Um diesen Prozess systematisch kontrollieren zu können, kommt im Software Engineering die Versionsverwaltung zum Einsatz. In diesem Forschungsfeld spielen eine Reihe von Begriffen eine zentrale Rolle, welche im nächsten Abschnitt genauer erläutert werden.

2.1 Basisterminologie

Softwareprodukte sind Dokumente. Ihre Inhalte werden im Laufe ihrer Entwicklung kontinuierlich erweitert und geändert. In diesem Rahmen spielt die Bedeutung der Begriffe Änderung, Entwicklungsstand und Weiterentwicklung eine zentrale Rolle. Die folgenden Unterabschnitte führen die Bedeutung der zentralen Begriffe ein, wie sie nachfolgend verwendet werden. Eine ausführliche Dokumentation der Begriffswelt der Versionsverwaltung innerhalb des SCM findet sich in (Scott und Nisse 2001; Reidar Conradi und Bernhard Westfechtel 1998).

Dokument, Dateneinheit

Dokumente im Software Engineering sind textuelle oder grafische Repräsentationen von Daten- und Kontrollstrukturen, die zur Beschreibung der Datenverarbeitungen durch digitale Rechenmaschinen dienen.

Im Rahmen der Versionsverwaltung werden Dokumente anhand der Struktur ihrer Repräsentation auf dem Rechner in *Dateneinheiten* gegliedert. Typisch ist dabei eine Gliederung in Verzeichnisse, Dateien und Textzeilen.

Änderung, Entwicklungsstand, Weiterentwicklung

Änderungen an einem Dokument werden i. F. auf Ebene von Dateneinheiten betrachtet. Dateneinheiten können in einem Dokument hinzugefügt, ersetzt oder gelöscht werden.

Ein *Entwicklungsstand* ist der Zustand der Repräsentation eines Dokumentes zu einem bestimmten Zeitpunkt.

Ausgehend von einem bestimmten Entwicklungsstand wird eine *Weiterentwicklung* eines Dokumentes als eine Sequenz zeitlich aufeinanderfolgender Änderungen definiert.

Die Begriffe dienen einer feingranularen Beschreibung der Bearbeitung von Dokumenten über bestimmte Zeiträume hinweg. Versionsverwaltungsmethoden und -werkzeuge unterstützen die Teamarbeit, indem sie Änderungen bzw. Entwicklungsstände an Dokumenten archivieren und den Teammitgliedern geeignet zur Verfügung stellen. Dabei verwalten sie eine sog. Historie von Versionen bzw. Revisionen in einem sog. Repository.

Version, Revision, Versionshistorie, Repository

Laut (Conradi und Westfechtel 1998) repräsentiert eine *Version* den Entwicklungszustand eines sich fortentwickelnden Artefaktes. Übertragen auf die Begriffserläuterungen oben heißt das, dass eine Version eine Identität für einen bestimmten Entwicklungsstand festlegt.

Aktuelle Versionsverwaltungssysteme speichern durch den Anwender festgelegte Entwicklungszustände als Version ab, um diese zu archivieren, um bei Bedarf zu einem bestimmten Entwicklungsstand „zurückzukehren“ oder um die Version anderen Mitgliedern eines Entwicklungsteams zur Verfügung zu stellen. Außerdem besteht bei vielen Versionierungswerkzeugen die Möglichkeit, die im Rahmen einer neuen Version durchgeführten Änderungen zu dokumentieren.

Der Begriff *Revision* wird in der Versionsverwaltung oft synonym zum Begriff *Version* eingesetzt. Im Folgenden umfasst die Bedeutung dieses Begriffes eine bestimmte Version mit ihren Bezügen zu anderen Versionen. So bezieht sich nach der initialen Erstellung eines Dokumentes jede Version desselben mindestens auf eine andere, zeitlich vorher erzeugte Version (*Vorgängerversionen*). Ferner können durch Änderungen an einer Dokumentversion beliebig viele neue Versionen (*Nachfolgeversionen*) erstellt werden.

Der Begriff *Versionshistorie* beschreibt alle Revisionen eines Dokumentes von dessen Erstellung bis zu einer bestimmten Version.

Die Versionshistorie wird von Softwarewerkzeugen oft teamweit verfügbar gemacht. Einen zentralen Speicherort für Projekt-weit verfügbare Dokumentversionen bezeichnet man als *Repository*.

Damit die Bearbeitung eines komplexen Dokumentenbestandes im Team effizient gestaltet werden kann, muss die Entwicklung angemessen koordiniert werden. Ein Schwerpunkt dabei ist die inhaltliche Synchronisierung asynchron erzeugter Arbeitsergebnisse.

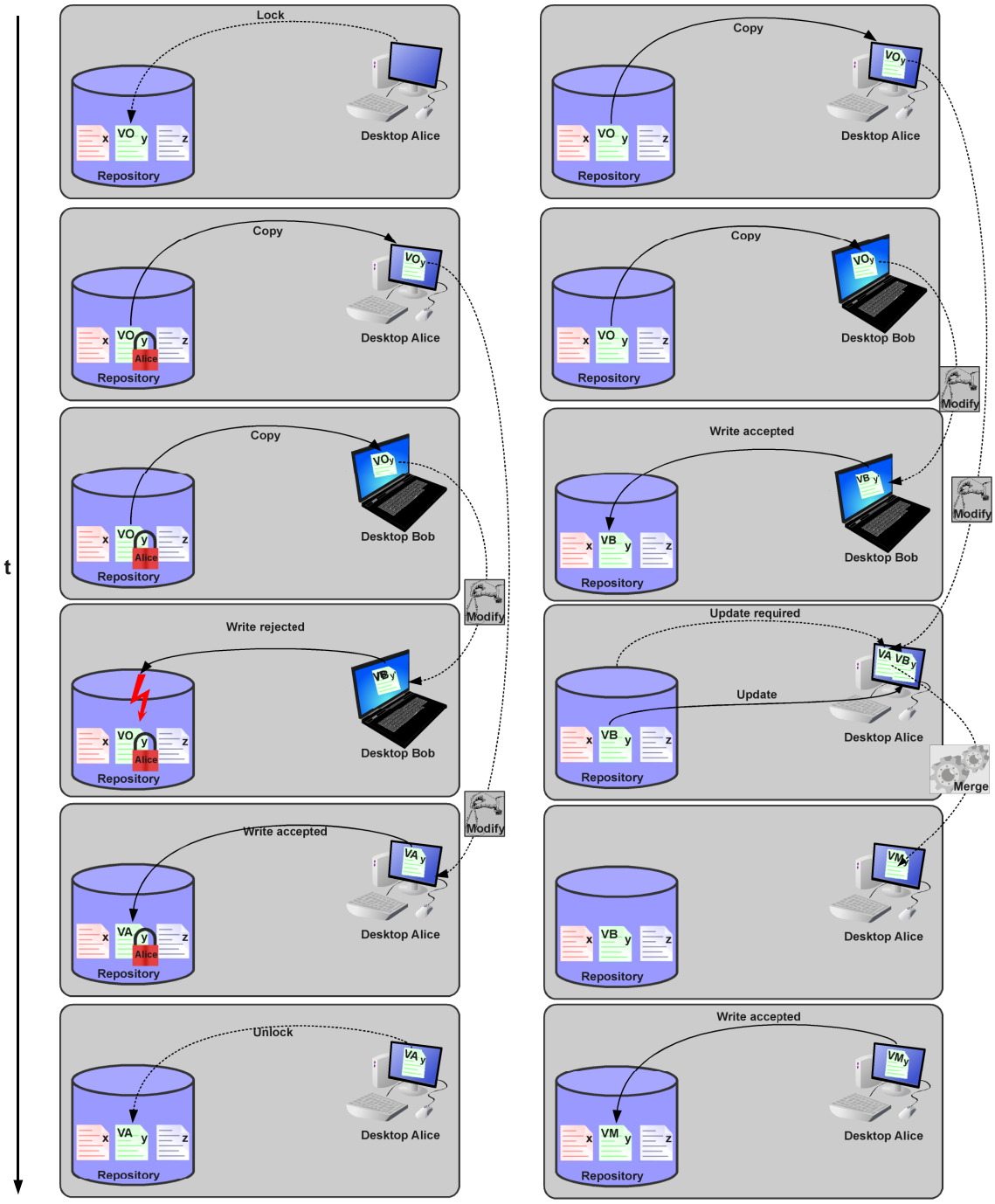


Abbildung II.2. Synchronisationsstrategien: Lock-Modify-Write vs. Copy-Modify-Merge

2.2 Inhaltliche Synchronisierung von Arbeitsergebnissen im Team

Zur Synchronisierung der Arbeitsergebnisse von Entwicklerteams sind zwei verschiedene Strategien in der Praxis vorherrschend – *Lock-Modify-Write* und *Copy-Modify-Merge*. (Prudêncio, Murta und Werner 2009) Beide Strategien sollen i. F. im Detail erläutert werden:

2.2.1 Lock-Modify-Write

Die *Lock-Modify-Write*-Strategie wird in der sog. pessimistischen Versionsverwaltung eingesetzt. Nach dieser Strategie werden die Bearbeitungsrechte für eine Version exklusiv an eine Person vergeben. Alle anderen Mitglieder des Teams können die Version zwar aus dem Repository lesen, dürfen diese aber zunächst nicht ändern.

Dies wird erreicht, indem das Repository die Speicherung von Nachfolgeversionen durch nicht berechtigte Bearbeiter ablehnt. Möchte ein Projektmitglied den Inhalt einer nicht gesperrten Dokumentversion verändern, so reserviert (engl. lock) es sich zunächst die entsprechenden Schreibrechte. Danach kann die entsprechende Person die gewünschte Bearbeitung (engl. modify) durchführen und das geänderte Dokument als neue Version im Repository speichern (engl. write).

Anschließend sollte die neu entstandene Version im Repository wieder zur Bearbeitung durch andere Projektmitglieder freigegeben (engl. unlock) werden. In Abbildung II.2 wird im linken Teil die beschriebene Bearbeitungsmethodik überblicksartig dargestellt.

Eines der auffälligsten Merkmale dieser Strategie ist, dass die gleichzeitige Bearbeitung einer Version durch mehrere Personen technisch verhindert wird. In diesem Punkt unterscheidet sich die Lock-Modify-Write-Strategie im Wesentlichen von der nachfolgend vorgestellten Strategie.

2.2.2 Copy-Modify-Merge

Die *Copy-Modify-Merge*-Strategie erlaubt die gleichzeitige Bearbeitung ein und derselben Version durch mehrere Bearbeiter. Sie ist das Schlüsselkonzept in der optimistischen Versionsverwaltung. Durch die parallele Bearbeitung entstehen mehrere voneinander unabhängige Versionen eines Dokumentes gleichzeitig (z. B. die Versionen *VA* und *VB* des Dokumentes *y* in Abbildung II.2 rechter Teil).

Um die parallele Entwicklung an einem Dokument wieder inhaltlich zu synchronisieren, müssen alle parallel durchgeführten Änderungen in einer Version integriert werden (engl. merge). Dieser Vorgang wird für Textdokumente und Quellcode durch aktuelle Versionsverwaltungssysteme (engl. *version control system – VCS*) weitgehend automatisiert durchgeführt.

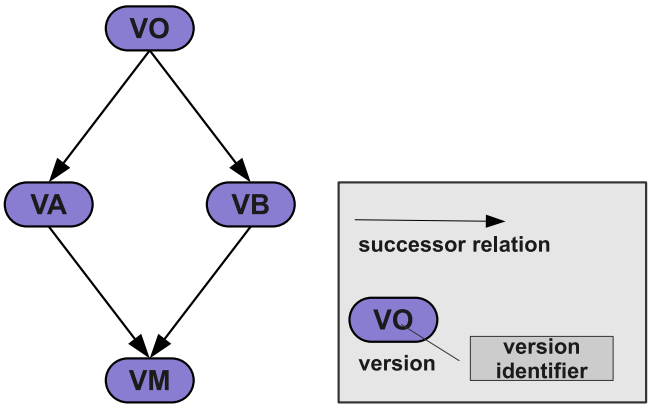


Abbildung II.3. logische Versionsbeziehungen des Beispiels in Abbildung II.2 rechter Teil

Durch die beschriebene Art der Synchronisierung teamweiter Dokumentbearbeitungen entstehen komplexere Beziehungsstrukturen zwischen Versionen. Abbildung II.3 zeigt die in dieser Arbeit verwendete grafische Notation zur Darstellung von Versionen und ihrer Beziehungen. Beziehungsstrukturen, die sich so beschreiben lassen, werden oft als Versionsgraph bezeichnet und lassen sich bei paralleler Entwicklung in Entwicklungszweige unterteilen:

Entwicklungszweig, Versionsgraph

Mit *Entwicklungszweig* (engl. *branch*) wird eine Sequenz von Revisionen bezeichnet, die durch aufeinanderfolgende Vorgänger- und Nachfolgerbeziehungen miteinander verknüpft sind. In jedem Entwicklungszweig gibt es genau eine Revision, die keine andere Revision aus dem Entwicklungszweig als Vorgänger hat (Start) und genau eine Revision, die keine andere Revision aus dem Entwicklungszweig als Nachfolger hat (Ende). Ein Entwicklungszweig kann als gerichteter Pfad wie in Abbildung II.4 dargestellt werden.

Mit *Versionsgraph* wird ein azyklischer, gerichteter Graph aus Versionen und ihren Vorgänger- und Nachfolgerbeziehungen bezeichnet (vgl. (Conradi und Westfechtel 1997)).

In Abbildung II.4 sind die Revisionen markiert, die mehrere Vorgänger haben. Zur Bestimmung des Inhalts der betroffenen Revisionen, muss ein sog. *Merge* durchgeführt werden. Dabei müssen *parallele Weiterentwicklungen* einer Version, *integriert* werden, damit der Inhalt einer Revision mit mehreren Vorgängern bestimmt werden kann. Das Merge-Verfahren spielt dabei die entscheiden Rolle für die optimistische Versionsverwaltung.

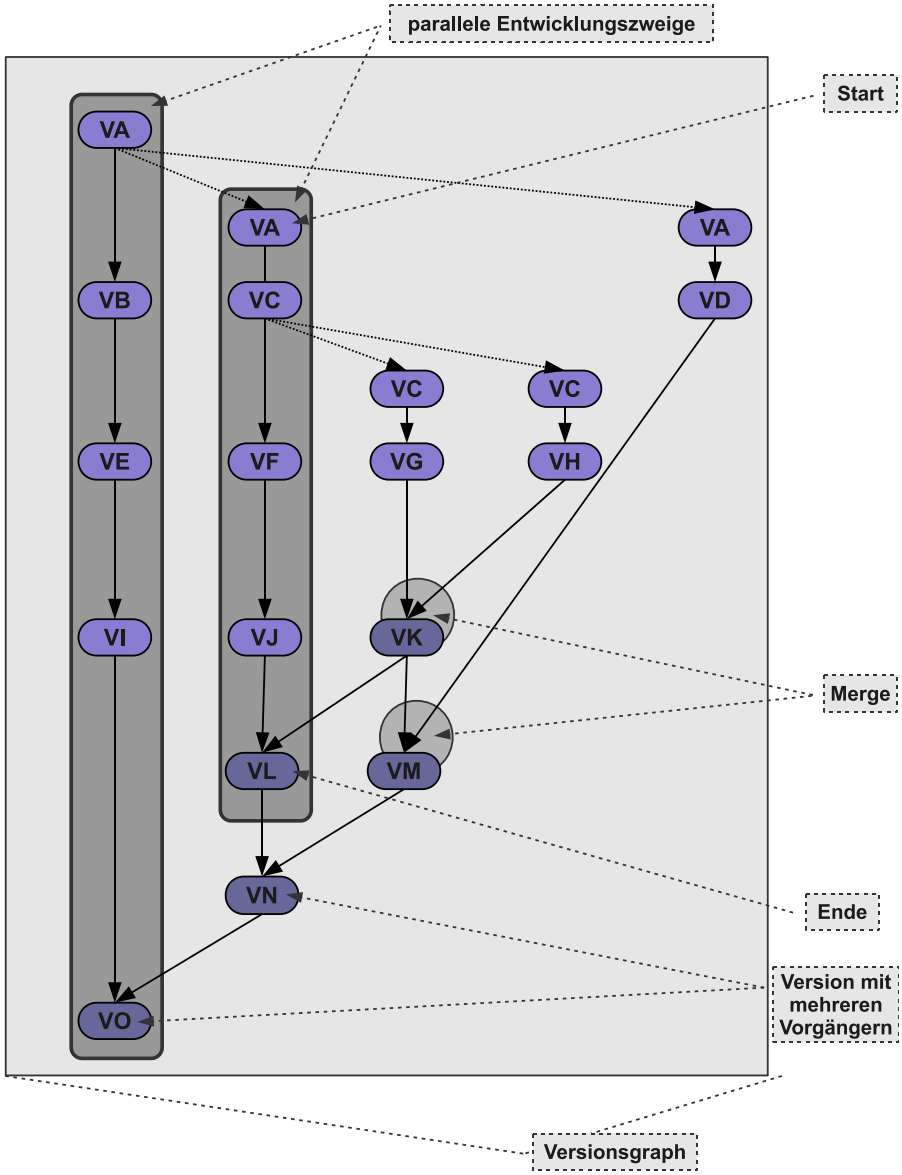


Abbildung II.4. Beispiel: Logische Sicht eines Versionsgraphen

2.2.3 Merge-Verfahren in der Versionsverwaltung

Arbeiten mehrere Teammitglieder parallel an den gleichen Versionen im Projektdatenbestand, so müssen unabhängigen Weiterentwicklungen einer Version von Zeit zu Zeit wieder integriert werden. Ohne eine geeignete Werkzeugunterstützung ist eine derartige Integration aber sehr aufwendig (Bendix 1995). Aus diesem Grund bieten alle optimistischen Versionsverwaltungssysteme eine automatische Unterstützung des Merge-Verfahren an. Dabei hat sich die Anwendung des 3-Wege-Merge-Prinzips (engl. *Three-Way-Merge*) (Mens 2002) weitgehend durchgesetzt. Es wird nachfolgend genauer erläutert.

2.2.4 3-Wege-Merge-Prinzip

Der Grundsatz hinter dem 3-Wege-Merge-Prinzip ist die kombinierte Anwendung parallel durchgeführten Änderungen. Abbildung II.5 illustriert dieses Vorgehen anhand eines Beispiels grafisch. Änderungen an einer Version werden dabei auf der Ebene von Dateneinheiten betrachtet. Im allgemeinen Fall kann eine Dateneinheit dabei in einer Revision hinzugefügt, geändert oder gelöscht worden sein.

In der Versionsverwaltung von Quellcode werden dazu Änderungen an Zeilen in Textdateien verarbeitet. Je nach Art der Änderungsanalyse unterscheidet man die verfügbaren Verfahren in zustandsbasierte Merge-Verfahren und operationsbasierte Merge-Verfahren.

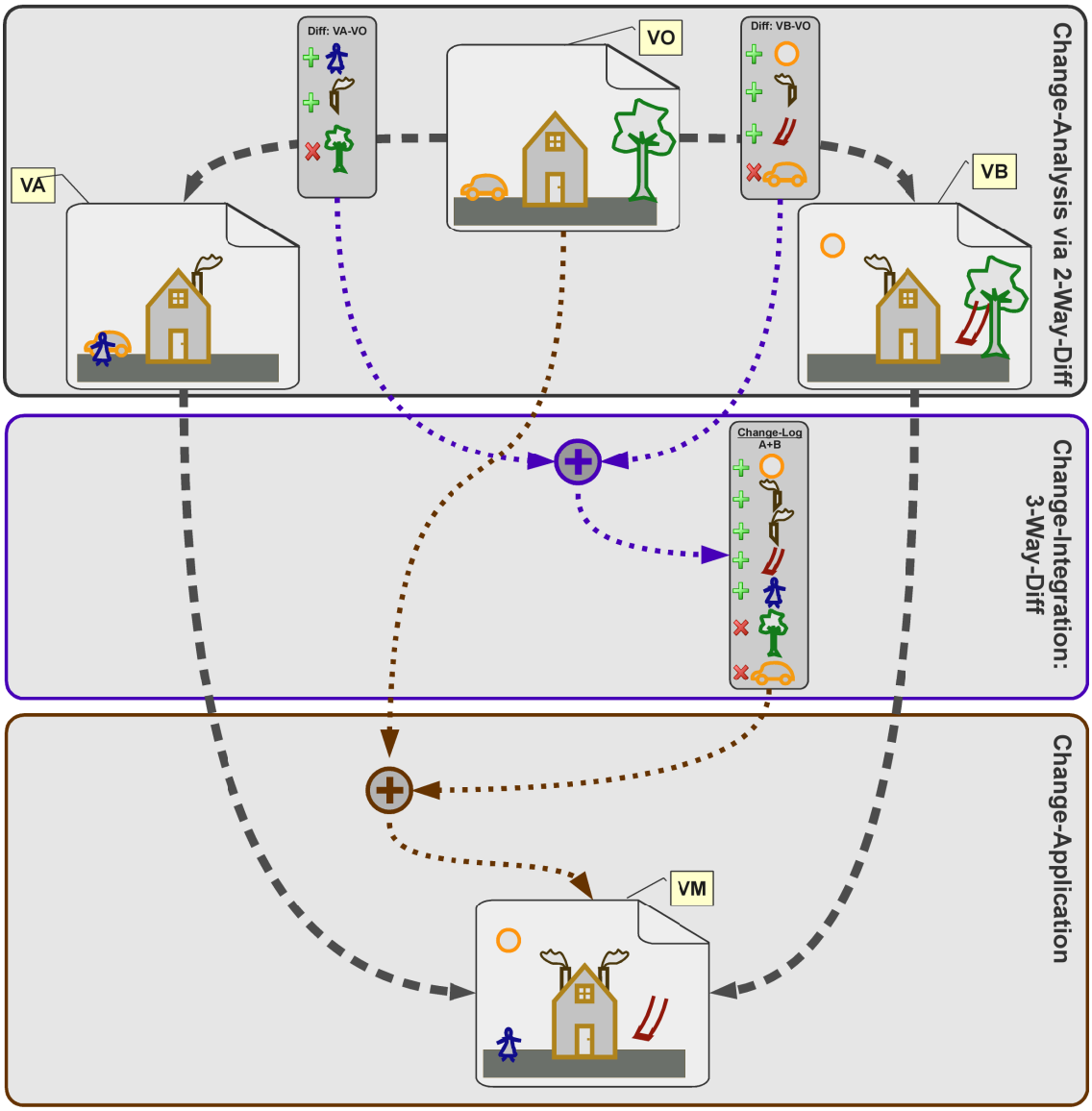


Abbildung II.5. Prinzip des 3-Wege-Merges am Beispiel

Bevor beide Arten der Änderungsanalyse vorgestellt werden, wird zunächst das Merge-Verfahren in einem Beispiel skizziert. Dazu stellt Abbildung II.5 die Behandlung von parallelen Änderungen im Bild dar.

Abbildung II.5 zeigt eine Version *VO*, die parallel geändert wird. Dabei entstehen zwei Versionen *VA* und *VB*. Die beiden Versionen *VA* und *VB* werden anschließend wieder zu einer Version mithilfe eines 3-Wege-Merge-Verfahrens integriert. Zur Anwendung des 3-Wege-Merge-Prinzips müssen drei Phasen durchlaufen werden. Diese sind erstens die Änderungsanalyse (engl. change analysis), jeweils zwischen Ausgangsversion (*VO*) und den parallelen Versionen (*VA*, *VB*), zweitens die Änderungsintegration (engl. change integration) und drittens die Anwendung der integrierten Änderungen auf die Ausgangsversion (*VO*) (engl. change application).

Zur Änderungsanalyse kommt im Beispiel ein 2-Wege-Vergleich (engl. 2-Way-Diff) zum Einsatz. Dabei werden im Beispiel die Versionen VO und VA bzw. VO und VB hinsichtlich Hinzufügungen und Löschungen von Elementen verglichen (Delta-Berechnung (Conradi und Westfechtel 1998)).

Der Vergleich im Beispiel ergibt, dass das Element *Baum* der Version *VO* in der Version *VA* gelöscht wurde zusätzlich aber ein *Kind* und ein *Schornstein* auf der rechten Dachseite hinzugefügt wurden. Das Ergebnis dieses Vergleiches ist in „Diff: *VA-VO*“ dargestellt. Analog kann ein 2-Wege-Vergleich zwischen *VB* und *VO* bestimmt werden. In der Phase der Änderungsintegration werden die Änderungsmengen *VA-VO* und *VB-VO* vereinigt, es entsteht ein 3-Wege-Diff-Ergebnis. Dieses wird in der letzten Phase – Änderungsanwendung – auf die Ausgangsversion *VO* angewendet, sodass die im Beispiel dargestellte Version *VM* entsteht.

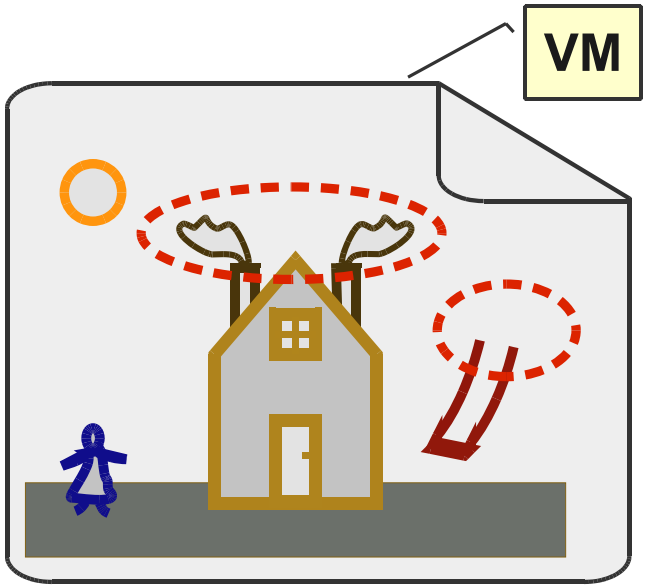


Abbildung II.6. Fehler im Merge-Resultat *VM* aus Abbildung II.5

Dabei kann die Version *VM* durchaus logische Fehler enthalten, obwohl die integrierten Versionen *VA* und *VB* jeweils fehlerfrei sind. In Abbildung II.6 sind die vorhandenen logischen Fehler in *VM* noch einmal markiert. Wie man in der Abbildung sieht, enthält *VM* ein Fragment einer *Schaukel*, da diese ohne Befestigung „in der Luft hängt“. Ferner werden die *Rauchwolken*

aus den *Schornsteinen* des *Hauses* in unterschiedliche Richtungen getrieben, was in der Realität nicht möglich ist.

Diese Fehler sind mit dem Wissen um die dargestellte Domäne eindeutig identifizierbar. Eine Beschreibung der Domänensyntax erlaubt somit die Erkennung von derartigen Fehlern im Rahmen der Validierung einer Beschreibung. Ob die Bearbeiter von *VA* und *VB* allerdings wirklich beabsichtigen, dass das *Haus* nach der Integration ihrer Versionen zwei *Schornsteine* bekommen soll, können nur sie selbst beantworten.

Wie oben erläutert ist die Identifizierung von Änderungen von Version zu Version ein grundlegender Baustein des Merge-Verfahrens. Die aktuell verfügbaren Arten der Änderungsanalyse lassen sich in die i. F. vorgestellten Arten zustandsbasierte und operationsbasierte Merge-Verfahren aufteilen.

2.2.5 Zustandsbasierte Merge-Verfahren

Die in der Praxis am weitesten verbreiteten Versionierungssysteme implementieren das zustandsbasierte Merge-Konzept (engl. state-based merge). Es sieht den Vergleich von Versionen vor, um Änderungen an Dateneinheiten zu identifizieren.

Das dazu genutzte Vergleichsverfahren wird mit *3-Wege-Diff* (engl. *Three-Way-Diff*) (Mens 2002) bezeichnet, wobei sich das Wort *Diff* vom englischen Begriff *Difference* in der Bedeutung von Unterschied herleitet. Beispielhaft wurde dieses Verfahren in Abbildung II.5 beschrieben. Voraussetzung für dieses Verfahren ist, dass die entsprechenden Versionen (bspw. *VA* und *VO*) in geeignete Einheiten zerlegt werden können, deren Hinzufügung bzw. Löschung automatisch analysiert werden kann.

Das zustandsbasierte Merge-Verfahren kann nicht zwischen der Änderung bzw. Löschung/Hinzufügung unterscheiden – vgl. dazu auch (Meisinger, Rausch und Sihling 2006) Somit sind die Änderungen (nur Hinzufügung bzw. Löschung), die durch dieses Verfahren analysiert werden können, äußerst unspezifisch. Dies birgt den Nachteil, dass bei der Änderungsintegration keine domänenspezifischen komplexen Änderungen oder auch der Bezug zwischen Änderungen berücksichtigt werden kann.

So kann beim Vergleich der Versionen *VA* und *VO* nicht automatisch bestimmt werden, ob die *Sonne* durch das *Auto* ersetzt wurde oder ob die *Sonne* nur gelöscht und das *Auto* hinzugefügt wurde. Der alleinige Vergleich zweier Schnappschüsse in der Dokumententwicklung ist Ursache für diesen Umstand.

Operationsbasierte Merge-Verfahren versuchen dieses Problem zu lösen und werden im nächsten Unterabschnitt erörtert. Trotz des angesprochenen Nachteils werden zustandsbasierte Merge-Verfahren in den meisten Fällen bevorzugt. Im Gegensatz zu operationsbasierten Verfahren sind sie unabhängig von einer Werkzeug-spezifischen Sprache zur Speicherung von Änderungen. Ihre universelle Einsetzbarkeit unabhängig von spezifischen Dokumenteditoren ist der Hauptgrund für ihre starke Verbreitung.

2.2.6 Operationsbasierte Merge-Verfahren

Im Gegensatz zu zustandsbasierten Merge-Verfahren muss bei vielen Ansätzen zu operationsbasierten Merge-Verfahren (engl. operation-based merge) kein Vergleich von Versionen in der Änderungsanalysephase berechnet werden. Bei manchen Ansätzen wird versucht, spezifische Änderungsoperationen aus zwei aufeinanderfolgenden Versionen „herauszurechnen“ (Dantas, Murta und Werner 2005).

Viele operationsbasierte Merge-Verfahren setzten aber voraus, dass die Änderungen zwischen Versionen explizit durch ein Bearbeitungswerkzeug für entsprechende Dokumente erfasst werden. Somit stehen für die Integration von Versionen entsprechende parallele Änderungsprotokolle zur Verfügung. Mit den darin dokumentierten Änderungen und einer Vorgängerversion lässt sich jede Version in der Historie bestimmen.

Je nach Sprache zur Speicherung der Änderungsprotokolle lassen sich komplexe Änderungsoperationen (bspw. Ersetzungen von Elementen und domänenspezifische Änderungsoperationen) abbilden. Ein großer Vorteil operationsbasierte Merge-Verfahren ist die Ermöglichung einer sehr aussagekräftigen automatischen Konsistenzanalyse asynchroner Änderungen im Vergleich zu zustandsbasierten Verfahren (Ignat und Norrie 2004; Koegel, Hermannsdoerfer, von Wesendonk u. a. 2010). Der entscheidende Nachteil operationsbasierter Merge-Verfahren ist allerdings ihre Werkzeugabhängigkeit.

Voraussetzung für die Etablierung entsprechender Merge-Verfahren, die die erhöhte Leistungsfähigkeit in der Konsistenzanalyse nutzen, ist eine standardisierte, werkzeugübergreifende Speicherung Format-spezifischer (bzw. modellierungssprachspezifischer) Änderungsoperationen. Dadurch könnten bspw. Dokumente, die von unterschiedlichen Werkzeugen bearbeitet wurden, mit entsprechenden Merge-Techniken verwaltet werden. Eine andere Alternative, an der zurzeit geforscht wird, ist eine generischer Ansatz zur Aufzeichnung spezifischer Änderungen (Herrmannsdoerfer und Koegel 2010).

Im Umfeld des modellbasierten Software Engineerings wird vereinzelnd am Einsatz operationsbasierten Verfahren geforscht (vgl. (Helming u. a. 2009)). Auch im Gebiet des Variantenmanagements denkt man über die Anwendung der genannten Verfahren nach (vgl. (Ternité 2009).

2.2.7 3-Wege-Merge für Textdaten – Text-Merge-Verfahren

Wie schon angesprochen wird das 3-Wege-Merge-Prinzip am häufigsten in der Versionsverwaltung von Textdateien eingesetzt und hier insbesondere zur Verwaltung von Quellcode. Zur Anwendung des erwähnten 3-Wege-Vergleichs werden die Änderungen an Textdateien, wie bspw. Java-Dateien in Abbildung II.7, auf Basis von Textzeilen verglichen. Textzeilen entsprechen somit den Dateneinheiten, die der 3-Wege-Vergleich des Merge-Verfahrens betrachtet.

Im Beispiel in Abbildung II.7 sieht man, dass die zwei mit (i) markierten Textzeilen in der linken Version hinzugefügt wurden und dass die mit (ii) markierte Zeile in der rechten Version hinzugefügt wurde. Ferner unterscheidet sich die mit (iii) markierte Zeile in beiden Versionen voneinander. Schaut man sich den Inhalt der Zeile an, so sieht man, dass sich in dieser Zeile der Vergleichsoperator geändert hat. Für den zeilenbasierten Vergleichsmechanismus ist allerdings nicht zu erkennen (typisch für zustandsbasierte Merge-Verfahren), ob es sich um eine Änderung der Zeile oder um die Löschung der „alten“ und Hinzufügung einer „neuen“ Zeile handelt.

Da keine gleichen oder direkt benachbarten Zeilen in dem Beispiel geändert wurden, kann der 3-Wege-Merge auf Basis der identifizierten Änderungen erfolgreich abgeschlossen werden. Das Ergebnis ist in Abbildung II.8 dargestellt.

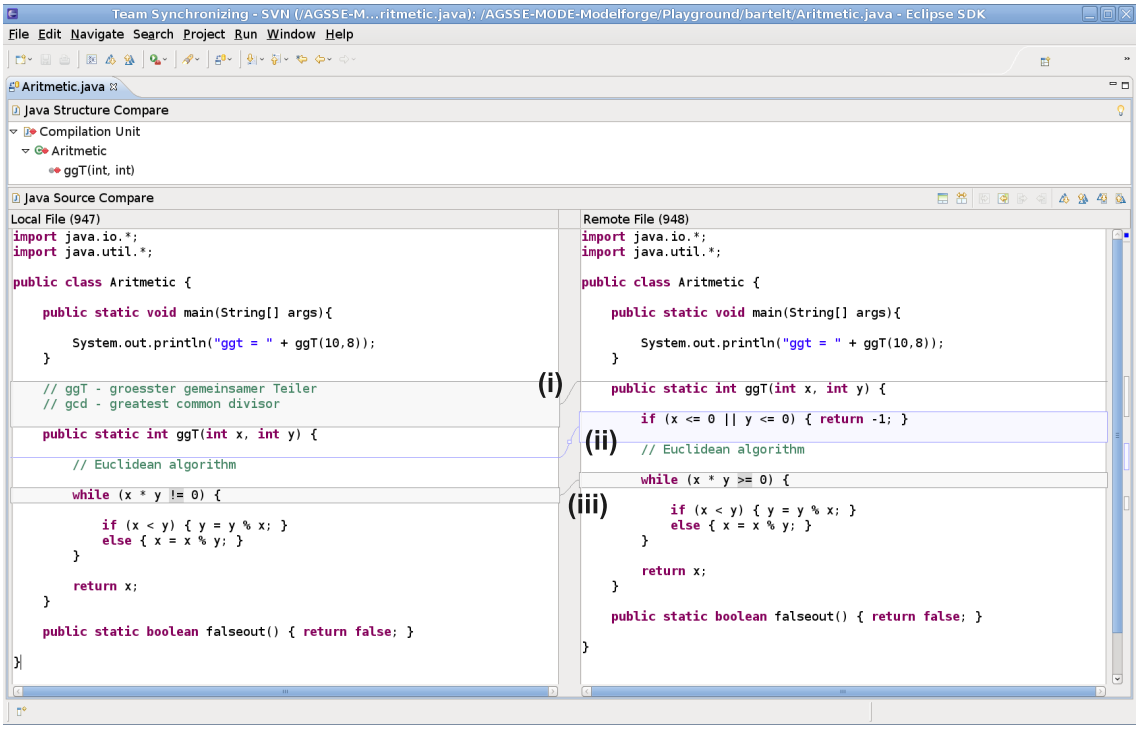


Abbildung II.7. 3-Wege-Vergleich von Quellcode durch Subversion in Eclipse am Beispiel

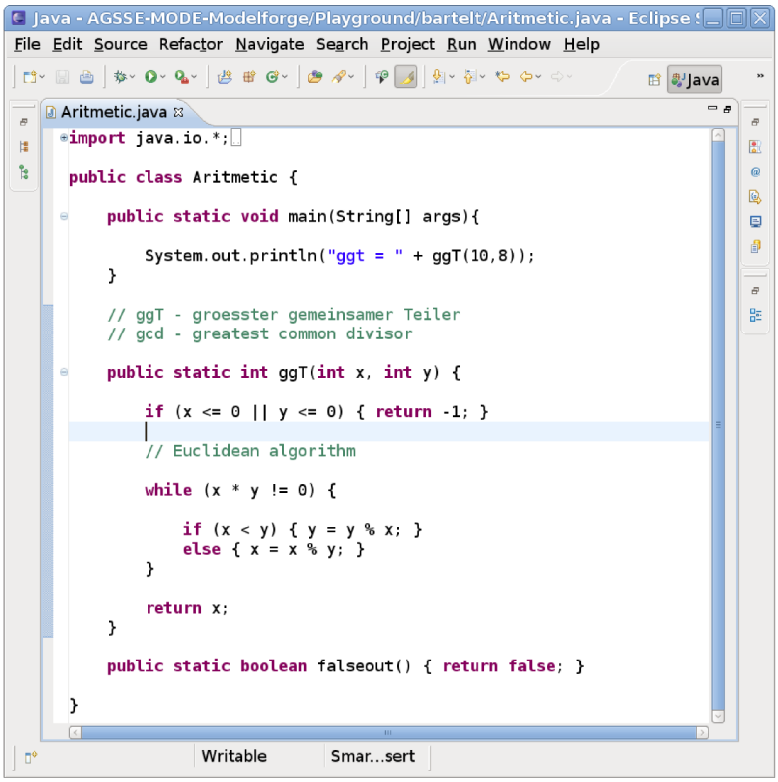


Abbildung II.8. Ergebnis des 3-Wege-Merges nach Vergleich in Abbildung II.7

Nachdem nun das grundlegende Prinzip und die Anwendung des Merge-Verfahrens vorgestellt wurden, das im kollaborativen Software Engineering am erfolgreichsten eingesetzt wird, wird im Abschnitt 3 auf die modellbasierte Softwareentwicklung eingegangen. Die Eigenschaften von Softwaremodellen und ihre im Vergleich zu Code und Textdateien andersartige formale Syntax verhindern eine unangepasste Anwendung etablierter optimistischer Versionierungssysteme bei der Verwaltung von Modellinformationen. Doch bevor auf die Eigenschaften von Softwaremodellen eingegangen wird stellt Abschnitt 2.3 die populärsten Versionsverwaltungssysteme des Software Engineerings überblicksartig vor:

2.3 Aktueller Stand der SCM-Werkzeugunterstützung

Zur Unterstützung der Versionsverwaltung bei Entwicklungsprozessen in Teams gibt es eine Vielzahl an Softwarewerkzeugen. Diese Versionsverwaltungssysteme konzentrieren sich auf die Unterstützung von Entwicklerteams in der Quellcodeverwaltung. Unterschieden werden können Versionsverwaltungssysteme in zentrale und verteilte Ansätze. Diese und ihre populärsten Vertreter werden i. F. überblicksartig erörtert.

2.3.1 Zentrale Versionsverwaltung

Zentrale Versionsverwaltungssysteme unterliegen i. d. R. einer Client-Server-Architektur. Der Server übernimmt dabei die Aufgabe, zentrale Repositorys für komplette Projektteams zu verwalten. Dabei veröffentlicht er in einem Netzwerk für die Entwickler eines Projektteams den aktuellen Datenbestand und die ggf. kommentierte Entwicklungshistorie. Die populärsten Open-Source-Systeme in diesem Bereich sind die auf den Grundlagen des *Revision Control*

Systems (*RCS*) (Tichy 1985) basierenden Systeme *Concurrent Versioning System* (*CVS*) (GNU Project 2010b) und *Subversion* (*SVN*) (Tigris.org 2010).

Kommerzielle Werkzeuge vernetzen zum Teil die Versionsverwaltung noch mit anderen Aktivitäten zur Teamkoordination wie bspw. Projektplanung und -controlling, synchrone Teamkommunikation oder Änderungsmanagement. Populäre Systeme in diesem Bereich sind *IBM Jazz Team Concert* (IBM 2010b), *IBM Clearcase* (IBM 2010c), *Microsoft Team Foundation Server* (Microsoft 2010) und *Borland StarTeam* (Borland 2010b).

2.3.2 Verteilte Versionsverwaltung

Bei der verteilten Versionsverwaltung wird von den Entwicklern kein zentrales Repository genutzt. Jeder Entwickler führt neue Versionen seinem „eigenem lokalen (Änderungs-)Repository“ zu. Außerdem können auch anderen Teammitgliedern eigene Versionen bzw. sog. Patches zur Verfügung gestellt werden.

Patches dokumentieren i. d. R. zeilenbasiert Änderungen an einer Textdatei. Patches können auch wieder auf einer parallel entwickelten Version „eingespielt“ werden, sodass auch hier parallele Änderungen auf Textzeilenbasis in einer Version integriert werden können. Bei verteilten Versionierungsansätzen arbeiten alle Teammitglieder unabhängig voneinander (Clatworthy 2007) und übernehmen mehr in Eigenverantwortung die Synchronisation ihrer inhaltlichen Änderungen.

Viele der verbreitetsten Softwarewerkzeuge wi*e Bazaar* (Canonical Ltd. 2010), *Git* (Git Authors 2010) und *Bitkeeper* (BitMover Inc. 2010) erlangten ihre Bekanntheit durch ihren Einsatz bei der Entwicklung des Linux-Kernels.

3 Modellbasierte Softwareentwicklung

Modelle sind seit je her in allen Ingenieurwissenschaften ein probates Mittel zur Unterstützung der Kommunikation. Dabei sind drei elementare Charakteristika – Abstraktionseigenschaft, Darstellungseigenschaft und Abbildungseigenschaft – allen Modellen gemein.

Die Abstraktionseigenschaft erlaubt die Vernachlässigung bestimmter Eigenschaften des zu modellierenden Gegenstandes, die für den Problemkontext, in dem das Modell Einsatz findet, unbedeutend sind. Die Darstellungseigenschaft erlaubt eine für Menschen (eindeutig) interpretierbare Ausdrucksform der modellierten Informationen. Diese Ausdrucksform wird auch als die konkrete Syntax eines Modells bezeichnet. Im Allgemeinen handelt es sich dabei um eine Repräsentation durch grafische Symbole, um eine textuelle Repräsentation oder um Mischformen davon. Das Abbildungsmerkmal verlangt, dass zu jedem Modell ein entsprechendes Original ausgeprägt ist oder in Zukunft ausgeprägt sein kann. Nach (Stachowiak 1973) besitzen Modelle darüber hinaus das pragmatische Merkmal, da sie immer für einen bestimmten Zweck geschaffen werden.

Im Bereich der Informatik werden Modelle zum Hard- und Softwareentwurf eingesetzt. Um die physische und logische Struktur und das Laufzeitverhalten von Softwaresystemen zu beschreiben, kommt eine Vielzahl an Beschreibungstechniken zum Einsatz. Das Software Systems Engineering ist hierbei die Disziplin, die für softwarebasierte Systeme Beschreibungstechniken entwickelt und darüber hinaus zahlreiche Methoden und Werkzeuge für den Entwurf und die Pflege und Wartung dieser Systeme bereitstellt.

Wie in den nachfolgenden Abschnitten beschrieben wird, konzentriert sich die modellbasierte Softwareentwicklung, als ein Teilgebiet des Software Systems Engineerings, auf die Erforschung von modellbasierten Entwurfsmethodiken, auf das Design von Modellierungssprachen sowie auf die Erforschung von Verfahren und Werkzeugen zur Verarbeitung von Modellen.

Die folgenden Abschnitte geben dazu eine kompakte Einführung in die Grundlagen der modellbasierten Softwareentwicklung. Dabei konzentrieren sich die Erläuterungen auf die Einführung zentraler Begriffe und die Beschreibung von Methoden und Technologien, die in den nachfolgendenden Kapiteln von besonderer Bedeutung sind.

3.1 Modellbasierte Entwicklung von Software Systemen

Während der Softwareentwicklung insbesondere großer, langlebiger Systeme spielen Modelle eine wichtige Rolle. Sie erfüllen dabei unterschiedliche Aufgaben. In der Analysephase dienen sie zum Abgleich der Kundenanforderungen mit Analyse- und Entwurfsmodellen. Während der Entwurfsphase dienen einzelne Modellversionen den Softwareingenieuren als Hilfsmittel zur fachlichen Kommunikation und zur inhaltlichen Synchronisation ihrer Arbeit. In der Realisierungsphase sind Modelle eine verbindliche Spezifikation für die Programmierung bzw. dokumentieren den Quellcode, um Wartung, Anpassung und Erweiterung des Systems zu vereinfachen.

Die Modellierung von Softwaresystemen in größeren Teams ist vorrangig für die Analyse- und Entwurfsphase von Bedeutung und wird in den nachfolgenden Kapiteln genauer betrachtet. Die Teamarbeit an Modellen fordert von den beteiligten Ingenieuren ein fundiertes Wissen über die Syntax der verwendeten Modellierungssprache und ein einheitliches Verständnis zu deren Bedeutung.

Nur so kann das Auftreten von Missverständnissen bzgl. der gemeinsamen modellierten Inhalte vermindert werden. Bevor in Abschnitt 3.3 auf die syntaktische Spezifikation eingegangen wird, sollen zunächst einige im Software Engineering weitverbreitete Modellierungssprachstandards vorgestellt werden. Auf deren Basis wird in Abschnitt III.3 ein Anwendungsbeispiel vorgestellt, das der Illustration der in dieser Dissertation vorgestellten Lösungskonzeption dient.

3.1.1 Modellierungssprachen in der Softwareentwicklung

Im Bereich des Software Engineering hat sich in den letzten Jahren immer breiter die syntaktische und semantische Standardisierung von Modellen mithilfe von wohldefinierten Modellierungssprachen durchgesetzt. Je nach Formalisierungsgrad gibt eine Modellierungssprache verschiedenen Akteuren die Möglichkeit Modelle zu erstellen, die von allen konform interpretiert werden können. Obwohl Software im Gegensatz zu anderen Engineering-Produkten ein immaterielles Gut ist, kann ihre logische Struktur und ihr Verhalten als textuell beschriebenes oder grafisches Modell in einem Dokument festgehalten werden.

Struktur und Verhaltensbeschreibungen

Üblicherweise wird in der Softwaremodellierung nach Strukturbeschreibung und Verhaltensbeschreibung unterschieden. Im Software Engineering haben sich dafür eine Reihe von informell, semi-formell und formal spezifizierten Modellierungssprachen etabliert. Neben der beispielhaften Illustration dieser Sprachen sollen sie überblicksartig erläutert werden.

Entity Relationship Modelle und Klassenmodelle:

Entity Relationship Modelle (*ER-Modelle*) gehörten zu den ersten syntaktisch formal standardisierten Beschreibungstechniken zur Datenmodellierung, die in der Informatik (Chen 1976) genutzt wurden. Diese Modelle erlauben die Modellierung von Datenstrukturen auf Basis von getypten, attributierten Entitäten und getypten Beziehungen. Entitätstypen beschreiben dabei eine Menge gleichartiger Entitätsausprägungen, die eine Entsprechung in der realen Welt haben. Gleichartig bedeutet in diesem Zusammenhanghang, dass die betreffenden Entitäten Werte für die gleichen Eigenschaften (Attribute, Beziehungen) belegen.

Im Zuge der Verbreitung objektorientierter Sprachen wurde die Beschreibungstechnik der Entity Relationship Modelle zu *Klassenmodellen* weiterentwickelt. Mit Klassenmodellen können den Entitätstypen neben den Attributen auch Operationen zugewiesen werden. Dies ermöglicht es im Sinne der Objektorientierung, Daten explizit in Beziehungen zu deren Verarbeitung zu setzen. Die entsprechenden Entitätstypen werden in diesem Zusammenhang *Klassen* und ihre Ausprägungen *Objekte* genannt. Weiterhin wurde für Klassenmodelle eine schier unüberschaubare Anzahl syntaktischer Erweiterungen veröffentlicht. Zur diesbezüglichen Recherche sei hierbei auf die UML (Object Management Group (OMG) 2010c) verwiesen, die entsprechende Erweiterungen im großen Stil aufgreift und versucht, diese in einem gemeinsamen Standard einzubetten. Bei der Spezifikation von Modellierungssprachen durch Metamodelle hat sich der Einsatz von Klassendiagrammen zur formalen Syntaxbeschreibung weitgehend durchgesetzt.

Komponentenmodelle:

Komponentenmodelle (engl. *component models*) gehören zu den Strukturbeschreibungen für Softwaresysteme. Sie dienen der Kapselung einzelner Softwarebausteine. Die Beschreibungsmittel von Komponentenmodellen sind Komponenten (oder Komponententypen), Schnittstellen und Beziehungen zwischen Schnittstellen und Komponenten. Diese werden unterteilt in Schnittstellenrealisierung (engl. *interface realization*) und Nutzungsbeziehungen (engl. *usage relation*). Dabei bedeutet die Verbindung einer Komponente mit einer Schnittstelle über eine Schnittstellenrealisierung, dass die Komponente eine Implementierung für das durch die Schnittstelle beschriebene Verhalten bereitstellt. Im Sinne der Kapselung soll bei der Nutzung von Komponentenmodellen von einer detaillierten Beschreibung ihrer Implementierung abstrahiert werden. Detailliertere Beschreibungstechniken wie Kompositionsstrukturdiagramme (*engl. composition structure model*) ermöglichen zusätzlich die Beschreibung von Kompositionshierarchien von Komponenten. Weitere Details über syntaktische Möglichkeiten und die Semantik von Komponentenmodellen und Kompositionsstrukturdiagrammen können (Rupp, Queins und Zengler 2007) entnommen werden.

Anwendungsfallmodelle:

Anwendungsfallmodelle (*engl. use case models*) werden in der Anforderungsanalyse eingesetzt, um das Verhalten eines Softwaresystems anhand der gewünschten Interaktion mit den Benutzern zu beschreiben. Dabei wird von der internen Interaktion zwischen einzelnen Systemkomponenten abstrahiert. Die UML bietet zur Modellierung in diesem Bereich Anwendungsfalldiagramme an. Zentrale Sprachmittel in Anwendungsfallmodellen sind die *Anwendungsfälle* selbst, *Akteure* als natürliche Personen oder andere externe Systeme. Die durch die Anwendungsfälle auszudrückende Bedeutung ist meistens informell durch die Verwendung sinnvoller natürlichsprachlicher Namen beschrieben. Darüber hinaus können

Anwendungsfälle durch Ablaufbeschreibung, Vor- und Nachbedingungen, Ausnahmen, Regeln, u. v. m. detaillierter beschrieben werden. In jedem Anwendungsfall muss mindestens ein Akteur mit dem System, das den Anwendungsfall realisiert, interagieren. Anwendungsfälle werden i. d. R. aus den funktionalen Anforderungen der zukünftigen Anwender des Systems abgeleitet. Sie beschreiben das nach außen hin für den Anwender sichtbare Verhalten eines Softwaresystems. Anwendungsfallmodelle bilden oft den Ausgangspunkt für die Verfeinerung der Softwarespezifikation hin zur Softwarearchitektur durch Strukturierung und Kapselung von funktionalen Verhaltensaspekten in softwarearchitektonische Einheiten.

Aktivitätsdiagramme:

Aktivitätsdiagramme sind eine Form der Verhaltensmodellierung. Sie erlauben die Beschreibung von Kontroll- und Datenflüssen durch eine Vielzahl von Sprachmitteln. Die Grundstruktur von Aktivitätsdiagrammen sind gerichtete Graphen, deren Knoten unterschiedlichen Modellelementen entsprechen können und durch deren Kantenausrichtung der Kontroll- oder Datenfluss beschrieben wird. Die grundlegenden Modellelemente von Aktivitätsdiagrammen sind *Aktionen*, die die Ausführung von Anweisungen beschreiben. Neben den Aktionen existieren weitere Modellelemente wie Alternativen, Schleifen, die durch Verzweigungs- und Vereinigungsknoten zwischen den Kontrollflusskanten ausgedrückt werden. Parallelabläufe und deren Synchronisierungen werden durch Split- und Join-Knoten beschrieben. Der Datenfluss wird mithilfe von sog. Objektknoten beschrieben, die als Ein- bzw. Ausgangsparameter für die nachfolgende bzw. vorangegangenen Aktionen dienen. Die Semantik von Aktivitätsdiagrammen in der UML 2.0 ist auf Basis der Tokensemantik von Petrinetzen festgelegt (Rupp, Queins und Zengler 2007).

3.1.2 Universelle Standards versus domänenspezifische Modellierung

Bis Ende der 80iger Jahre wurden eine Vielzahl verschiedener grafischer Notationen zur Beschreibung von Software genutzt. Da in den Anfängen diese Notationen nicht präzise genug beschrieben und standardisiert waren, ergaben sich während des Entwurfs zusätzliche Abstimmungsaufwände bzgl. Form und Bedeutung der verwendeten Sprachelemente bzw. traten diesbzgl. Missverständnisse auf.

Um diesem Missstand zu begegnen, wurden Initiativen gestartet, die erstmals mit der Veröffentlichung der ersten UML-Version eine umfangreiche Sammlung aufeinander abgestimmter Beschreibungstechniken zur Software- und Systementwicklung zu Verfügung stellten (Booch, Rumbaugh und Jacobson 1998). Auf Basis der verschiedenen, bereits bis Ende der 80iger Jahre verbreiteten grafischen Modellierungsansätze für die Softwareentwicklung wurde mit der UML ein universeller Sprachstandard geschaffen, der heutzutage zum Fachwissen der meisten Softwareingenieure zählt.

Dabei bietet die UML grafische Sprachmittel, die sich in der Praxis zur intuitiven Beschreibung von Softwaresystemen bewährt haben und die international allgemein anerkannt sind. Weiterhin bietet die UML in einem beschränkten Rahmen domänenspezifische Erweiterungen und Anpassungen ihrer Spezifikation an.

Zusätzlich zur UML werden in der modellbasierten Softwareentwicklung auch domänenspezifische Sprachen eingesetzt, für die die UML-Semantik nicht spezifisch genug ist bzw. nicht entsprechend angepasst werden kann (Selic 2007). Sowohl die Softwareentwicklung auf Basis der UML als auch auf Grundlage domänenspezifischer Sprachen wird i. F. skizziert.

Unified Modelling Language (UML)

Mitte der 90iger Jahre wurden aus einer Reihe von bis dahin getrennt entwickelten, hauptsächlich objektorientierten Modellierungsmethoden (Booch, Rumbaugh und Jacobson 1998) die ersten Versionen der *Unified Modeling Language* (*UML*) (Object Management Group (OMG) 2010c) erstellt und veröffentlicht. Im Jahr 1997 wurde die UML dann bei der Object Management Group (OMG) zur Standardisierung eingereicht. Seitdem steht die Weiterentwicklung der UML unter der Hoheit der OMG.

Die UML bietet eine Vielzahl an Möglichkeiten zur grafischen Struktur und Verhaltensmodellierung von (Software-) Systemen. Die Vereinheitlichung bewährter syntaktischer Sprachelemente zur Software und Systembeschreibung, die Standardisierung der Sprache durch die OMG und ihre freie Verfügbarkeit sind wesentliche Gründe für ihre weltweite Verbreitung. So dienen UML-Modelle heute vielen Softwareingenieuren als Ausdrucksmittel für einen fachlichen Informationsaustausch.

Um eine möglichst eindeutige Beschreibungstechnik mit der UML zu schaffen, wurde im Rahmen ihrer Standardisierung eine semi-formale Spezifikation der Sprache veröffentlicht. In dieser wird durch ein Metamodell zur Beschreibung gültiger abstrakter Syntaxgraphen, zusätzlichen OCL-Bedingungen (sog. OCL-Constraints) und erklärenden Text die Syntax und Semantik der Sprache beschrieben

Die Sprachsyntax wird dabei durch das Metamodell vollständig formal beschrieben. Neben dem Syntaxgraphen werden weitere Bedingungen an die Syntax der Sprache gestellt. Diese sog. statische Semantik wird durch die OCL-Bedingungen formal spezifiziert. Diese formalen Teile der Spezifikation ermöglichen eine automatische syntaktische Korrektheitsanalyse, welche für eine effiziente Suche nach Modellierungsfehlern in großen Modellen unerlässlich ist.

Zwar unterstützt die UML die Modellierung von (Software-) Systemen in allen Entwurfsphasen, ihre Beschreibungsmittel sind aber innerhalb des Bereiches der Software- und Systementwicklung allgemein gehalten und domänenunabhängig.

Zur „leichtgewichtigen“ Anpassung der Sprache selbst bietet die UML das Konzept der Profile an (Robert u. a. 2009). Profile ermöglichen es, den Sprachstandard hinsichtlich domänenspezifischer Erweiterungen in einem eng begrenzten Rahmen anzupassen, ohne eine vollständig neue Modellierungssprache definieren zu müssen (sog. *interne DSL*).

Für umfangreiche („schwergewichtige“) Änderungen sind Profile allerdings nicht ausreichend. Für diesen Zweck bietet das Forschungsgebiet der modellgetriebenen Softwareentwicklung (engl. *Model Driven Architecture* – MDA) systematische Entwurfsmethoden zur Erstellung von domänenspezifischen Sprachen (sog. *externe DSL*) an, die im folgenden Abschnitt eingehender diskutiert werden.

Domänenspezifische Sprachen (DSL)

Die Entwicklung der UML war ein wichtiger Schritt, um eine universelle Modellierungssprache zur präziseren Spezifikation und zur fachlichen Kommunikation auf verschiedenen Abstraktionsniveaus im Software Engineering zu schaffen. Software-Engineering-Produkte werden in fast allen Bereichen des persönlichen und industriellen Alltags zur maschinellen Informationsverarbeitung eingesetzt. Sie sind somit mehr als Produkte

anderer Ingenieurwissenschaften in einen externen Anwendungskontext (sog. Domäne) integriert.

Aufgrund der Vielfältigkeit der Anwendungsgebiete können universelle Modellierungssprachen wie die UML nur zur Beschreibung sehr allgemeiner, domänenunspezifischer Softwarestrukturen verwendet werden. Die Nutzung domänenspezifischer, bedeutungsmäßig sehr ausdrucksstarker Entitäten und Beziehungen ist nur auf Basis einer domänenspezifischen Anpassung der UML durch domänenspezifische Profile oder durch die Verwendung domänenspezifischer Modellierungssprachen (engl. *domain specific language – DSL*) möglich.

Durch das Design ausdrucksstarker Modellierungssprachen gelingt es, allgemeingültiges Domänenwissen schon in der Spezifikation der Modellierungssprache formal zu verankern (Kleppe 2008). Dieses Vorgehen ermöglicht, die Beschreibung unkorrekter Sachverhalte in einem domänenspezifischen Modell schon bei der Analyse seiner Syntax automatisch zu erkennen. Abbildung II.9 und Abbildung II.10 verdeutlichen diesen Vorteil durch ein Beispiel.

In Abbildung II.9 sieht man ein exemplarisches Metamodel, das die „Eltern-Beziehung“ in Bezug auf männliche und weibliche Personen beschreibt. Dabei gibt die oben abgebildete OCL-Bedingung an, dass jede Verwandtschaftsrolle eines Menschen nur maximal eine Instanz einer männlichen und nur eine einer weiblichen Rolle als Vater oder Mutter haben kann.

Im dazugehörigen domänenspezifischen Modell wurden über die „Vater-Beziehung“ zwei Modellelemente verknüpft, was die angegebene OCL-Bedingung verletzt. Durch die formale Beschreibung der Verwandtschaftszusammenhänge im domänenspezifischen Metamodell also schon auf Ebene der Sprachsyntax wird eine sehr detaillierte automatische Validierung des Modells ermöglicht.

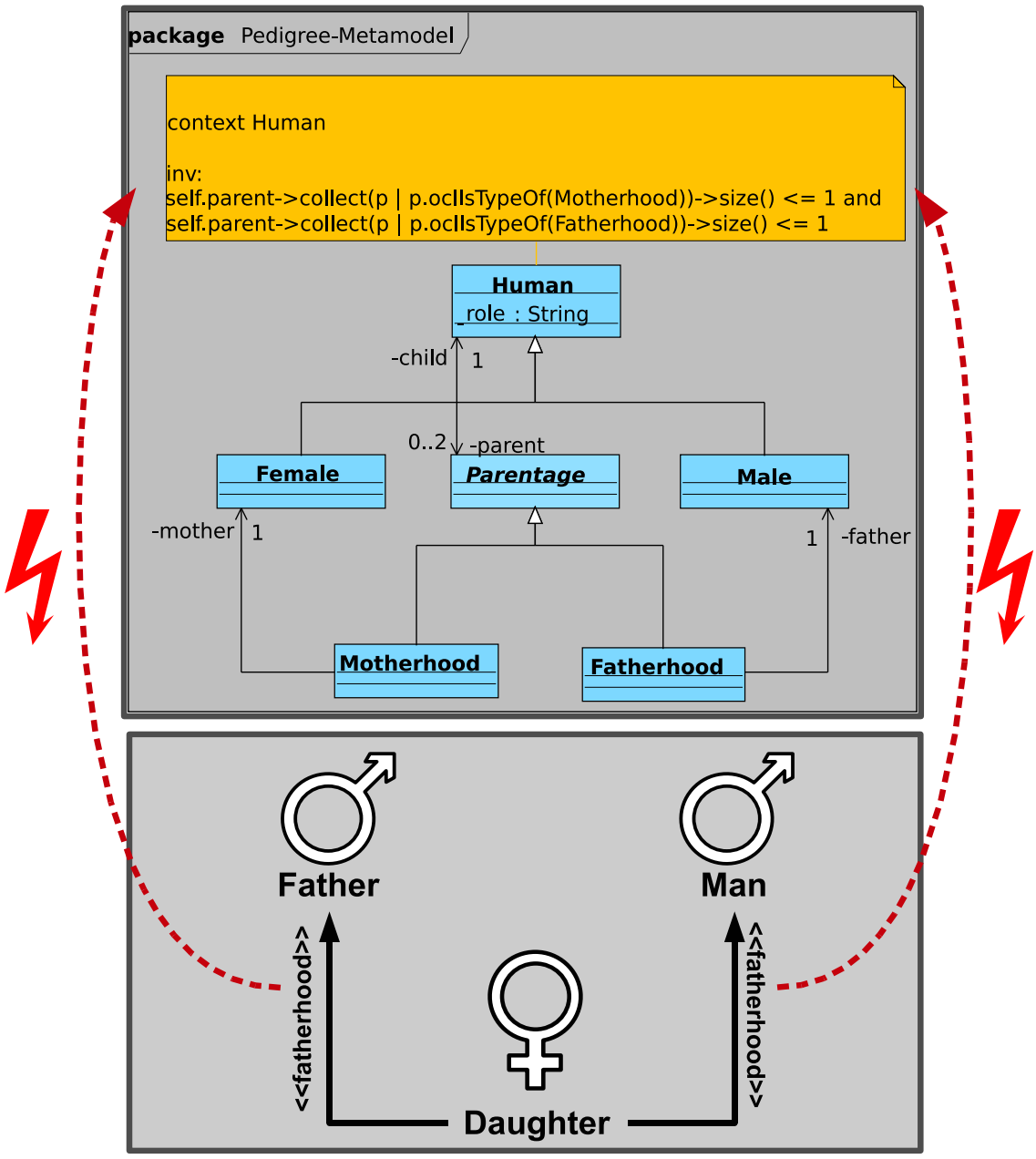


Abbildung II.9. Ausdrucksfähigkeit von DSLs erlaubt automatische Erkennung domänenspezifischer Modellierungsfehler

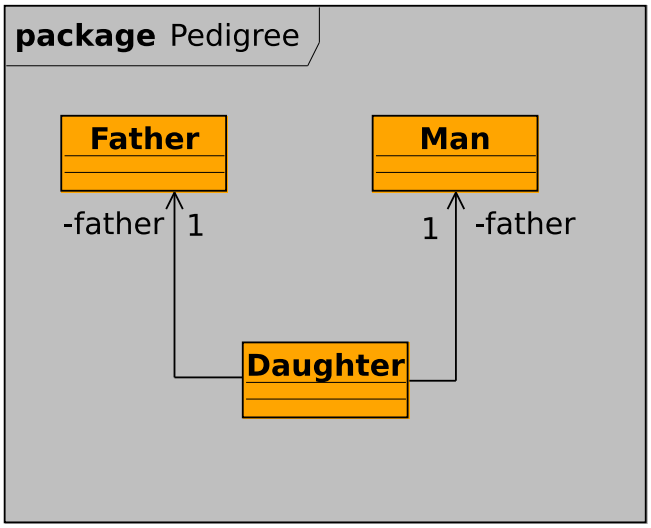


Abbildung II.10. UML-Modellierung als Klassendiagramm ermöglicht keine automatische Erkennung von domänenspezifischen Modellierungsfehlern

Im UML-Modell in Abbildung II.10 ist eine Erkennung dieses Fehlers nicht möglich, da hier ganz allgemein Klassen und Assoziation zwischen Klassen zur Modellierung des Sachverhaltes genutzt wurden. Der Fehler im Modell erschließt sich dem Modellierer hier nur durch die intuitive Interpretation der Namen von Klassen und Rollen und kann nicht automatisch erkannt werden. Modelle domänenspezifischer Modellierungssprachen haben zwei wesentliche Vorteile für Domänenexperten:

Effizientere Kommunikation zwischen Modellierern. Domänenexperten haben i. d. R. ihre eigene sehr spezialisierte Fachsprache. Innerhalb einer Domäne gibt es meist ein klares Verständnis zur Bedeutung der verwendeten Fachausdrücke. Oft ist aber die Abbildung domänenspezifischer Sachverhalte durch Modelle in universellen Modellierungssprachen nicht präzise und gleichzeitig intuitiv verständlich genug möglich. Grund ist der hohe Abstraktionsgrad der Sprachelemente in universellen Modellierungssprachen wie der UML. Durch das Ausdrücken domänenspezifischer Fachterme in der syntaktischen Sprachspezifikation können sehr intuitiv (im Domänenkontext) erlernbare Sprachen erstellt werden, auf deren Basis mit vergleichsweise wenigen Modellelementen semantisch sehr ausdrucksstarke Modelle entworfen werden können.

Effektivere automatische Syntaxvalidierung von Modellen. Das Abstraktionsniveau universeller Modellierungssprachen wie der UML ist sehr hoch. Auf der Spezifikationsebene dieser Sprachen lassen sich deswegen spezifische Zusammenhänge zwischen domänenspezifischen Entitätstypen nicht ausdrücken. Dies steht im Gegensatz zu den Ausdrucksmöglichkeiten von DSLs. Die Erfassung von domänenspezifischen Beziehungen zwischen Entitäten in einer DSL-Sprachspezifikation ermöglicht eine sehr detaillierte statische Analyse von domänenspezifischen Modellen auf Ebene der syntaktischen Korrektheit (incl. Bedingungen der sog. statischen Semantik). Im Vergleich zur Validierung von Modellen universeller Modellierungssprachen ermöglicht diese Analyse die Erkennung von Modellierungsfehlern, die auf Widersprüche zum Domänenwissen zurückzuführen sind.

Die Verbreitung des Einsatzes domänenspezifischer Modellierungssprachen im Software Engineering steht aktuell erst am Anfang. Viele domänenspezifische Modellierungssprachen sind erst in Kooperationsprojekten zwischen Wissenschaft und Industrie in der Entwicklung und Praxiserprobung (Luoma, Kelly und Tolvanen 2004). Allerdings werden zahlreiche domänenspezifische Architekturbeschreibungssprachen im Systems-Engineering einiger sicherheitskritischer Domänen wie Luftfahrt oder Bahntechnik eingesetzt und haben sich teilweise etabliert.

Der kurze Überblick und die Beispiele in diesem Abschnitt zeigen, dass sich Softwareingenieure vielfältigen Modellierungssprachen bedienen – seien sie nun standardisiert oder domänenspezifisch. Die effektive Modellierungsarbeit setzt dabei eine möglichst umfangreiche Werkzeugunterstützung voraus, u. a. zum Erstellen von Modellen, zum Synchronisieren paralleler bearbeiteter Modellversionen oder zur automatischen Validierung hinsichtlich Modellierungsfehlern. Dazu müssen Modellinformationen automatisch verarbeitet werden. Grundlage dafür ist eine formale und maschinenlesbare Spezifikation des Syntaxgraphen eines Modells (und möglichst umfangreicher kontextsensitiver Bedingungen der statischen Semantik) der zu untersuchenden Modellierungssprache. Der aktuelle Stand der Forschung in diesem Gebiet wird in den beiden nächsten Abschnitten genauer beleuchtet.

3.2 Automatische Validierung der Korrektheit von Software durch statische Analyse ihrer formalen Syntax

Modelle dienen in der Informatik sowohl der Problemformulierung (Spezifikation) als auch der Lösungsformulierung (Ablaufbeschreibung/Programm). Zur Prüfung der Korrektheit von Software (Modelle und Programmcode) müssen in der Regel zwei Fragen beantwortet werden (Balzert 1997, bd. 2): i) Wurde die Software richtig entwickelt? und ii) Wurde die richtige Software entwickelt?

3.2.1 Verifikation versus Validierung

Zur Beantwortung der Frage i) wird bewertet, ob ein Programm eine vorgegebene Spezifikation erfüllt. Dies kann manuell oder mit Mitteln der formalen *Verifikation* (zum Teil) automatisiert werden. Die formale Verifikation (Prüfung einer Software gegen deren Spezifikation) spielt für die nachfolgenden Betrachtungen der Dissertation keine Rolle.

Um die Frage ii) zu beantworten, muss beurteilt werden, ob die entwickelte Software die Funktionalität in dem Maße erbringt, wie es sich die Anforderungsträger letztendlich vorstellen. Die dafür notwendige Prüfung wird mit *Validierung* bezeichnet. Dabei unterscheidet man die Validierung der Softwaresyntax von der Validierung des Laufzeitverhaltens bei Ausführung auf der Zielplattform.

Zum Zeitpunkt der Modellierung eines Softwaresystems in der Analyse- und Entwurfsphase kann das Laufzeitverhalten noch nicht validiert werden, da i. d. R. vor der Realisierung keine Ausführung auf der Zielplattform möglich ist. Im Gegensatz dazu ist die Validierung der Syntax von Software durchaus schon in der Modellierungsphase möglich.

3.2.2 Statische Analysen zur Validierung von Programmen und Softwaremodellen

Die Validierung der Softwaresyntax wird auch als statische Analyse bezeichnet. Sie wird im Bereich der Programmierung durch den Compiler der jeweiligen Programmiersprache automatisiert.

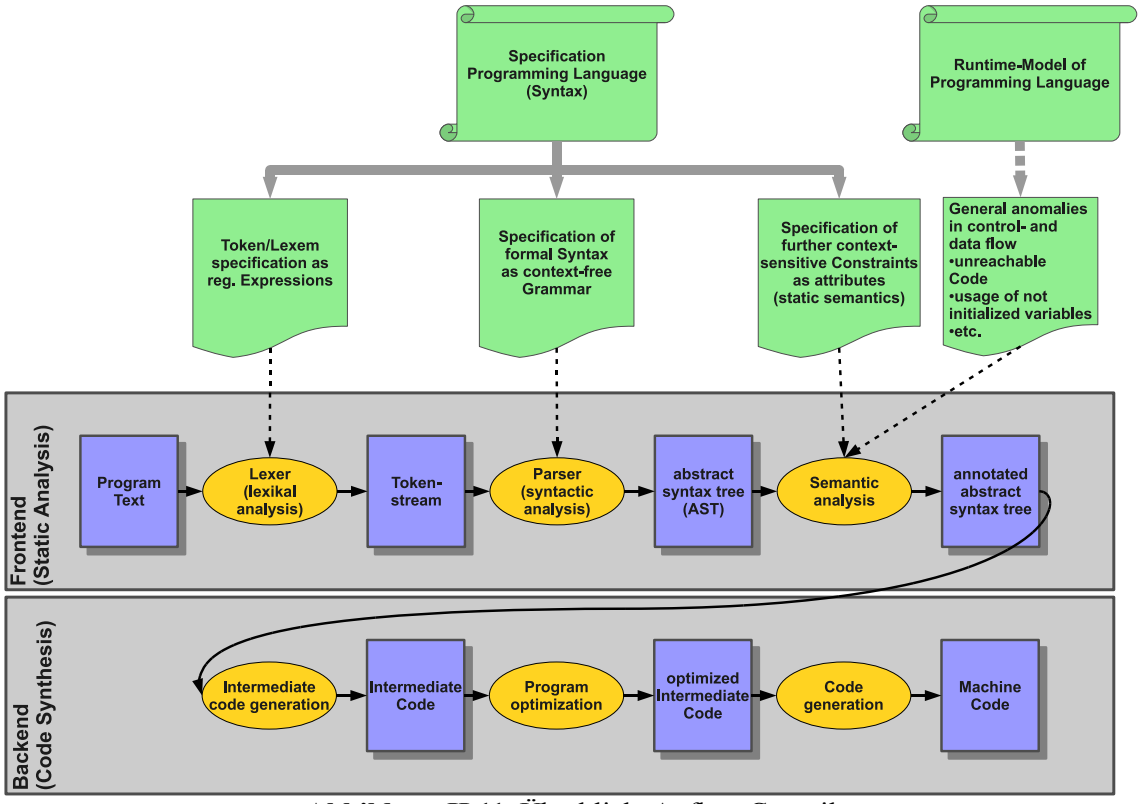


Abbildung II.11. Überblick: Aufbau Compiler

Dabei überprüft das Frontend des Compilers in der Analysephase, ob der durch den Programmierer entwickelte Quellcode der syntaktischen Spezifikation der Programmiersprache entspricht. Außerdem werden weitere Plausibilitätsprüfungen (bspw. in der Daten- und Kontrollflussanalyse) durchgeführt. Abbildung II.11 stellt die Phase der statischen Analyse von Programmen durch den Compiler dar und benennt die Teile der Spezifikation einer Programmiersprache, die für den jeweiligen Analyseschritt gebraucht werden.

Obwohl die statische Analyse ausschließlich anhand der Syntax von Programmen vorgenommen wird (→ keine Programmausführung), werden die Abschnitte der Analyse oft mit lexikalischer, syntaktischer und semantischer Analyse bezeichnet.

Zur syntaktischen Analyse gehört die Prüfung, ob ein Programm ein Wort einer kontextfreien Grammatik ist. Die Prüfung weiterer kontextsensitiver Bedingungen (ausgedrückt als Attribute einer kontextfreien Grammatik) wird oft mit semantischer Analyse bezeichnet. Insbesondere ist zu beachten, dass die Phase der semantischen Analyse (Prüfung der statischen Semantik) nicht mit der formalen Verifikation (Prüfung einer Spezifikation gegen ein Programm) verwechselt wird.

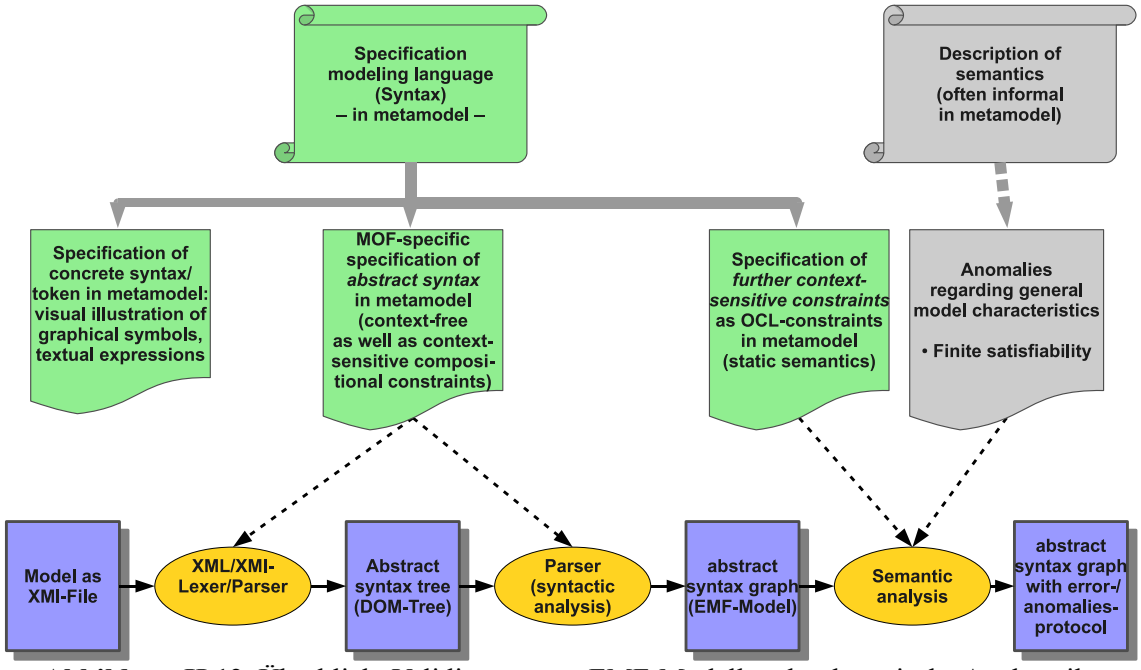


Abbildung II.12. Überblick: Validierung von EMF-Modellen durch statische Analyse ihrer Syntax

Analog zu Programmcode können Modelle im Rahmen einer statischen Analyse validiert werden. Allerdings werden hier die Begriffe syntaktische bzw. semantische Analyse oft nicht in Verbindung mit kontextfreier bzw. kontextsensitiver Syntaxspezifikation gebracht. Stattdessen wird mit syntaktischer Analyse oft die Prüfung eines Modells gegen die MOF/XSD-spezifische Syntaxspezifikation innerhalb des Metamodells bezeichnet (vgl. Abbildung II.12). Unter semantischer Analyse werden dann weitere Prüfungen kontextsensitiver Nebenbedingungen des Metamodells (oft als OCL-Constraints ausgedrückt) zusammengefasst.

Unabhängig von der Unterscheidung Begriffe zur syntaktischen und semantischen Korrektheitsprüfung handelt es sich um eine statische Analyse der Modellsyntax zur Validierung. Der Begriff semantische Korrektheitsanalyse darf demzufolge nicht verwechselt werden mit der formalen Verifikation (Prüfung gegen eine Spezifikation).

Nachdem nun das Vorgehen bei der statischen Analyse von Software (Programme und Modelle) im Rahmen der Validierung genauer betrachtet wurde, konzentriert sich der nächste Abschnitt auf die vorherrschenden Alternativen zur Spezifikation von Modellierungssprachen im Software Engineering.

3.3 Spezifikation von Modellierungssprachen im Software Engineering

Die Modellsyntax ist die durch Menschen bzw. Maschinen erfassbare Repräsentation der in einem Modell enthaltenen Informationen. Neben der Syntax muss die Bedeutung der Sprachmittel – die sog. Semantik – einheitlich festgelegt werden. Erst dadurch wird die

korrekte Interpretation der repräsentierten Informationen bei allen, die mithilfe von Softwaremodellen kommunizieren, ermöglicht.

Um eine angemessene Kommunikation zwischen Softwareingenieuren zu unterstützen, genügt i. d. R. der Einsatz von möglichst genau, aber informell beschriebenen Modellierungssprachen. Für eine maschinelle Verarbeitung wie der statischen Korrektheitsanalyse der Modellsyntax (sog. Modellvalidierung) bedarf es allerdings einer formalen, maschinenlesbaren Spezifikation. Diese lässt sich i. A. untergliedern in die folgenden vier Aspekte (vgl. (Clark, Evans und Kent 2002; Karsai u. a. 2003) und Abbildung II.12:

i. Beschreibung der konkreten Syntax (visuell/rezeptive Darstellung der Sprachmittel),

ii. Beschreibung der abstrakten Syntax (i. d. R. MOF-spezifische Syntaxspez. → Syntaxgraph) und der statischen Semantik (i. d. R. OCL-Constraints) meist durch ein Metamodell,

iii. Abbildung der konkreten auf die abstrakte Syntax (Zuordnung visueller und abstrakter Darstellung) und

iv. Abbildung der abstrakten Syntax auf eine semantische Domäne (Zuweisung der Bedeutung syntaktischer Sprachmittel).

Wie in den Punkten oben erwähnt, wird bei der Repräsentation von Modellen mittels grafischer Modellierungssprachen die abstrakte von der konkreten Syntax unterschieden. Die Festlegung der konkreten Syntax ist die Grundlage für die durch Menschen möglichst intuitiv interpretierbare, visuelle Darstellung eines Modells. Sie definiert beschriftete und unbeschriftete geometrische Formen wie Polygone und Linien, die Elementen der abstrakten Syntax entsprechen. Bspw. werden die Klassen der UML als Rechtecke, Anwendungsfälle als Ellipsen und Assoziationen als Linien dargestellt. Für die Modellvalidierung spielt die Art ihrer visuellen Darstellung eine untergeordnete Rolle und wird an dieser Stelle nicht weiter ausgeführt. Im Folgenden wird hingegen der Punkt ii. fokussiert.

In der modellbasierten Softwareentwicklung ist das Konzept der Metamodellierung die am häufigsten genutzte Alternative zur Beschreibung von Modellierungssprachen. Die formale Metamodellspezifikation von abstrakten Syntaxgraphen und statischer Semantik ist für die folgenden Kapitel von besonderer Bedeutung und wird anschließend im Detail betrachtet. Sie bildet die Grundlage für eine automatisierte Validierung (statische Analyse) von integrierten Modellversionen.

Um die Korrektheit eines Modells hinsichtlich der Syntaxspezifikation eines bestimmten Metamodells nachzuweisen, müssen Modell und Metamodell Maschinen-verarbeitbar repräsentiert sein. Ist diese Voraussetzung erfüllt, können, wie in Abbildung II.13 skizziert, automatisch Prüfwerkzeuge die Korrektheit eines Modells hinsichtlich eines Metamodells nachweisen bzw. entsprechende Fehler erkennen.

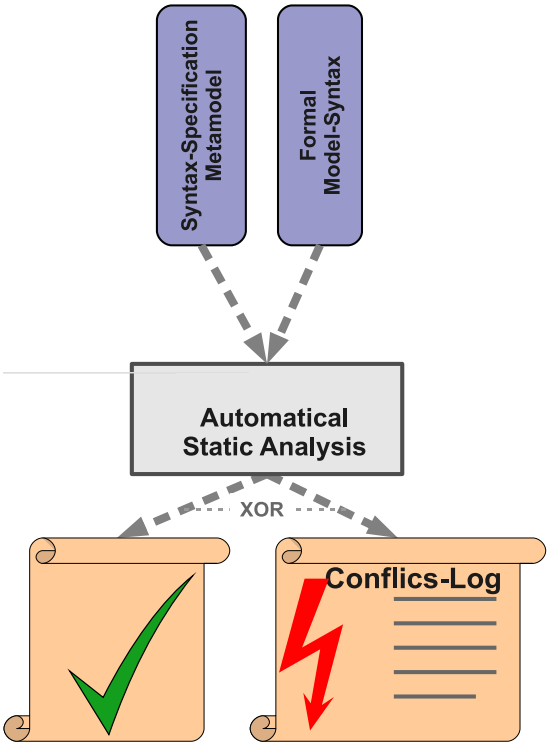


Abbildung II.13. Skizze - Automatische Validierung der Modellsyntax

Zur maschinellen Auswertung einer Sprachspezifikation muss das Metamodell einer Modellierungssprache wiederum formal und maschinenlesbar spezifiziert sein. Für diesen Zweck kommen wiederum Metamodelle zur Spezifikation von Metamodellen zum Einsatz. Diese werden als Meta-Metamodelle bezeichnet und sind zur maschinellen Verarbeitung von Modellierungssprachspezifikationen von entscheidender Bedeutung.

Im Forschungsfeld der modellbasierten Softwareentwicklung hat sich die *Meta-Object-Facility* (Abk. *MOF*) (Object Management Group (OMG) 2010b) weitestgehend zur Spezifikation von Metamodellen (bzgl. abstrakter Syntaxgraphen von Modellen) durchgesetzt. Daneben existieren weitere Standards wie die *XML-Schema-Definition* (Abk. XSD) (W3C 2010), *Relax-NG (OASIS 2001)* und *Schematron* (Schematron Comitee 2006).

Auf Basis der MOF spezifizierte Metamodelle werden oft mit zusätzlichen Bedingungen (sog. *OCL-Constraints*) angereichert, die durch die *Object Constraint Language* (*OCL*) (OMG 2010b) ausgedrückt werden. Im Forschungsgebiet der modellbasierten Softwareentwicklung wird der auf Basis der MOF spezifizierte Teil eines Metamodells zur Erstellung/Analyse des abstrakten Syntaxgraphen eines Modells genutzt und die Menge der zusätzlichen OCL-Bedingungen zur Prüfung der sog. statischen Semantik.

3.3.1 Statische Analyse abstrakter Syntaxgraphen von Modellen auf Basis von MOF-Metamodellspezifikationen

Die MOF hat sich seit 2003 als Standard zur Spezifikation von Modellierungssprachen im Software Engineering weitgehend durchgesetzt. Ihre Spezifikation wird durch die Object Management Group (OMG) gepflegt, weiterentwickelt und in fortlaufenden Releases

veröffentlicht. Sie beschreibt eine Schichtenarchitektur und ein Metamodell, das als Grundlage zur Syntaxspezifikation von Modellierungssprachen dient.

Neben der semi-formalen Metamodellbeschreibung enthält das Spezifikationsdokument zahlreiche natürlichsprachliche Erklärungen und Kommentare, in denen die Semantik der eingeführten Entitäten informell erläutert wird. Neben der Anwendung zur Spezifikation von domänenspezifischen Modellierungssprachen wurden zahlreiche standardisierte Modellierungssprachen wie die UML oder das *Software Process Engineering Metamodel* (Abk. *SPEM*) (OMG 2008a) auf Grundlage der MOF spezifiziert. Die Schichtenarchitektur der MOF und ihr Metamodell sind zentrale Grundlagen für den Inhalt der folgenden Kapitel und werden in den beiden folgenden Abschnitten im genauer erläutert:

3.3.2 Schichten-Architektur der MOF

Nach der Schichtenarchitektur der MOF werden Modellierungsebenen unterschieden, zwischen denen eine Typ-Instanz-Beziehung besteht. Basierend darauf werden in der modellbasierten Softwareentwicklung i. A. die vier in Abbildung II.14 illustrierten Modellierungsebenen unterschieden.

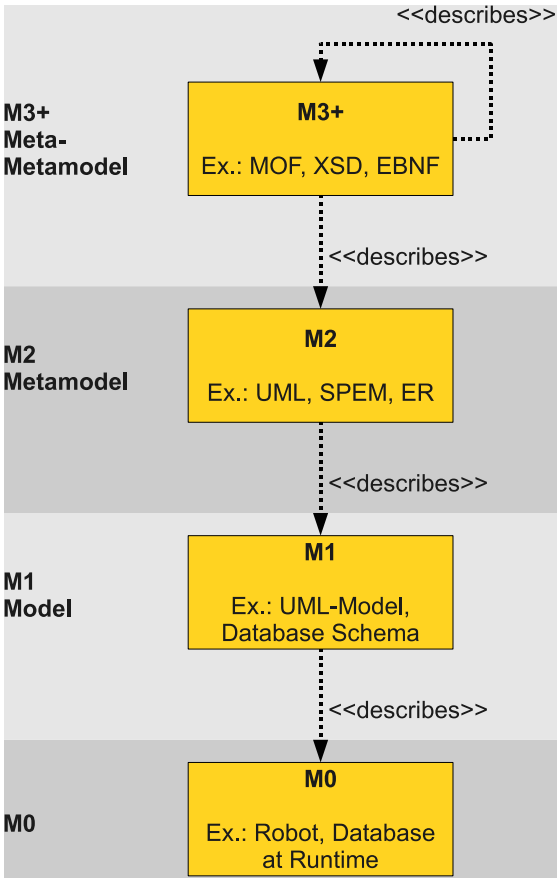


Abbildung II.14. Hierarchische Schichtenarchitektur der MOF

Dabei korrespondieren konkrete Objekte in der realen Welt mit der Ebene M0. Modelle, die die Struktur und das Laufzeitverhalten von Systemen beschreiben, werden der Ebene M1 zugeordnet. Metamodelle, die die abstrakte Syntax und ggf. die statische Semantik von Modellierungssprachen spezifizieren, gehören der Ebene M2 an. Meta-Metamodelle der Ebene M3 bieten schließlich, domänenunabhängig, universelle syntaktische Sprachmittel zur Spezifikation von Modellierungssprachen an. So wird bspw. das UML-Metamodell (Ebene M2) als eine Instanz der des MOF-Metamodells (Ebene M3) spezifiziert.

Um zu überprüfen, ob ein Modell (Ebene M1) bzgl. seiner Sprachspezifikation korrekt ist, muss analysiert werden, ob der abstrakte Syntaxgraph des Modells auf M1 zu der Menge der Graphen gehört, die das Metamodell der Ebene M2 spezifiziert.

Soll eine solche Syntaxüberprüfung automatisch durchgeführt werden, so muss die Syntaxspezifikation (Metamodell der Ebene M2) formal und maschinenlesbar zur Verfügung stehen. Um dies zu gewährleisten, wird ein Meta-Metamodell wie das der MOF auf der Ebene M3 spezifiziert, das formal angibt, wie Metamodelle von Modellierungssprachen repräsentiert sein müssen. Die Ebenen-Abhängigkeiten ließen sich so auf die gleiche Weise mit zusätzlichen Ebenen fortschreiben; allerdings sind die vier Ebenen ausreichend um Modelle und Metamodelle von Modellierungssprachen präzise zu beschreiben und maschinell zu verarbeiten. Das MOF-Meta-Metamodell dient dabei rekursiv als sein eigenes Metamodell.

Die Syntax des MOF-Meta-Metamodells basiert auf einem Ausschnitt der UML. Neben den Strukturbeschreibungen, die durch den abstrakten MOF-Syntaxgraphen im Metamodell beschrieben wird, werden zusätzliche OCL-Bedingungen angegeben, um die Syntaxspezifikation weiter durch Korrektheitsbedingungen der sog. statische Semantik zu beschreiben.

Damit Sprachspezifikationen von Modellierungssprachen maschinell eingelesen und verarbeitet werden können, wurde das MOF-Metamodell in mehreren Frameworks implementiert. Die meisten Implementierungen realisieren allerdings nur einen Kernausschnitt des MOF-Metamodells, der sich weitestgehend mit dem Paket *Essential MOF* (Abk. *EMOF)* in der MOF-Spezifikation deckt.

Implementierungen der MOF

Das wohl inzwischen am weitesten verbreitete Framework zur Implementierung der MOF ist das *Eclipse Modeling Framework* (*EMF*) (Eclipse Foundation 2010). Da die prototypische Implementierung des in dieser Dissertation beschriebenen Lösungskonzeptes ebenfalls auf EMF basiert, wird i. F. auf dieses Framework Bezug genommen. Neben der durch IBM vorangetriebenen MOF-Implementierung EMF startete Sun Micro Systems mit dem Framework *Meta Data Repository (MDR)* (Sun Microsystems 2010) eine ähnliche Entwicklung. Im Umfeld des Forschungsgebietes der Graphersetzungssysteme entstammt das auf der Fujaba Toolsuite (Fujaba development group 2010) basierende Metamodellierugs-Framework *MOFLON* (moflon.org 2010) mit ebenfalls verwandter Zielsetzung.

Eclipse Modeling Framework (EMF)

Das *Eclipse Modeling Framework (EMF)* wurde im Rahmen des Eclipse-Projektes federführend von IBM entwickelt. Es ist ein Framework zur Unterstützung der werkzeuggestützten Software- und Systemmodellierung auf Basis des sog. EMF-Datenmodells, das die Verarbeitung von (Meta-)Modellen ermöglicht. Außerdem bietet es mit XMI (Object

Management Group (OMG) 2010a) einen Standard zur persistenten Speicherung und zum Austausch von Modellen und Metamodellen zwischen Softwarewerkzeugen an.

Innerhalb des EMF-Frameworks ist die Meta-Metamodellierungssprache ECore im Datenmodell verankert. ECore ist eine Sprache zur Beschreibung von Metamodellen auf Basis der MOF-Spezifikation. Dabei deckt ECore im Wesentlichen die Beschreibungsmittel der EMOF ab. Abbildung II.15 zeigt das Meta-Metamodell der Sprache ECore in der von der UML bekannten Notation als Klassendiagramm.

Das EMF-Framework legt für die Sprache ECore eine Semantik fest. Dafür stellt es Modell-zu-Text-Transformationen bereit, die auf Basis von ECore spezifizierte Metamodelle in Java-Code transformieren. Mit dem generierten Java-Code lassen sich anschließend durch EMF-basierte Softwarewerkzeuge beliebige Modelle des entsprechenden Metamodells laden und weiter verarbeiten.

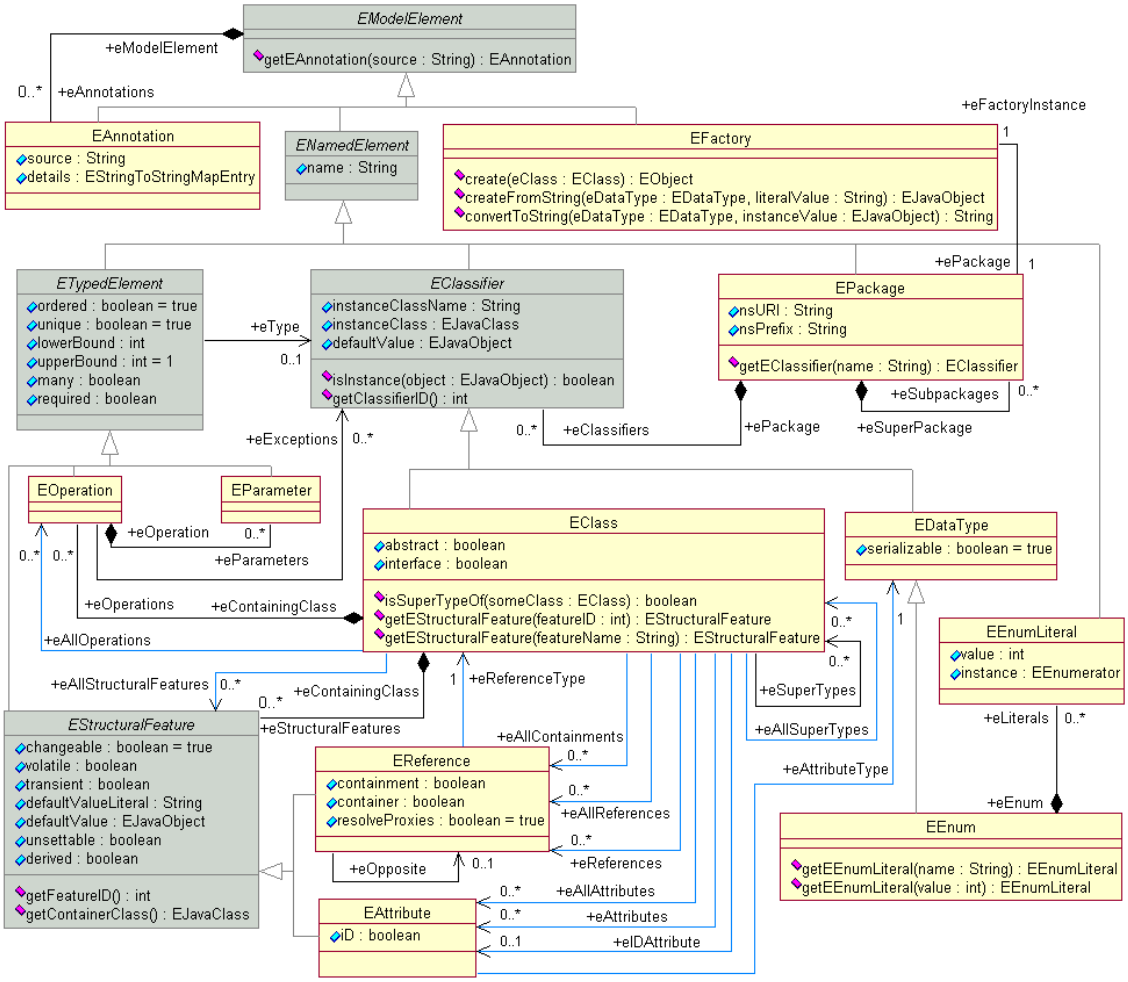


Abbildung II.15: ECore-Metamodell

Für grundlegende Modellverarbeitungen stellt EMF teils als Referenzimplementierung schon eine Werkzeugunterstützung bereit. Diese unterstützt das Erstellen, Ändern und Erweitern von Modellen mittels Modelleditoren, die auf Basis des Metamodells generiert werden. Außerdem bietet das Framework Werkzeugunterstützung bei für die Modellvalidierung u. v. m.

Weitere Eclipse-Projekte wie zur Entwicklung des *Graphical Editing Framework (GEF)* und des *Graphical Modeling Framework (GMF)* (Eclipse Foundation 2010) bauen auf der durch EMF gegebenen abstrakten Modellsyntax auf und widmen sich der Unterstützung von Spezifikationen konkreter, grafischer Modellsyntaxen für domänenspezifische Sprachen bzw. entsprechenden Abbildungen zwischen konkreter und abstrakter Modellsyntax. Eclipse-Projekte zur Entwicklung der *Atlas Transformation Language (ATL)* beschäftigen sich mit der Implementierung einer Modelltransformationssprache für EMF-Modelle auf Basis des *Query View Tranformation* (Abk. *QVT*) -Standards (OMG 2008b) insbesondere im Rahmen der *Model Driven Architecture* (Abk. *MDA*) (OMG 2010a). Aufgrund dieser Voraussetzungen eignet sich das EMF-Framework gut als technische Plattform für die Erstellung einer Modellierungsumgebung für Sprachdesign und domänenspezifische Modellierung.

Neben der Beschreibung der Syntax einer Modellierungssprachen sollen bei deren Spezifikation oft weitere Bedingungen zur Beschreibung der sog. statischen Semantik formuliert werden. Für diesen Zweck wird vielfach die Object Constraint Language (OCL) (OMG 2010b) ergänzend eingesetzt.

3.3.3 Validierung der statischen Semantik von Modellen durch OCL-Constraints im Metamodell

Im Jahr 1995 wurde die *Object Constraint Language* (*OCL*) (OMG 2010b) im Rahmen eines Projektes bei der IBM entwickelt. Populärer wurde sie durch den von der OMG empfohlenen Einsatz im Rahmen des Entwurfes von UML-Modellen. Dabei wird die OCL hauptsächlich verwendet, um als logische Sprache insbesondere Invarianten und Vor- und Nachbedingungen von Methoden in objektorientierten Strukturen durch formale textuelle Ausdrücke beschreiben zu können.

Die auf Basis Der MOF spezifizierte Syntax von Modellen kann so um weitere (kontextsensitive) Bedingungen zur statischen Semantik angereichert werden, die auf Basis der OCL möglichst präzise und standardisiert ausgedrückt werden können. Eine formale Semantik für OCL ist allerdings durch die OMG nicht spezifiziert, allerdings gibt es zahlreiche Vorschläge und Implementierungen der OCL, durch die die intuitive Semantik der informellen Spezifikation präzise festgelegt wird (Demuth 2004).

Zur Spezifikation der MOF und der UML wurden zahlreiche OCL-Bedingungen in die jeweiligen Metamodelle aufgenommen, um die statische Semantik der Modellierungssprachen möglichst formal zu beschreiben.

III Synchronisierung kollaborativ entwickelter Softwaremodelle

Automatische Versionsverwaltungssysteme (engl. *versions control system - VCS*) wie CVS, Subversion oder Bitkeeper bilden heute das Herzstück bei der Synchronisierung von Teamarbeit im Bereich der Programmierung. Mit ihnen lassen sich selbst riesige Quellcodebasen, an denen hunderte Entwickler asynchron/parallel arbeiten, ohne große Probleme verwalten.

In der Anforderungsanalyse und im Entwurf von Software arbeiten meist weniger Ingenieure kollaborativ zusammen. Dennoch ist ihre Arbeitsweise oft ebenfalls asynchron und stark verzahnt, außerdem sind die Teams groß genug, sodass auch hier eine möglichst effiziente Synchronisierung der Arbeitsergebnisse eine wichtige Rolle spielt.

Leider lässt sich der Erfolg in der Verwaltung von Quellcode nicht ohne Weiteres auf die Arbeitsweise der Analyse- und Entwurfsphase übertragen. In diesen Phasen werden verstärkt (semi-)formale Softwaremodelle zur Spezifikation, Codegenerierung und Dokumentation eingesetzt.

Insbesondere die Art der Modelldatenrepräsentation unterscheidet sich stark von Quellcode textueller Programmiersprachen. Dieser Unterschied ist die Hauptursache für das Scheitern aktueller Versionsverwaltungssysteme an einer praktikablen softwaregestützten Arbeitssynchronisierung in der modellbasierten Softwareentwicklung.

Diese Erkenntnis soll durch die folgenden Abschnitte verdeutlicht werden. Ziel des Kapitels ist es, die Konsequenzen für die Konzeption einer leistungsfähigen Versionsverwaltung für Softwaremodelle zu erörtern und damit die inhaltliche Gliederung von Kapitel IV zu skizzieren.

Zunächst wird dazu detailliert auf die Versionsverwaltung für Quellcode eingegangen. Dabei stehen die Aspekte im Fokus, die in hohem Maße die Synchronisierung der Entwicklungsergebnisse parallel und asynchron arbeitender Teams betreffen. Der Vergleich mit der Quellcodeverwaltung ist hilfreich, um ein besseres Verständnis der Herausforderungen bei der Versionsverwaltung von Modellen zu erzeugen.

Im Abschnitt 2 wird das Merge-Verfahren zur Integration parallel erstellter Quellcode-Versionen aus Anwendersicht im Detail analysiert. Anschließend wird ein Modellierungsbeispiel vorgestellt, das ein domänenspezifisches Metamodell auf Basis der MOF für Anforderungsanalyse und Architekturentwurf einführt. Dieses Beispiel dient nachfolgend der Erläuterung von Herausforderungen und Lösungsansätzen.

Die einzelnen Phasen dieses Verfahrens aus Abschnitt 2 werden in Abschnitt 4 auf die kollaborative modellbasierte Softwareentwicklung übertragen. Dabei werden die Schwächen aktueller Merge-Konzepte bei der Verarbeitung von Softwaremodellen herausgestellt und die Herausforderungen in der Konzeption eines Merge-Verfahrens für Softwaremodelle erläutert.

1 Synchronisierung von Teamarbeit in der Quellcodeverwaltung – eine Erfolgsgeschichte

Bevor modellbasierte Entwicklungsansätze stärkere Verbreitung gefunden haben, waren Quellcode und natürlichsprachliche Textdokumente (bspw. Anforderungen, Spezifikationen) die einzigen Daten, die bei der Softwareentwicklung zur gemeinschaftlichen Bearbeitung zur Verfügung gestellt werden mussten. Technische Unterstützung benötigte die kollaborative Arbeit hauptsächlich in der Koordination des Entwicklerteams.

Erste Versionsverwaltungswerkzeuge wie SCCS (Rochkind 1975) oder RCS (Tichy 1985) erlaubten die Protokollierung von Änderungen und die Archivierung sowie den lokalen Zugriff auf einzelne Dateiversionen. Diese Werkzeuge wurden sukzessive erweitert, um die Datei- und Quellcodeverwaltung ganzer Softwareprojekte zu unterstützen.

Für die Zusammenarbeit eines Entwicklerteams ist dabei die Koordinierung der Bearbeitungszugriffe auf einzelne Dokumente von zentraler Bedeutung. Oft müssen mehrere Entwickler an demselben Dokument arbeiten. Eines der Hauptprobleme dabei ist das Einpflegen der Änderungen verschiedener Entwickler in eine gemeinsame Version. Zur Lösung dieses Koordinationsproblems konnten sich in der Vergangenheit zwei verschiedene Strategien durchsetzen:

- *Pessimistische Versionsverwaltung*
- *Optimistische Versionsverwaltung*

Beide Strategien unterscheiden sich hinsichtlich ihres Ansatzes zur Arbeitssynchronisation. Die Auswirkungen auf die Teamarbeit sowie ihre entsprechenden Vor- und Nachteile werden in den beiden folgenden Abschnitten deutlich gemacht:

1.1 Teamkoordination mittels pessimistischer Versionsverwaltung

Nach dem Prinzip der pessimistischen Versionsverwaltung erfolgt die Synchronisierung von Dokumentbearbeitungen auf Basis von Sperrmechanismen (engl. *locking mechanism*). Diese verhindern ein gleichzeitiges Ändern derselben Version durch mehrere Bearbeiter. Alle Änderungen an einer Datei müssen durch die Bearbeiter zeitlich nacheinander eingepflegt werden (sog. *Lock-Modify-Write*-Strategie vgl. Abschnitt II.2.2.1). Diese Einschränkung hat in drei Bereichen wesentliche Konsequenzen:

- Koordinierung bei der Bearbeitung der teamweiten Quellcodebasis.
  Sollen mehrere Bearbeiter einzelne Versionen von Quellcodedateien weiterentwickeln, indem ihnen in bestimmten Zeitfenstern ein exklusives Bearbeitungsrecht zugestanden wird, so müssen sie sich u. U. aufwendig abstimmen. Sowohl die Reihenfolge als auch die Zeitspanne personenbezogener Bearbeitungen müssen im Team vereinbart werden, um ein Mindestmaß an Planbarkeit der Arbeitsaufwände zu garantieren. Durch die starken zeitlichen Abhängigkeiten in der Arbeitsplanung wirken sich Zeitverzögerungen in den Bearbeitungsaktivitäten Einzelner auf die zeitliche Planung des gesamten Teams aus. Dadurch kommt es u. U. zu vermeidbaren Zeitverzögerungen (Prudêncio, Murta und Werner 2009). Durch die Vermeidung von Abhängigkeiten zwischen Bearbeitungszeiträumen einzelner Entwickler können die benannten Ineffizienzen vermieden werden, wie im folgenden Abschnitt zur optimistischen Versionsverwaltung beschrieben werden wird. Dennoch waren bis zur Entwicklung von optimistischen Versionierungswerkzeugen pessimistische Verfahren in der Quellcodeverwaltung weit verbreitet.

- Werkzeugunterstützung
  Zu den bekanntesten pessimistischen Versionierungswerkzeugen gehört das Open-Source-Werkzeug *Revision Control System* (Abk. *RCS*) (Tichy 1985) und das proprietäre *Visual SourceSafe* (Abk. *VSS*) von Microsoft (Microsoft 2003). Pessimistische Versionsverwaltungsstrategien werden bis heute vorzugsweise für die Verwaltung von Binärdateien, XML-Dateien und anderen aktuell nicht optimistisch verwaltbaren Dateiformaten eingesetzt. Sie verwalten Versionen allein auf Dateiebene unabhängig von ihrer inhärenten Datenrepräsentation. Allerdings wurden in der Quellcodeverwaltung pessimistische Strategien weitgehend abgelöst (Prudêncio, Murta und Werner 2009).

1.2 Teamkoordination mittels optimistischer Versionsverwaltung

Zur Flexibilisierung der Kollaboration von Entwicklerteams wurden optimistische Verfahren zur Versionierung von Textdateien entwickelt. Schwerpunkt dieser Entwicklung war die Bereitstellung einer effizienteren Koordination bei der kollaborativen Programmierung.

Im Gegensatz zur pessimistischen Versionsverwaltung erlaubt die optimistische Verwaltung die asynchrone Weiterentwicklung von Versionen durch mehrere Bearbeiter. Eine zeitliche Abstimmung der Bearbeiter bzgl. des Schreibzugriffs auf Versionen ist i. d. R. nicht nötig. Wird eine Version parallel weiterentwickelt, so müssen die vorgenommenen Änderungen wieder zu einer Folgeversion integriert werden. Das dabei angewendete Prinzip *Copy-Modify-Merge* wurde in Abschnitt II.2.2.2 im Detail erläutert.

Die Strategie verbessert i. d. R. die Effizienz des Entwicklerteams im Vergleich zu pessimistischen Verfahren, da Koordinationsaufwände beim Bearbeitungszugriff auf gemeinsame Dokumente weitgehend entfallen. Allerdings bringt die optimistische Versionsverwaltung auch neue Herausforderung mit sich:

- Koordination bei der Bearbeitung der teamweiten Quellcodebasis.
  Explizites Ziel der optimistischen Versionsverwaltung ist die Vereinfachung der Koordination von Teams bei der gemeinsamen Bearbeitung einzelner Versionen. Das wird erreicht, indem Sperrmechanismen, wie sie aus der pessimistischen Versionsverwaltung bekannt sind, nicht angewendet werden, sondern den Bearbeitern maximale Freiheit in der Versionsbearbeitung eingeräumt wird. Eine aufwendige Koordination personenbezogener Bearbeitungszeiträume kann dadurch entfallen.

Allerdings erfordert das Verfahren eine inhaltliche Synchronisierung von parallel durchgeführten Änderungen am Quellcode. Die Ursache dafür liegt in der Merge-Phase des Copy-Modify-Merge-Prinzips. Um den manuellen Synchronisationsaufwand so gering wie möglich zu halten, automatisieren entsprechende Softwarewerkzeuge wesentliche Aufgaben der Merge-Phase.

Versionsverwaltungssysteme wie bspw. CVS (GNU Project 2010b) oder Subversion (Tigris.org 2010) pflegen parallel durchgeführte Änderungen an einer Textdatei mithilfe eines 3-Wege-Merge-Verfahrens zeilenweise ein (vgl. Abschnitt II.2.2.3). Dabei kann es zu Konflikten wegen widersprüchlich geänderter Codezeilen einerseits oder zu Verletzungen der Syntaxspezifikation (kontextfreie Struktur des Syntaxbaums oder kontextsensitive Nebenbedingungen) der Programmiersprache andererseits kommen.

Derartige Konflikte müssen i. d. R. manuell durch die Programmierer gelöst werden. Die Lokalisierung der Konflikte geschieht aber meist automatisiert durch einen Compiler oder Debugger. Einige Kollaborationsumgebungen für die Softwareentwicklung wie bspw. Jazz von IBM (IBM 2010b) betten die syntaktische und semantische Codeanalyse in die Konfliktanalyse nach einem Merge-Vorgang ein. Die automatischen Analysetechniken sollen die Programmierer in die Lage versetzen, möglichst effizient parallele Änderungen korrekt in eine gemeinsame Datenbasis zu integrieren.

- Werkzeugunterstützung zur Verwaltung verschiedener Datenformate.
  Alle etablierten optimistischen Versionierungswerkzeuge vergleichen während der Merge-Phase parallele Weiterentwicklungen mit ihrer gemeinsamen Ausgangsversion (vgl. zustandsbasiertes Merge-Verfahren, Abschnitt II.2.2.5). Dies dient der Identifizierung der Änderungen in den betreffenden Weiterentwicklungen.

  Alle populären Softwaresysteme in diesem Bereich spezialisieren sich auf die Verwaltung von kollaborativ zu entwickelnden Quellcode (engl. *Source Code Control System – SCCS*). Bei der Anwendung des 3-Wege-Merge-Verfahrens analysiert der angesprochene Vergleichsmechanismus Änderungen an Quellcodedateien auf der Ebene von Textzeilen.

  Dazu werden Textvergleichsverfahren (Abk. *Diff* für engl. *Differcence*) wie *Unix-diff* (Hunt und McIlroy 1976) verwendet. Leider haben optimistische Versionierungstechniken im Software Engineering außerhalb der Quellcodeverwaltung bisher kaum Anwendungsfelder gefunden. Keiner etablierten Versionierungssoftware gelingt es zurzeit, die Merge-Phase so zu implementieren, dass auch andere Software-Engineering-Produkte mit formaler Syntax wie bspw. Modelle in gleicher Weise effizient verwaltet werden können.

Der erfolgsentscheidende Faktor für die Anwendbarkeit optimistischer Versionierungsverfahren ist die Unterstützung einer praktikablen Merge-Phase. Sie wird deswegen im folgenden Abschnitt im Detail betrachtet.

2 Analyse des Merge-Verfahrens für Quellcode aus Anwendersicht

Merge-Verfahren im Rahmen der optimistischen Versionsverwaltung von Quellcode basieren auf dem 3-Wege-Merge-Prinzip (vgl. II.2.2.3) und dem darauf aufbauenden Text-Merge-Verfahren (vgl. II.2.2.7). Zusätzlich zur Integration der reinen Textänderungen durch das Text-Merge-Verfahren werden Programmierer bei der Codeanalyse durch Software unterstützt, um Merge-bedingte Fehler zu analysieren. Das werkzeuggestützte Merge-Verfahren für Quellcode lässt sich aus Anwendersicht in zwei Phasen einteilen:

i. Integration von Quellcodeänderungen mit einem Text-Merge-Verfahren

ii. Merge-bedingte, automatische Validierung integrierter Quellcodeversionen und anschießende manuelle Fehlerbehebung

Heutzutage sind Programmiersprachen i. d. R. textuelle Sprachen. Quellcode ist somit in Textdateien repräsentiert. Zum Zusammenführen paralleler Änderungen am Quellcode kommen Text-Merge-Verfahren zum Einsatz:

2.1 Integration von Quellcodeänderungen mit Text-Merge-Verfahren

Wie in Abschnitt II.2.2.7 beschrieben, besteht die logische Struktur des Text-Merge-Algorithmus aus den drei Phasen – Zerlegung von Ausgangsversion und zu integrierenden Versionen in eine Sequenz von Textzeilen, Vergleich dieser Sequenzen hinsichtlich Änderungen und Anwendung des 3-Wege-Merge-Prinzips beim Integrieren der Zeilenänderungen. Im Anschluss müssen ggf. Merge-Konflikte aufgrund inkonsistenter Änderungen behoben werden. In Abbildung II.7 und Abbildung II.8 wurde dieses Vorgehen dargestellt. Im Beispiel zeigt Abbildung II.8 das Ergebnis des Text-Merge-Algorithmus anhand eines Quellcodeausschnitts. In dem Beispiel treten keine weiteren Konflikte auf, sodass die dargestellte Ergebnisversion dem finalen Merge-Ergebnis entspricht.

Allerdings können während des Zusammenführens von Textversionen Konflikte aufgrund widersprüchlicher Änderungen an Textzeilen auftreten. In diesen Fällen müssen die Entwickler die entsprechenden Konflikte manuell lösen, bevor die zweite Phase des Merge-Vorgangs gestartet werden kann. Abbildung III.1 zeigt dazu ein Beispiel.

Die Abbildung zeigt, wie zwei Entwickler unabhängig voneinander in den Weiterentwicklungen *VA* und *VB* die Methode zur Berechnung des größten gemeinsamen Teilers (Abk. *ggt* oder engl. greatest common divisor – *gcd*) jeweils unterschiedlich in *ggT* und *gcd* umbenannt haben. Diese Änderungen sind inkonsistent zueinander. Wie die Methode *ggt* nach dem Einpflegen der Änderungen letztendlich benannt werden soll, kann nicht automatisch entschieden werden.

Um ein Merge-Ergebnis zu bestimmen, müssen die Entwickler deswegen den Konflikt manuell lösen. Die Erkennung und Darstellung derartiger Merge-Konflikte ist heute weitgehend automatisiert. Durch diese Softwareunterstützung wird die manuelle Konfliktbehebung erheblich vereinfacht.

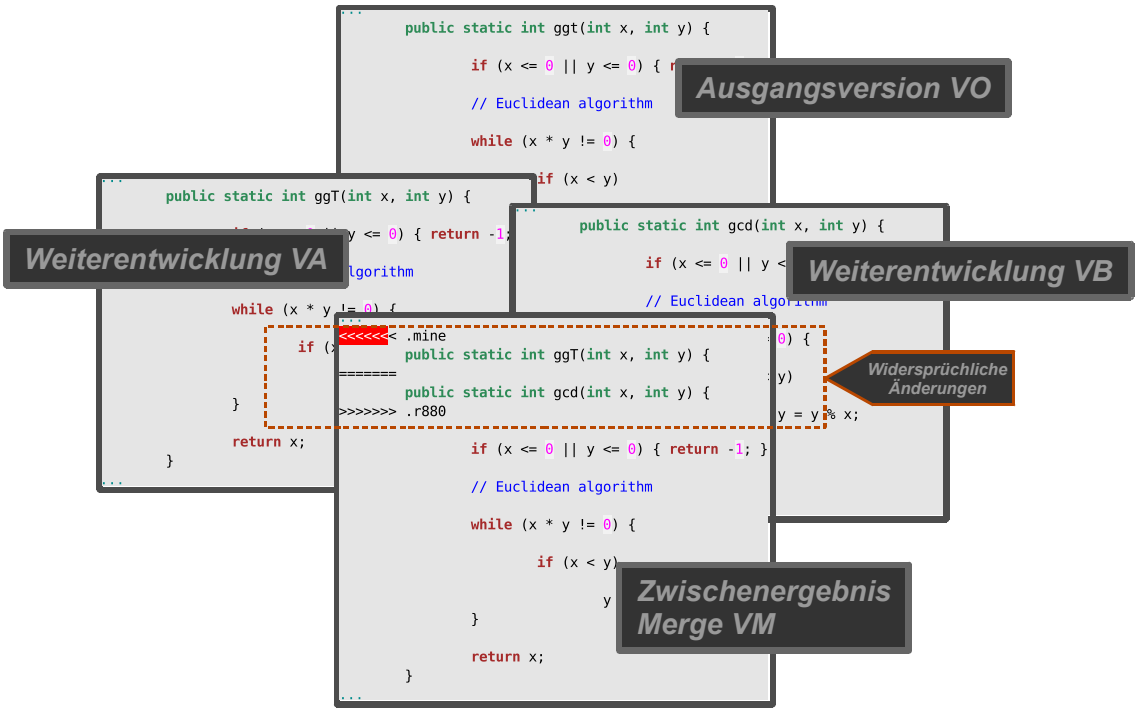


Abbildung III.1. Behandlung widersprüchlicher Änderungen bei der optimistischen Versionsverwaltung von Quellcode

Nachdem parallel erzeugte Quellcodeversionen durch das Text-Merge-Verfahren zusammengeführt wurden, wird i. d. R. im zweiten Schritt das Merge-Ergebnis einer statischen Codeanalyse im Rahmen der Kompilierung unterzogen.

2.2 Merge-bedingte Validierung in integrierten Quellcodeversionen

Was nicht kompiliert, darf nicht eingecheckt werden! Dies ist ein allgemein anerkannter Grundsatz bei der Programmierung, wenn es um die Synchronisierung der Teamarbeit mithilfe eines Versionsverwaltungssystems geht. Programmierfehler, die durch den Compiler automatisch erkannt werden, müssen demzufolge vor der Veröffentlichung einer Version durch den Verursacher behoben werden.

Hinter dem formulierten Grundsatz verbirgt sich der Anspruch, Fehler im Quellcode zu lokalisieren, deren Ursachen ausschließlich auf die Integration paralleler Änderungen durch das Text-Merge-Verfahren zurückzuführen sind. Denn es gilt:

Werden zwei fehlerfreie Quellcodeversionen mit einem Text-Merge-Verfahren zusammengeführt und enthält dessen Ergebnis Fehler, so lassen sich diese Fehler, auf eine gescheiterte automatische Integration paralleler Code-Änderungen zurückführen.

Merge-bedingte Verletzung der kontextfreien Syntaxspezifikation von Programmiersprachen

Die Lokalisierung von Merge-bedingten syntaktischen Fehlern lässt sich für Quellcode vollständig automatisieren. So zeigt Abbildung III.2 ein Beispiel, in dem zwei syntaktisch fehlerfreie Quellcodeversionen mit etablierten Text-Merge-Verfahren zusammengeführt werden. Im Merge-Ergebnis wird von Eclipse allerdings ein Syntaxfehler erkannt. Die entsprechende Fehlermeldung, wegen eines inkorrekten *if-else*-Konstrukts ist in Version *VM* dargestellt.

Fehler bzgl. der kontextfreien Syntaxspezifikation von Programmiersprachen, wie beispielhaft in Abbildung III.2 dargestellt, werden durch die Syntaxanalyse des Compilers erkannt und lokalisiert. Der Syntaxbaum, der durch die kontextfreie Grammatik einer Programmiersprache bestimmt wird, lässt sich beim Auftreten entsprechender Fehler nicht aufbauen. Sind alle syntaktischen Fehler im Programmcode bereinigt, so lässt sich dieser interpretieren, bzw. auf der jeweiligen Maschine ausführen.

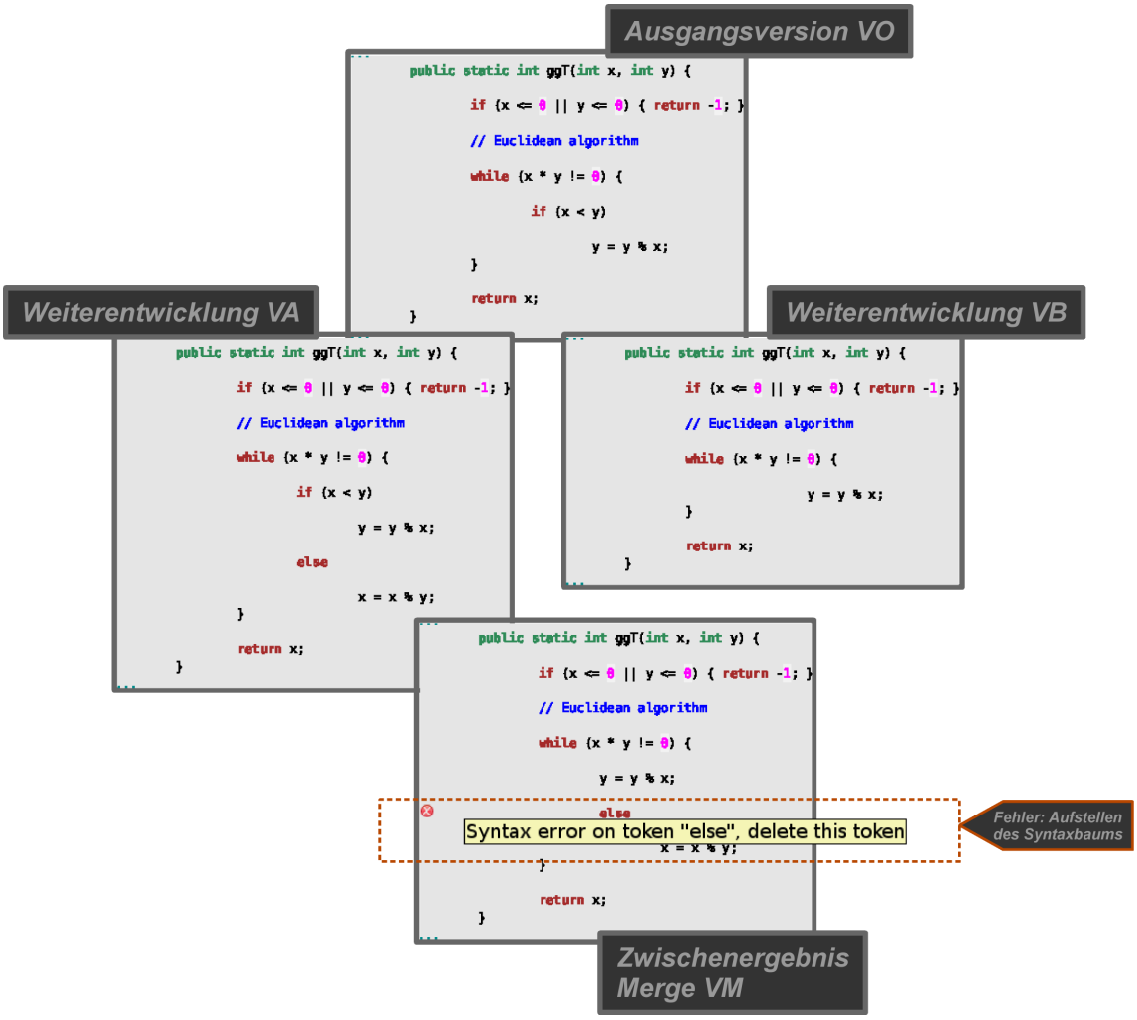


Abbildung III.2. Mergebedingter Syntaxfehler - Beispiel

Neben den diesen Fehlern beim Aufstellen des Syntaxbaums eines Programms können weitere Fehler durch eine weitergehende statische Analyse der Programmsyntax erkannt werden, die sich bei Ausführung eines Programms erst zur Laufzeit bemerkbar machen würden.

Diese werden i. F. im Fall von Programmcode als Fehler in der statischen Semantik bezeichnet. Automatische Verfahren, die solche Fehler erkennen können, nutzen häufig die Attributierung der kontextfreien Grammatik. Damit lassen sich weitere kontextsensitive Beziehungen zwischen syntaktischen Einheiten (sog. Token) ausdrücken bzw. prüfen:

Merge-bedingte Verletzung kontextsensitiver, syntaktischer Korrektheitsbedingungen (statische Semantik) von Programmiersprachen

In der semantischen Analyse von Programmcode wird die Spezifikation kontextsensitiver Bedingungen an die Syntax einer Programmiersprache (sog. statische Semantik) gegen die Programmsyntax geprüft. Bei Programmiersprachen ist diese i. d. R. durch Attribute einer kontextfreien Grammatik realisiert.

In Abbildung III.3 ist ein Beispielszenario zu sehen, bei dem durch die Integration paralleler Änderungen ein typischer Fehler in der statischen Semantik durch Anwendung des Text-Merge-Verfahrens entsteht. Dieser Fehler kann im Syntaxbaums des Java-Codes erkannt werden. Die Bedingung, die dazu erfüllt sein muss, ist: Vor der Initialisierung einer Variable muss diese deklariert worden sein.

Durch die Erkennung der in Version *VM* verletzten Bedingung wird verhindert, dass zur Laufzeit einer Variable ein Wert zugewiesen wird, für die noch kein Speicherplatz reserviert worden ist. Somit wird durch die sog. semantische Analyse des Quellcodes schon die Behebung bestimmter Fehler zur Compile-Zeit ermöglicht, die sonst erst zur Laufzeit erkannt werden würden.

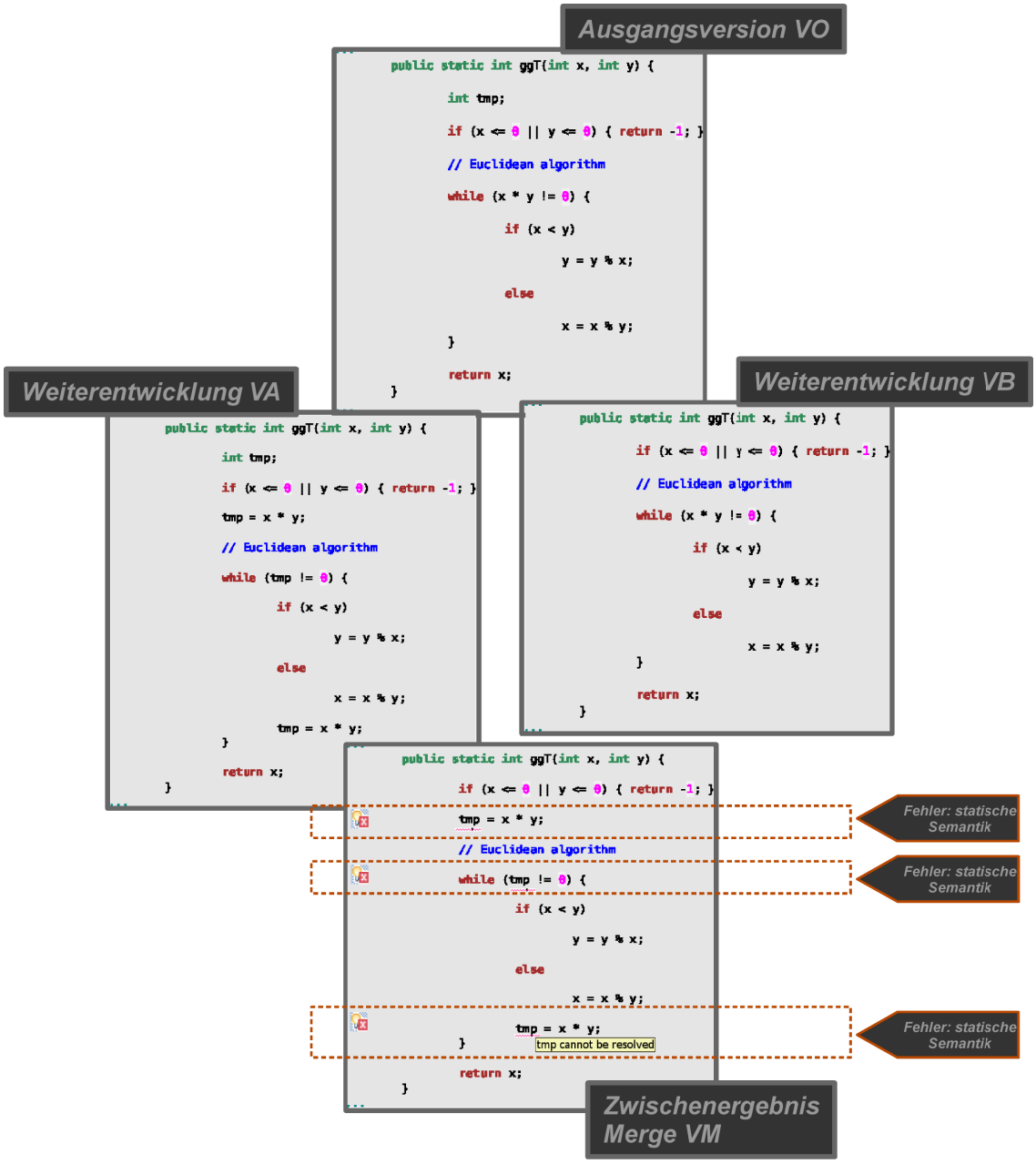


Abbildung III.3. Merge-bedingter Syntaxfehler bzgl. statischer Semantik - Beispiel

Merge-bedingte semantische Anomalien im Quellcode

Neben der Vermeidung von Laufzeitfehlern können mit statischen Code-Analysen weitere Programmfehler erkannt werden. Ein Beispiel dafür ist in Abbildung III.4 dargestellt.

Das Codebeispiel in der Abbildung zeigt, dass der Code-Block in der WHILE-Schleife nie ausgeführt werden kann, da die entsprechende Bedingung immer FALSE ergibt. Wie im Bild zu sehen erkennt die in Eclipse integrierte Java-Codeanalyse diese Problematik und warnt dementsprechend den Programmierer.

Hintergrund dieser Warnung ist die Annahme, dass i. d. R. kein Code mit dem Ziel implementiert worden ist, diesen bei Programmausführung nicht zu berücksichtigen. Dabei wird angenommen, dass der Entwickler nicht das im Code umgesetzt hat, was er eigentlich ausdrücken wollte. Derartige Situationen werden i. F. auch als semantische Anomalie bezeichnet.

Eine automatische Erkennung der oben beschriebenen Fehlersituationen wird erst durch die formal und maschinenverarbeitbar spezifizierte syntaktische Spezifikation von Programmiersprachen ermöglicht und kann bspw. durch symbolische Ausführung des Codes automatisch erkannt werden.

Die Behebung entsprechender Fehler muss aber manuell durchgeführt werden. Durch die automatische Code-Analyse werden die Programmierer aber insoweit unterstützt, dass der manuelle Aufwand für Erkennung, Lokalisierung und Ursachenanalyse von Fehlern in der Syntax fast vollständig reduziert wird. Nach der Behebung aller Merge-bedingten Fehler im Quellcode ist das Merge-Verfahren für Quellcode abgeschlossen.

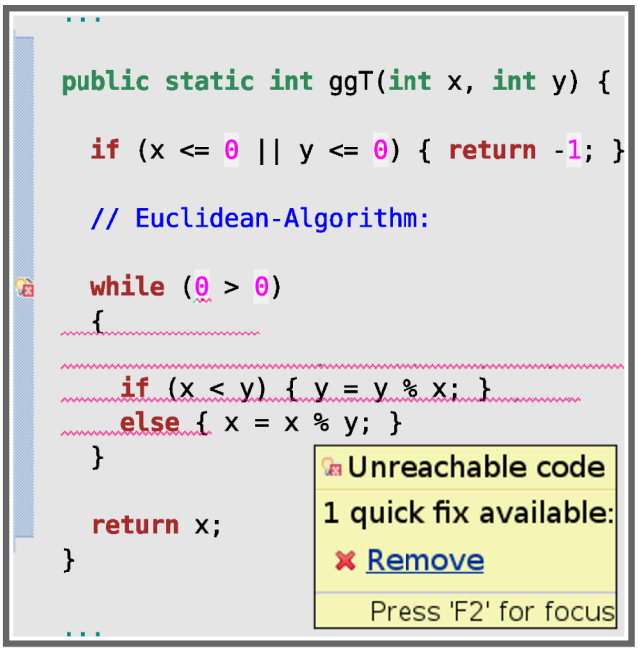


Abbildung III.4. Semantische Anomalie im Quellcode

Fazit

In der optimistischen Versionsverwaltung von Quellcode ist das vorgestellte zweistufige Merge-Verfahren weit verbreitet. Ein Grund dafür ist die skizzierte Automatisierung wesentlicher Schritte zur Synchronisation der Entwicklungsergebnisse durch eine umfangreiche und teils gut integrierte Werkzeugpalette. Zusammenfassend lässt sich festhalten, dass die softwareunterstützte Versionsverwaltung von Quellcode eine einzigartige Erfolgsgeschichte hinter sich hat und in Softwareprojekten unverzichtbar geworden ist.

Auf die kollaborative Softwaremodellierung lassen sich die Erfolge zur Unterstützung der kollaborativen Programmierung leider nicht ohne Weiteres übertragen. So scheitern alle etablierten Versionsverwaltungssysteme an einer ähnlich praktikablen Versionsverwaltung von Softwaremodellen.

Soll die Modellierungsarbeit von Teams in der Praxis koordiniert werden, muss aktuell auf pessimistische Versionierungswerkzeuge zurückgegriffen werden oder Änderungen müssen äußerst aufwendig manuell integriert werden. Da modellbasierte Ansätze immer stärkere Verbreitung in der Softwareentwicklung finden, werden seitens der Industrie und Wissenschaft vielfach Lösungen eingefordert, die eine gegenüber der Quellcodeverwaltung ähnlich eine effiziente Modellverwaltung bieten sollen (Bendix und Emanuelsson 2009b; Bendix und Emanuelsson 2008; Bendix und Emanuelsson 2009a; Brunet u. a. 2006).

3 Kollaborative Softwaremodellierung in der Analysephase – ein Illustrationsbeispiel

Ziel des folgenden Abschnittes ist die Vorstellung eines Anwendungsbeispiels für die kollaborative Modellierung von Softwaresystemen, mit dem anschließend die korrekten Problemstellungen und Lösungsansätze dieser Dissertation im Detail illustriert werden können. Dazu wird ein Ausschnitt aus der exemplarischen Entwicklung eines Analysemodells im Requirements Engineering zur Bankkontenverwaltung beschrieben. An dieser Entwicklung sind im Beispiel zwei Entwickler beteiligt, die parallel an einem Modell arbeiten. Für die Modellierung wird eine domänenspezifisch externe Modellierungssprache speziell für die Analysephase eingesetzt, die im folgenden Abschnitt eingeführt wird.

Im Anschluss wird ein Entwicklungsstand eines Modells zur Bankkontenverwaltung beschrieben, welcher der Ausgangspunkt für zwei parallele Weiterentwicklungen des Modells ist. Abschließend werden die parallelen Änderungen der Entwickler erläutert, mit denen sie zwei Modellversionen unabhängig voneinander (asynchron) weiterentwickelt haben.

3.1 Ein Metamodell für die Anforderungsanalyse

Zur Unterstützung der Anforderungsanalyse bietet die UML Sprachmittel wie Anwendungsfalldiagramme (sog. Use-Case-Diagramme) zur Verhaltensbeschreibung und Komponentendiagramme zur Strukturbeschreibung an (vgl. Abschnitt II.3.1.1). Anforderungen als eigenständige Sprachelemente stehen nicht zur Verfügung. Auch werden keine Sprachmittel angeboten, die die Anforderungen mit der Grobarchitektur des Systems in Beziehung setzen und sich bspw. Architekturmuster bedienen.

Um das Design eines domänenspezifischen Metamodells zu illustrieren, wird in Abbildung III.5 exemplarisch ein Metamodell für die Anforderungsanalyse dargestellt.

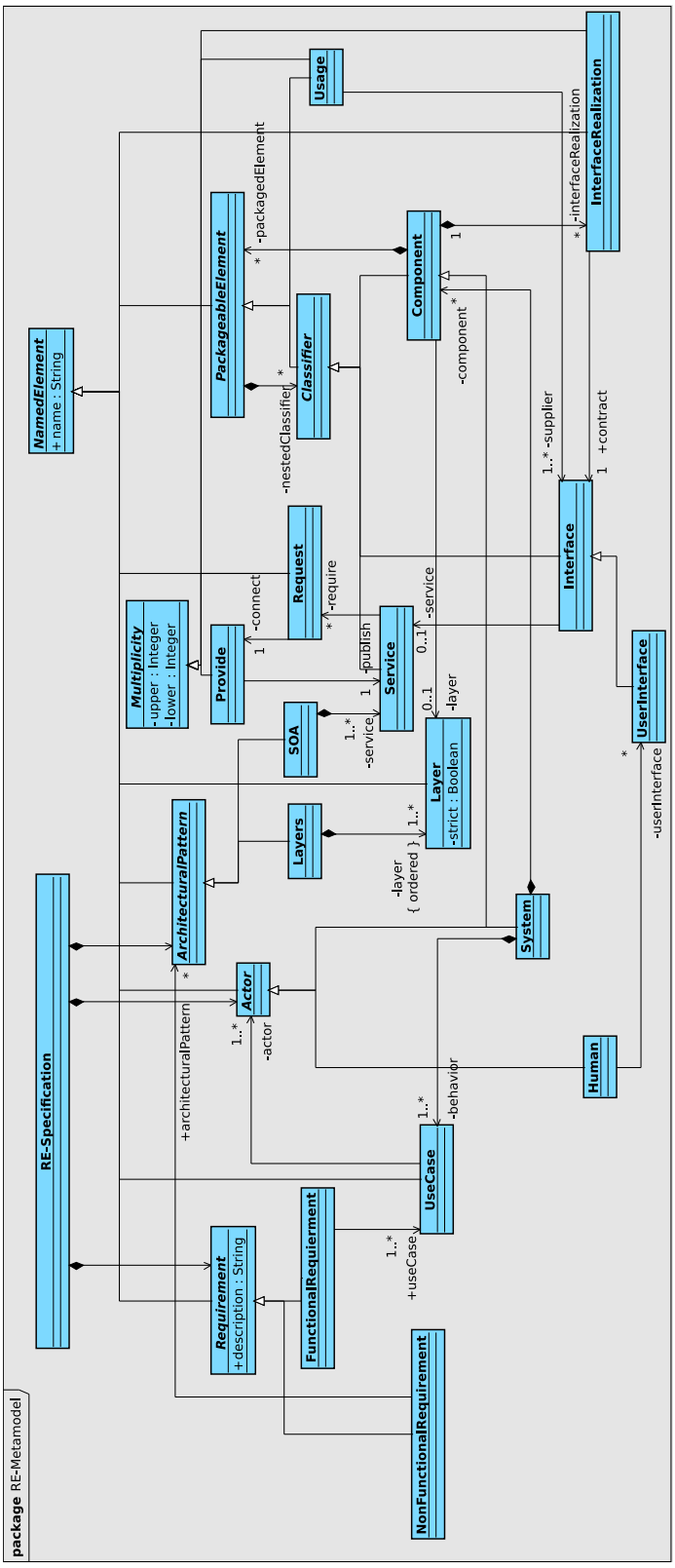


Abbildung III.5. Domänenspezifisches Metamodell für die Anforderungsanalyse – Ausschnitt

Das Metamodell ist auf Basis der Meta-Metamodellierungssprache ECore (vgl. Abschnitt II.3.3.2) erstellt worden und ist damit MOF-konform. Es ist nach der MOF-Schichtenarchitektur in die Ebene M2 einzuordnen.

Mit dem dargestellten Metamodell ist die Modellierung von Anforderungen, Anwendungsfällen und groben Komponentenstrukturen möglich. Dabei können populäre Architekturmuster wie Schichten- bzw. serviceorientierte Architektur zur Beschreibung der Grobarchitektur genutzt werden.

Jedes Modell, dessen Syntaxgraph dem RE-Metamodell in Abbildung III.5 entspricht, wird i. F. als syntaktisch korrekt bzgl. der MOF-Syntaxspezifikation angesehen. Allerdings lassen sich analog zur Spezifikation von OCL-Constraints im UML-Metamodell Bedingungen zur statischen Semantik formulieren. Für die folgenden Beispielmodelle soll die Einhaltung einer durch den Modellierer definierten Schichtenarchitektur in der Komponentenstruktur gewährleistet sein. Das bedeutet, dass keine Komponente einer unteren Schicht auf Komponenten einer oberen Schicht zugreifen darf.

Komponenten werden einer Schicht über die Referenz *layer* zugeordnet. Dabei ist die Referenz *layer* mit dem ECore-Attribut *ordered* versehen. Dieses bestimmt, dass die Reihenfolge der Abhängigkeiten zwischen den Schichten in einer strikten Schichtenarchitektur folgen muss. Die Beziehungen zwischen Komponenten werden durch die Realisierung (*InterfaceRealization*) und Nutzung (*Usage*) von Schnittstellen (*Interface*) modelliert.

Zwischen Komponenten lassen sich Nutzungsbeziehungen ausdrücken. Dazu referenzieren die *InterfaceRealization*- und *Usage*-Beziehungen ein gemeinsames *Interface*. Zudem können die *InterfaceRealization*- und *Usage*-Beziehungen mit Kardinalitäten versehen werden, da beide von der MOF-Klasse *Multiplicity* erben.

3.2 Kollaborative Modellierung einer Kontenverwaltung

Auf Grundlage des in Abbildung III.5 eingeführten Metamodells wird in diesem Kapitel ein exemplarischer Ausschnitt aus der kollaborativen Modellierung eines Kundenverwaltungssystems für Banken vorgestellt. Zwei Softwareingenieure – *Alice* (Abk. *A*) und *Bob* (Abk. *B*) – arbeiten zeitweise gleichzeitig, gemeinsam am gleichen Analysemodell zur Modellierung einer Bankkundenverwaltung.

Der nachfolgende Abschnitt beschreibt einen kleinen Ausschnitt aus dem Versionsgraphen der beiden Softwareingenieure. Dabei liegt das besondere Augenmerk auf der parallelen Entwicklung eines Modells und der anschließenden inhaltlichen Synchronisierung der Änderungen.

Abbildung III.6 zeigt dazu einen kleinen aber für die weiteren Erläuterungen ausreichend repräsentativen Ausschnitt aus der kollaborativen Modellierung eines i. F. vorgestellten Anforderungsanalysemodells.

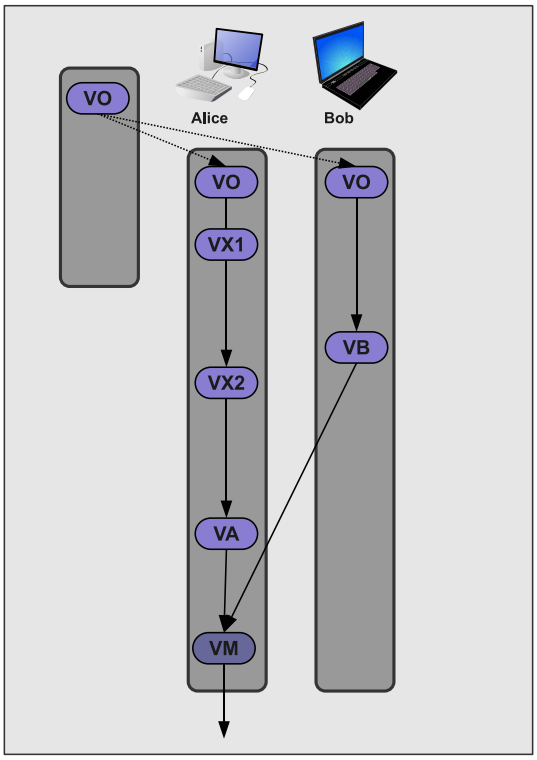


Abbildung III.6. Parallele Entwicklungsarbeit - Auszug Versionsgraph

In dem Versionsgraphen der Abbildung sind die Modellierer Alice und Bob schematisch dargestellt. Sie bearbeiten jeweils einen Entwicklungszweig, der durch eine Folge von Versionen *VO* bis *VA* bzw. *VO* bis *VB* dargestellt ist. Die Entwicklung beider Zweige geht aus von einer Version *VO* (engl. *origin version*). Diese stammt beispielsweise, wie in populären Versionsverwaltungssystemen üblich, aus einem teamweit verfügbaren Repository.

Nachdem Alice und Bob die von ihnen gewünschten Änderungen parallel zueinander nach und nach in die Versionen der Entwicklungszweige eingepflegt haben, stellt sich die Frage nach der Integration ihrer Änderungen an *VO* in eine gemeinsame Version. Durch ein automatisches Verfahren sollte idealerweise eine Modellversion *VM* bestimmt werden, welche die Änderungen von Alice und Bob enthält und die hinsichtlich merge-bedingter Fehler untersucht werden kann.

Im folgenden Abschnitt sollen Beispiele für die vier Versionen *VO*, *VA*, *VB* und *VM* vorgestellt werden, die zur Illustration von Problemen und Lösungsansätzen in den nachfolgenden Kapiteln genutzt werden.

Zunächst wird dabei die Version *VO* betrachtet. Im Modellausschnitt, der in Abbildung III.7 dargestellt ist, sind oben links drei funktionale und eine nicht-funktionale Anforderung dargestellt. Außerdem wurde eine strikte Drei-Schichten-Architektur spezifiziert und zwei Anwendungsfälle und drei Komponenten erstellt.

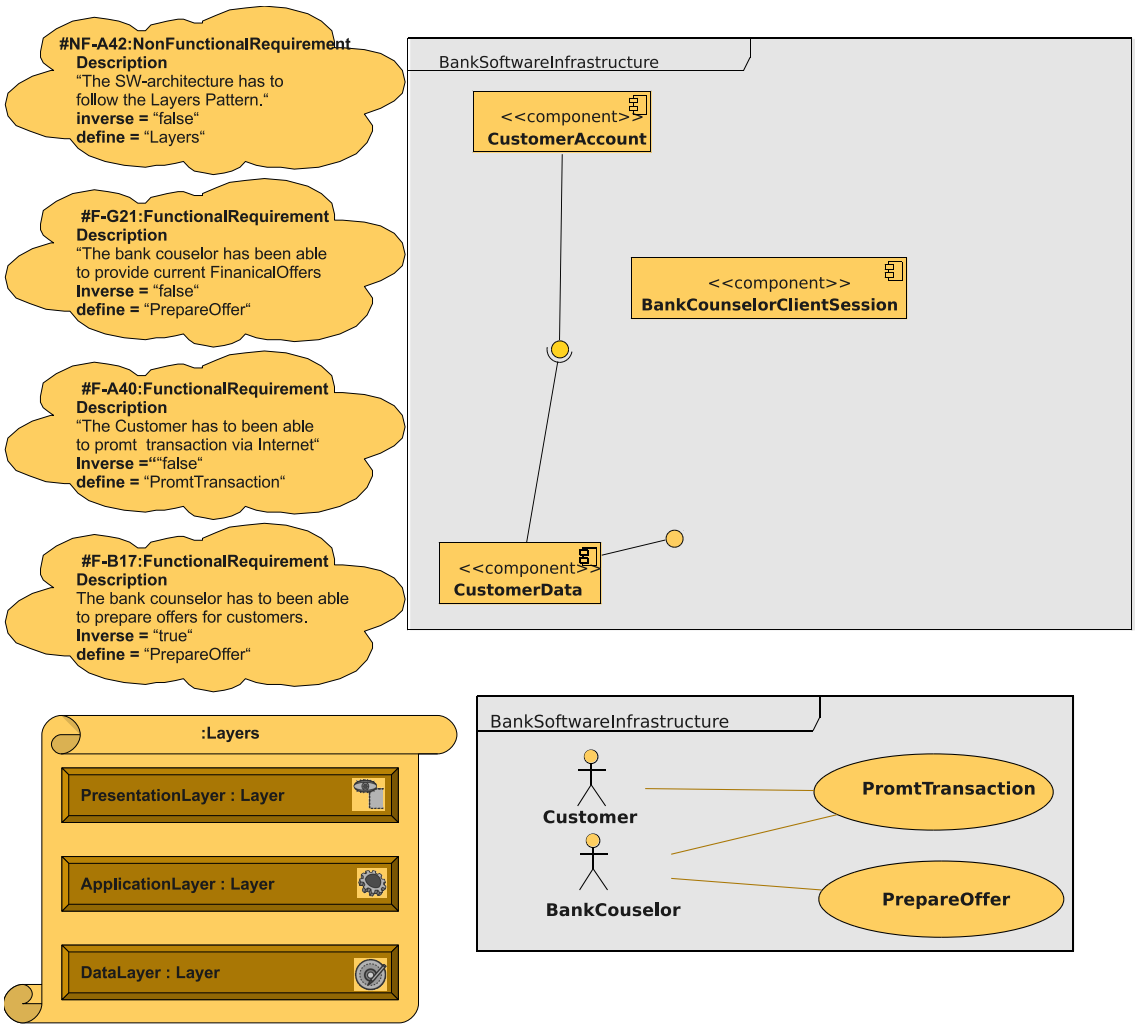


Abbildung III.7. Version VO des Beispielmodells zur Anforderungsanalyse

Auf Basis der Version *VO* modelliert Alice zwei neue Anforderungen und führt ein *MarketingSupportSystem* ein, dem sie eine vorhandene und drei neue Komponenten zuweist. Ihre Änderungen sind in Abbildung III.8 blau eingefärbt.

Sie assoziiert den Anwendungsfall *PrepareOffer* und einen neuen Anwendungsfall *SendNewsletter* mit dem neuen *MarketingSupportSystem* und führt die Rolle *Marketingmanager* ein. Darüber hinaus erstellt sie neue Schnittstellen in der Komponentenstruktur und assoziiert vier Komponenten mit der Anwendungs- bzw. Präsentationsschicht. Sie weist der Nutzungsbeziehung zwischen *CustomerAccount* und *BankCounselorClientSession* entsprechende Multiplizitäten zu.

Die Zuweisung von Komponenten zu Schichten wird durch die Markierung der Komponenten mit den jeweiligen Schichtensymbolen illustriert. Die Komponente *CustomerData* nennt sie in *CustomerProfile* um.

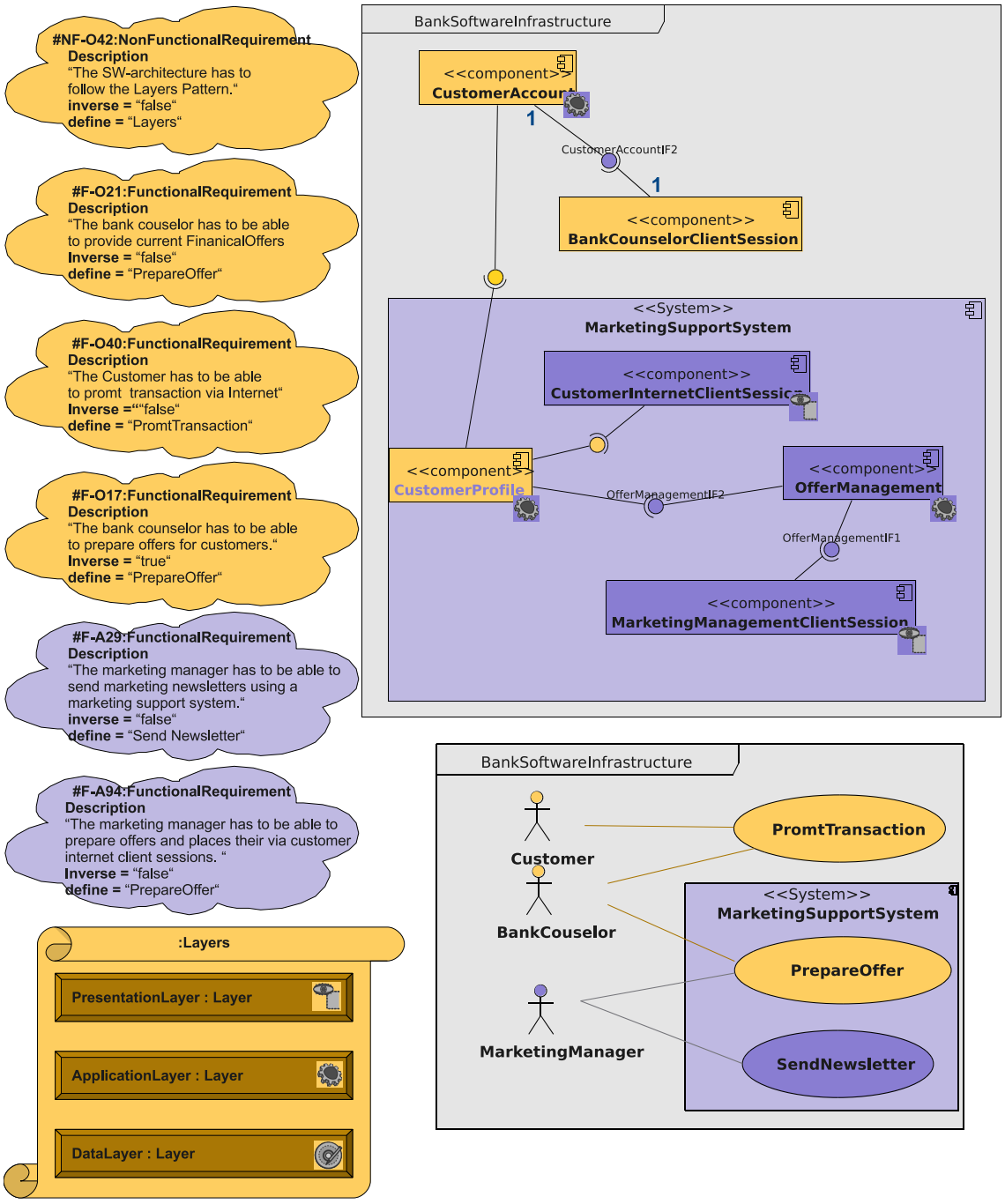


Abbildung III.8. Version *VA* des Beispielmodells zur Anforderungsanalyse bearbeitet durch *Alice*

Parallel zur Arbeit von Alice ändert Bob die Version *VO*, indem er eine neue Schicht in das Schichtenmuster einfügt und die neue Komponente *CustomerAccountTransaction* mit entsprechenden Schnittstellen erstellt. Auch er belegt die neu eingeführten Nutzungsbeziehungen zur neuen Komponente CustomerAccountTransaction mit Multiplizitäten. Seine Änderungen sind in Abbildung III.9 grün markiert.

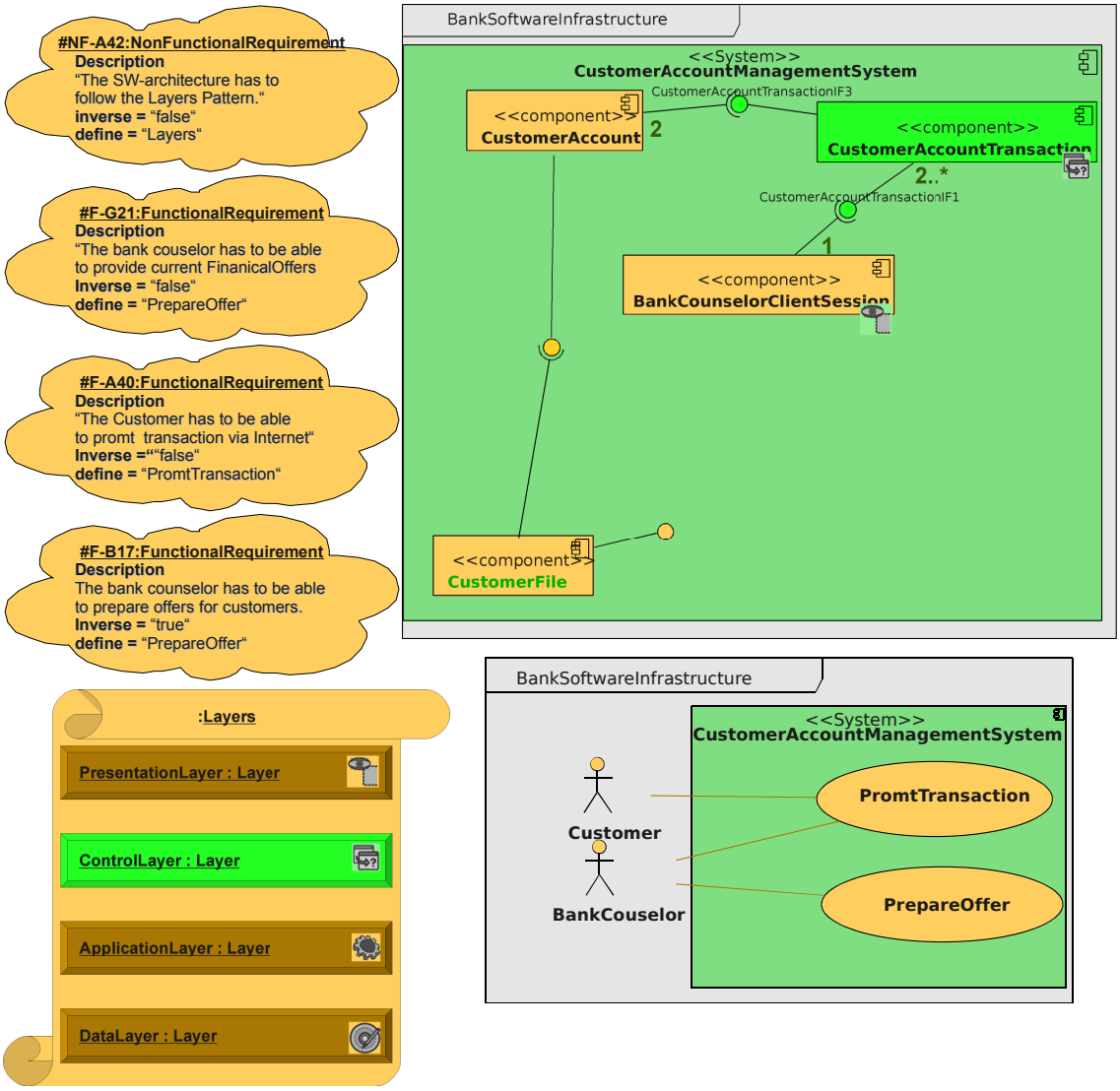


Abbildung III.9. Version *VB* des Beispielmodells zur Anforderungsanalyse bearbeitet durch *Bob*

Weiterhin weist er alle Komponenten und Anwendungsfälle einem neu erstellten System *CustomerAccountManagementSystem* zu und nennt die Komponente *CustomerData* in *CustomerFile* um. Integriert man nach dem 3-Wege-Merge-Prinzip (vgl.Abschnitt II2.2.3) nun die Änderungen am Modell, die von Version *VO* bis *VA* bzw. *VB* durchgeführt wurden auf logischer Ebene zu einem Modell, so erhält man die in Abbildung III.10 dargestellte Version *VM*.

In der Regel werden Modelldateien als werkzeugspezifische Binärdateien oder XML-Dateien auf Rechnern gespeichert. Würde es gelingen, aus Modelldateien die logische Änderungen so zu extrahieren, dass mit automatischen Verfahren ein 3-Wege-Merge durchgeführt werden

könnte, wäre das ein großer Fortschritt im Vergleich zum Status quo in der Softwaremodellierung.

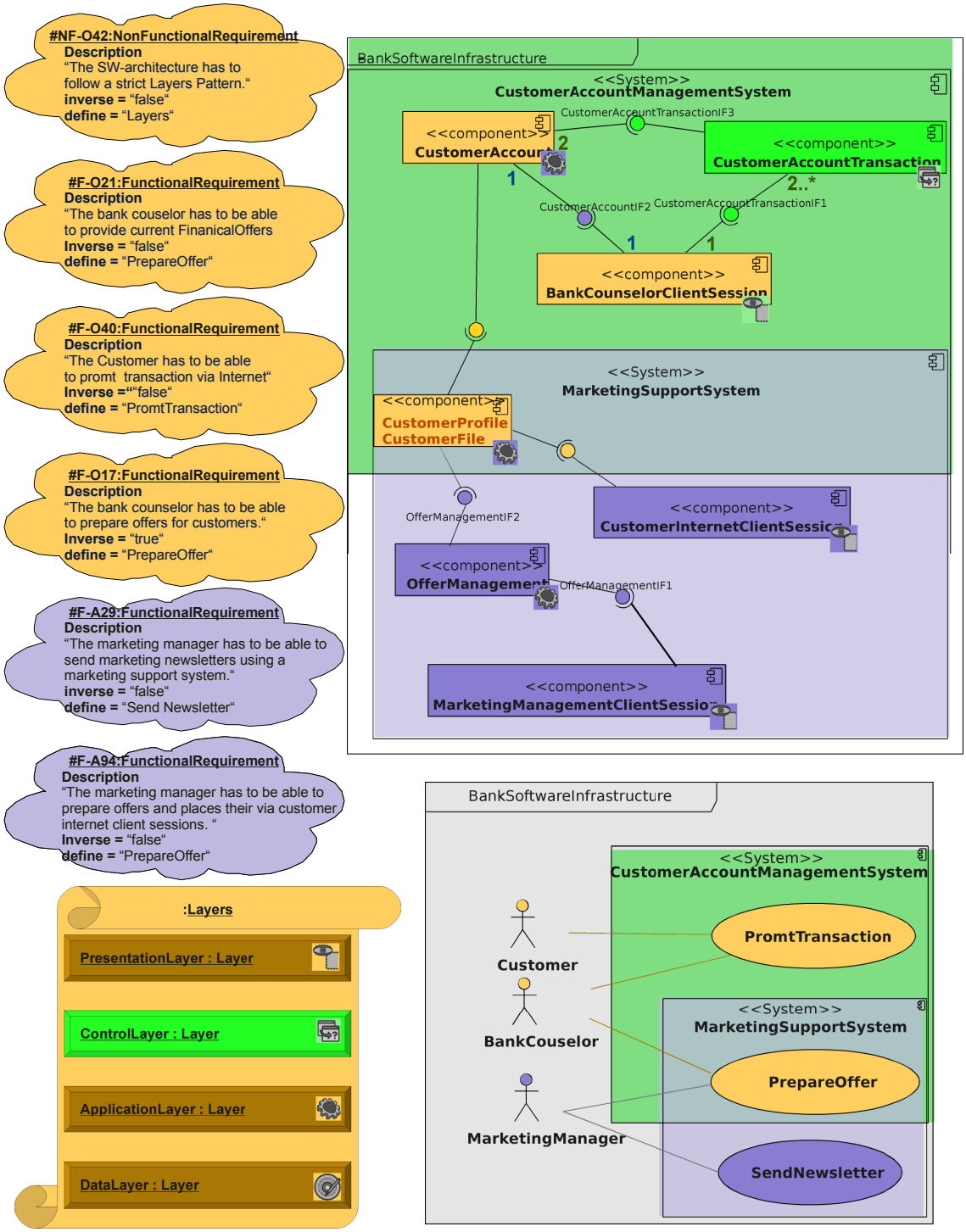


Abbildung III.10. Intuitive Anwendung des 3-Wege-Merge-Prinzips zur Erzeugung einer Version *VM* aus *VA* und *VB*

Aber selbst bei der Betrachtung logischer Modelländerungen, wie beim intuitiven Merge-Ergebnis in Abbildung III.10, können Fehler bzgl. der Metamodellspezifikation auftreten. Bspw. ist der Anwendungsfall *PrepareOffer* mit zwei Systemen assoziiert, was laut der Kompositionsbeziehung zwischen *System* und *UseCase* im Metamodell in Abbildung III.5 unkorrekt ist. Ferner widerspricht auch die Nutzungsbeziehung zwischen den Komponenten *CustomerAccount* und *CustomerAccountTransaction* der strikten Schichtenarchitektur, die durch die geordnete Liste in der Instanz des *Layers*-Musters (links unten) und der Zuweisung der entsprechenden Schichtensymbole zu den Komponenten zum Ausdruck kommt.

Ein Metamodell-unabhängiges Merge-Verfahren für Softwaremodelle, das ein Merge-Ergebnis wie in Abbildung III.10 automatisch erzeugen kann und dieses bzgl. der sprachspezifischen Korrektheit automatisch analysieren kann, wird im folgenden Abschnitt skizziert.

4 Ein Merge-Verfahren für Softwaremodelle

Grafische Modellierungssprachen im Software Engineering haben viele Gemeinsamkeiten zu textuellen Programmiersprachen, aber auch Unterschiede. In der Regel sind für Softwaremodellierungssprachen eine formale Syntaxspezifikation und mindestens eine semi-formale Beschreibung der Bedeutung von syntaktischen Konstrukten spezifiziert.

Ein wesentlicher Unterschied zwischen Quellcode und Softwaremodellen ist aber die Repräsentationsform der Informationen. Im Gegensatz zur textuellen Repräsentation von Quellcode ist die visuelle und XMI-basierte Repräsentationsform von Modellen nicht zur Anwendung von Text-Merge-Verfahren geeignet.

Dieser Umstand ist eine der Hauptursachen für das angesprochene Scheitern der Merge-Phase in der optimistischen Versionsverwaltung von Softwaremodellen. Der folgende Abschnitt belegt diese Aussage und skizziert eine Lösungsidee für ein angepasstes Merge-Verfahren für Softwaremodelle.

Würde man die Verfahrensweise des Merge-Verfahrens für Quellcode auf Softwaremodelle übertragen, so ergeben sich die folgende Arbeitsschritte:

i. Integration von Modelländerungen mit Text-Merge-Verfahren

ii. Merge-bedingte, automatische Validierung der Syntax integrierter Modellversionen und anschießende manuelle Fehlerbehebung

Die Übertragung dieser Vorgehensweise scheitert schon im ersten Schritt beim Einsatz von Text-Merge-Verfahren zur Integration asynchron durchgeführter Modelländerungen. Dies wird im folgenden Abschnitt illustriert, um anschließend die Ursachen für das Scheitern abzuleiten.

4.1 Scheitern der Integration asynchroner Modelländerungen mit Text-Merge-Verfahren

Das in Abbildung III.11 dargestellte Beispiel zeigt symptomatisch die Problematik in der Anwendung von Text-Merge-Verfahren auf Modellversionen. Das XMI-Format (Ausschnitt unten im Bild zu sehen) hat sich weitgehend zur Speicherung von Modellinformationen in der modellbasierten Softwareentwicklung durchgesetzt. Im Beispiel in Abbildung III.11 werden

die drei Modelldateiversionen *VO*, *VA* und *VB* so dargestellt, wie sie ein Modellierungswerkzeug anzeigt. Auf dem Rechner liegen sie hingegen als XMI-Dateien vor. Die Dateiversion *VM* kann nicht durch das Modellierungswerkzeug angezeigt werden und ist deshalb neben der Fehlermeldung ausschließlich in der XMI-Repräsentation dargestellt.

Schaut man sich die Ausgangsversion *VO* und die beiden geänderten Modelle der Versionen *VA* und *VB* an, so fällt es nicht schwer, sich eine Modellversion vorzustellen, in der nach dem 3-Wege-Merge-Prinzip die Änderungen von *VA* und *VB* logisch integriert sind (vgl. Abbildung III.12). Das Text-Merge-Verfahren von Subversion, das im Beispiel in Abbildung III.11 angewendet wurde, vermag dieses intuitive Merge-Ergebnis aber nicht zu erzeugen. Stattdessen werden zahlreiche Markierungen zu vermeidlichen Konflikten in die XMI-Datei des Modells eingefügt.

Die formale Syntax der XMI-Datei, die die Modellinformationen auf dem Rechner maschinenlesbar repräsentiert, wird durch dieses Vorgehen stark beschädigt. Sie lässt sich anschließend nicht mehr durch den XMI-Lexer/Parser im Modelleditor verarbeiten (vgl. II.3.2.2). Eine manuelle Reparatur derart beschädigter XMI-Dateien ist aus zwei Gründen nicht praktikabel:

- Die vermeintlich widersprüchlichen Änderungen, die in Version *VM* in Abbildung III.11 markiert wurden, resultieren aus der textbasierten Änderungsanalyse der XMI-Dateiversionen. Allerdings spiegeln Textänderungen, die in diesen Dateien erkannt werden, nicht die logischen Änderungen im Modell wieder. Im Allgemeinen werden dadurch zu viele vermeintliche Konflikte erkannt, als dass ihre manuelle Auflösung mit einem Texteditor als praktikable Alternative erscheinen würde.

  Durch Subversion werden allein in diesem einfachen Szenario 25% aller Zeilen in der Dateiversion *VM* als konfliktbehaftet markiert, obwohl sich inhaltlich die logischen Modelländerungen problemlos integrieren lassen würden. Bei der Verwaltung größerer in XML gespeicherter Modelle mit komplexeren Änderungen sieht die Situation noch drastischer aus (Meisinger, Rausch und Sihling 2006).

- Sobald die syntaktische Korrektheit der digitalen Modellrepräsentation in Form der XMI-Datei hinreichend stark durch das Text-Merge-Verfahren verletzt wurde, verweigern Modellierungswerkzeuge (XMI-Lexer/Parser) schlicht die visuelle Darstellung eines Modells. Grund dafür ist die Verletzung der XMI-Syntax, die deren Interpretation verhindert.

  Im dargestellten Fall bricht das Modellierungswerkzeug Topcased die Verarbeitung mit der in Abbildung III.11 dargestellten Fehlermeldung ab. Dadurch fehlt dem Modellierer ebenfalls jede Grundlage, das Merge-Ergebnis auf Basis der gewohnten konkreten Modellsyntax zu interpretieren. Hieran zeigt sich ein wesentlicher Unterschied zur Quellcodeverwaltung: Konflikte aufgrund widersprüchlicher Code-Änderungen können mithilfe der textuellen Markierungen durchaus von Programmierern interpretiert werden (vgl. Abbildung III.1).

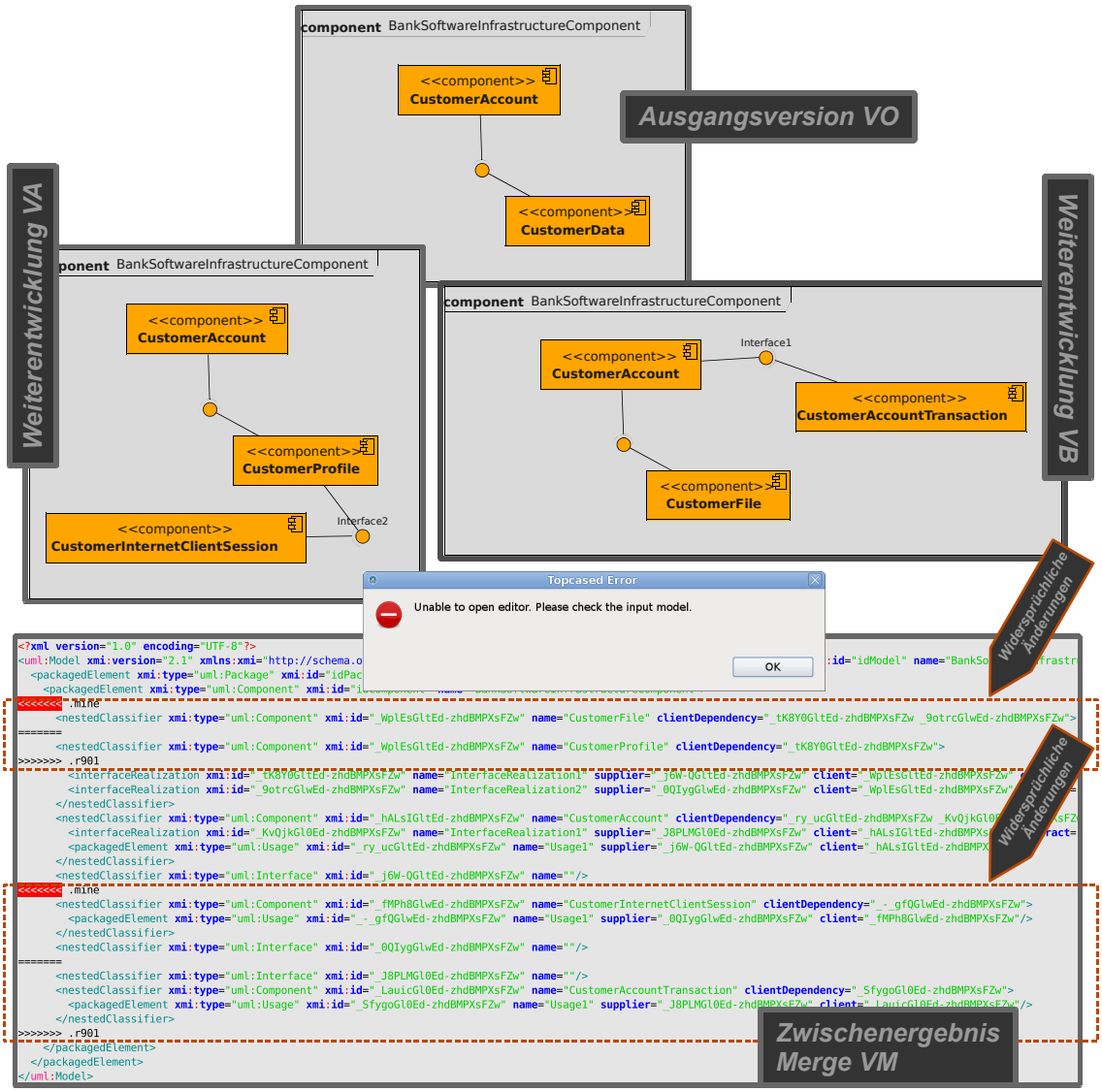


Abbildung III.11. Scheitern von Text-Merge-Verfahren bei der Integration paralleler Modelländerungen

Da das Text-Merge-Verfahren (Mens 2002) einerseits viele vermeintliche Konflikte an den Modelldateien erkennt, die nicht auf inhaltlich widersprüchliche Änderungen am Modell zurückzuführen sind und andererseits syntaktische Beschädigungen an den Modelldateien hervorruft, die die grafische Darstellung des Modells verhindern, ist es für den Einsatz in der optimistischen Versionsverwaltung von Softwaremodellen ungeeignet. Der zeilenbasierte 2-Wege-Vergleich zur Analyse von Änderungen zwischen Modellversionen erweist sich im Gegensatz Quellcodeversionen als unzweckmäßig.

Aufgrund des Scheiterns der Integration von Änderungen im Schritt i. (vgl. Seite 87) kann eine anschließende modellierungssprachspezifische Fehlerprüfung der Modelldatei zunächst nicht durchgeführt werden. Mit dem morphologischen Merge-Verfahren, das in den beiden nächsten Abschnitten skizziert wird, können parallele Änderungen wie intuitiv erwartet

integriert werden. Die dann anschließend mögliche automatische Validierung des Modells durch statische Analyse der Syntax wird in Abschnitt 4.4 motiviert.

4.2 Intuitive Integration logischer Modelländerungen

Wie im letzten Abschnitt angesprochen lassen sich die Modelländerungen der Versionen *VA* und *VB* in Abbildung III.11 nach dem 3-Wege-Merge-Prinzip ohne Weiteres intuitiv integrieren. Betrachtet man statt Änderungen an Textzeilen in der XMI-Datei die inhaltlichen/logischen Modelländerungen an syntaktischen Bausteinen wie die Namensänderung der Komponente *CustomerData* und das Hinzufügen von *CustomerAccountTransaction* und *CustomerClientSession* mit den entsprechenden Schnittstellen als Änderungen, so ergibt sich nach dem 3-Wege-Merge-Prinzip ein Modell, wie es in Abbildung III.12 dargestellt ist. Der Konflikt durch die unterschiedliche Umbenennung von *CustomerData* muss natürlich vorher manuell aufgelöst worden sein (Im Beispiel zugunsten von *CustomerProfile*).

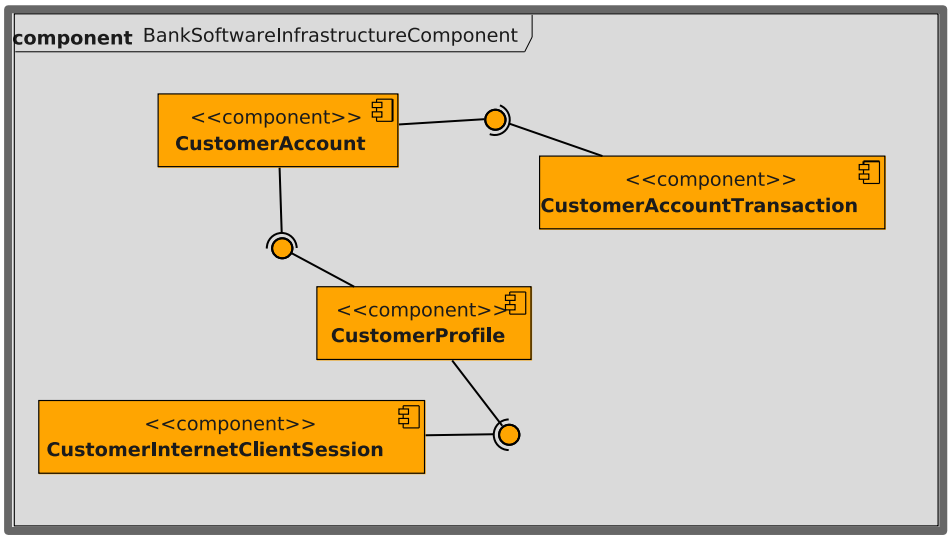


Abbildung III.12. Intuitives Merge-Ergebnis durch Integration logischer Änderungen nach dem 3-Wege-Merge-Prinzip

Das Modell in Abbildung III.12 wäre somit ein adäquates Ergebnis der Änderungsintegration von *VA* und *VB*. Es lässt sich sowohl im XMI-Format als auch grafisch mit einem Modellierungswerkzeug darstellen; Konflikte wegen widersprüchlicher Änderungen in der Modelldatei, wie sie noch in Abbildung III.11 zu sehen sind, müssen nicht berücksichtigt werden.

Dennoch kann es zu potenziell widersprüchlichen Änderungen des Modells auch bei der Betrachtung logischer Modelländerungen kommen, sofern Modellinformationen in parallelen Weiterentwicklungen auf unterschiedliche Art und Weise geändert wurden. Die Behandlung solcher Konflikte ist analog zur Situation bei widersprüchlichen Zeilenänderungen im Rahmen des Text-Merge-Verfahrens.

Eine Möglichkeit zur Darstellung dieser Konflikte wird in Abbildung III.13 gezeigt. Von zentraler Bedeutung ist hierbei die Darstellung des Konfliktes in der vom Modellierer

gewohnten konkreten Syntax des Modells. Dazu muss, wie in der Abbildung zu sehen, die Modelldatei der Version *VM* durch das Modellierungswerkzeug fehlerfrei verarbeitet und angezeigt werden können. Die Markierung der widersprüchlichen Modelländerung muss in der Modelldarstellung des Werkzeugs für die Modellierer intuitiv verständlich sein.

Das Beispiel in Abbildung III.13 zeigt die Umbenennung der Komponente *CustomerData* in den parallel weiterentwickelten Versionen *VA* und *VB*. In diesen wurde die Komponente einerseits in *CustomerFile* und andererseits in *CustomerProfile* umbenannt.

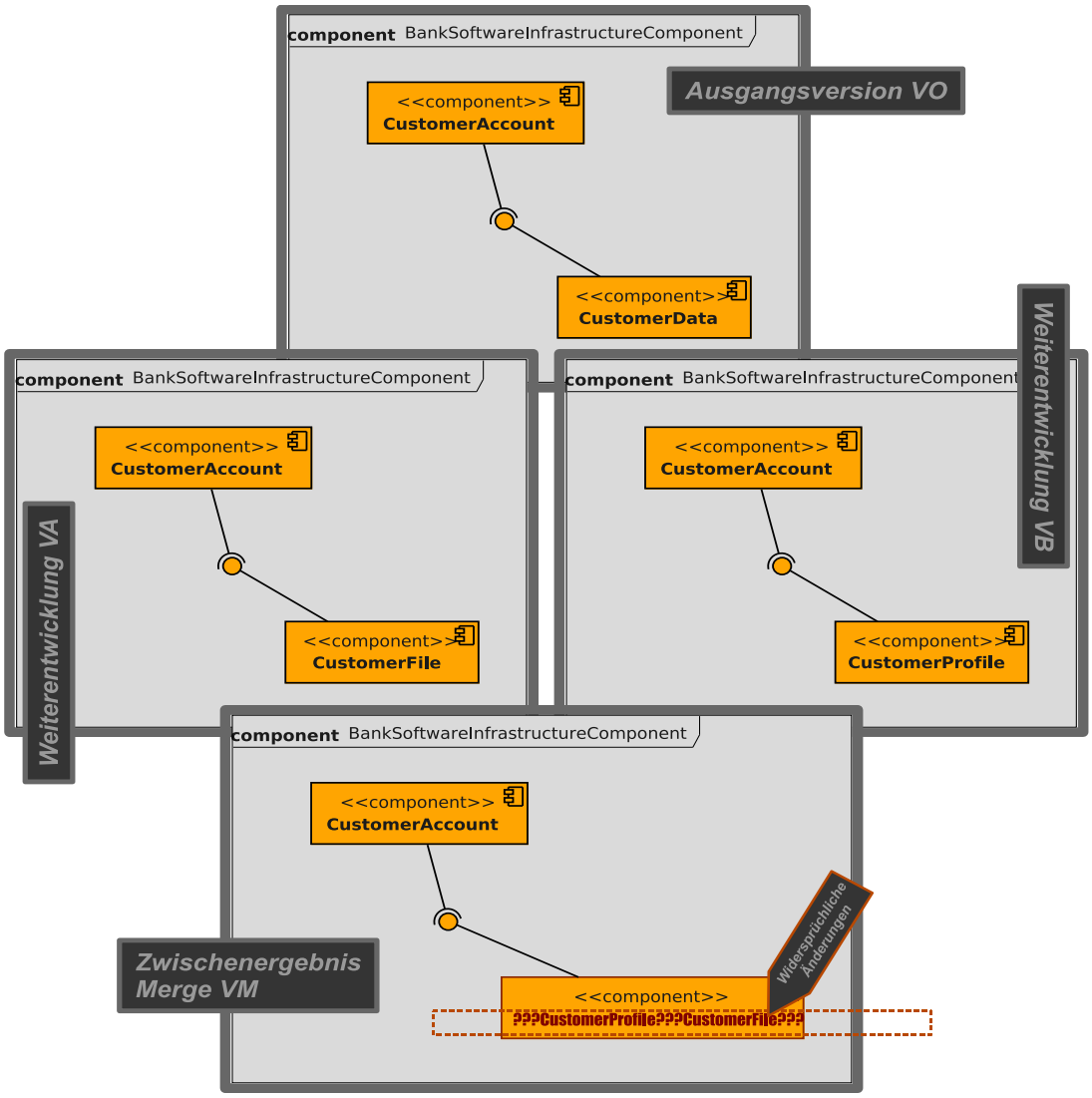


Abbildung III.13. Widersprüchliche Änderungen beim morphologischen Merge-Verfahren

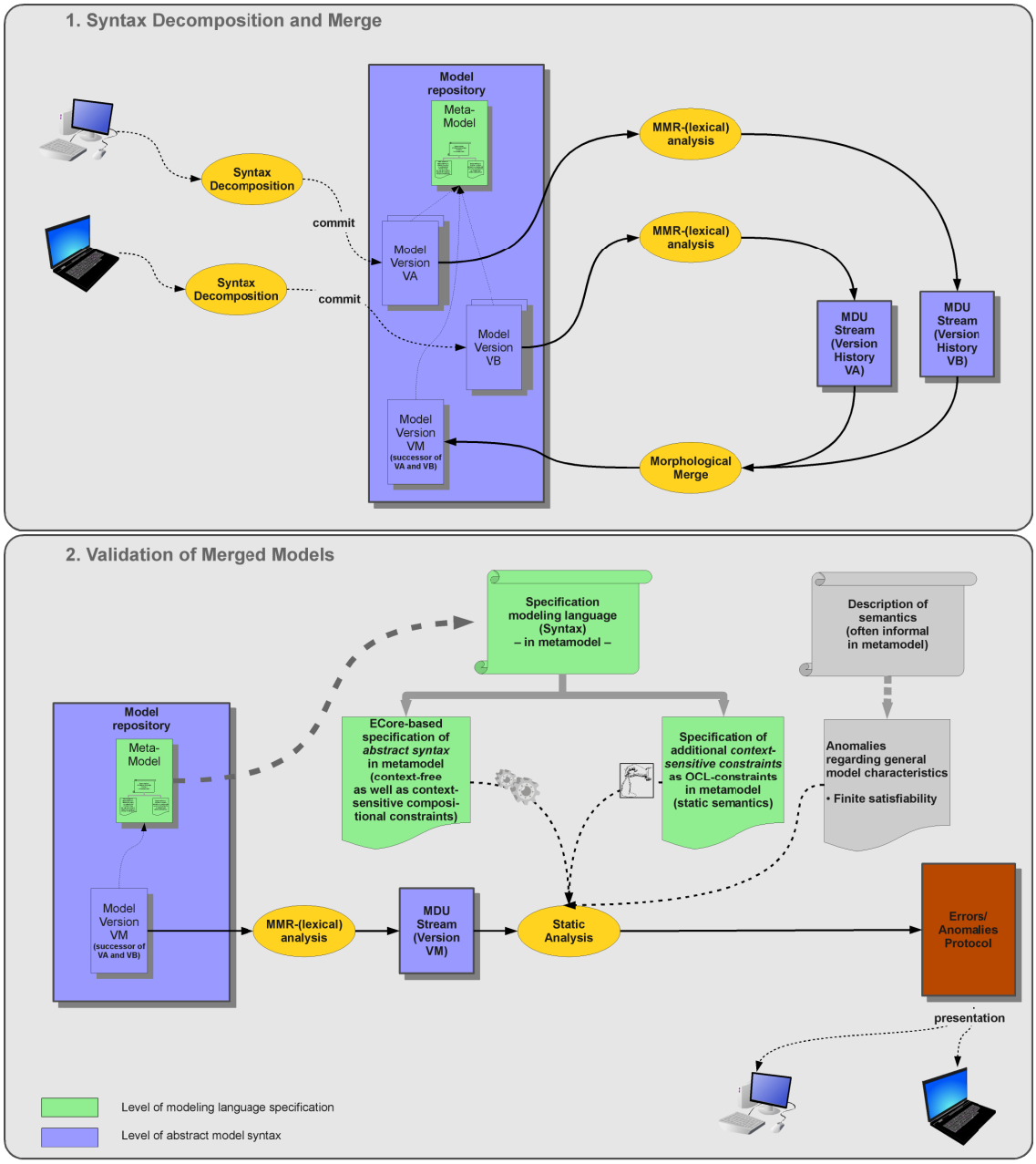


Abbildung III.14. Überblick - Korrekte Integration asynchroner Modellbearbeitungen

Ein Softwarewerkzeug, das die Integration paralleler Modelländerungen unterstützt, kann in diesem Beispiel nicht automatisch entscheiden, wie die Komponente letztendlich benannt werden soll und somit das Problem nicht automatisch beheben. Eine Werkzeugunterstützung zur Erkennung und Analyse dieses Konfliktes ist aber insbesondere bei der Verwaltung komplexer, im Team bearbeiteter Modelle unerlässlich für eine effiziente Synchronisation der Teamarbeit.

Mit dem skizzierten Verfahren wird das Einpflegen von asynchronen Änderungen in eine gemeinsame Modellversion ermöglicht. Dabei muss eine eventuelle Konfliktlösung bzgl. widersprüchlichen Änderungen rechnergestützt angeboten werden.

Das durch das Merge-Verfahren bestimmte Ergebnismodell kann hinsichtlich seiner Modellierungssprache weitere Merge-bedingte Fehler enthalten. Analog zur Korrektheitsanalyse von Quellcode nach Anwendung des Text-Merge-Verfahrens bedarf es auch hier ein effizientes Verfahren zur Erkennung und Analyse von Fehlern.

Abbildung III.14 stellt das motivierte Verfahren überblicksartig da. Es teilt den Prozess zur Integration von Änderungen asynchron erstellter Versionen in zwei Phasen auf. In der ersten Phase werden auf Basis einer Zerlegung der zu integrierenden Modellversionen in syntaktische Elementarbausteine (Syntax Decomposition) parallel durchgeführte Änderungen zu einer neuen Version zusammengeführt (Morphological Merge).

Das Ergebnis dieses Merges wird in der zweiten Phase durch eine statische Analyse validiert. Grundlage dieser Untersuchung bildet die Syntaxspezifikation des Metamodells (basierend auf MOF/OCL) und eine Beschreibung statisch analysierbarer Anomalien in Modellen.

Bevor in Abschnitt III.4.4 auf die Validierung von Modellen eingegangen wird, soll im folgenden Abschnitt skizziert werden, wie das beschriebene intuitive Merge-Verfahren automatisiert werden kann, um eine geeignete Werkzeugunterstützung zu implementieren.

4.3 Konzept eines morphologischen Merge-Verfahrens für Modelle auf Basis syntaktischer Zerlegung

Das in Abschnitt II.2.2.3 beschriebene 3-Wege-Merge-Prinzip macht keine Vorgaben zur Strukturierung einer Version in Dateneinheiten, deren Änderungen durch ein Merge-Verfahren betrachtet werden. Text-Merge-Verfahren nutzen eine Zerlegung von Dateien in Textzeilen. Diese Art der Zerlegung ist ursächlich für das oben beschriebene Scheitern von Text-Merge-Verfahren bei der Verwaltung von Modelldateiversionen.

Durch eine alternative Zerlegung von Modelldateien kann allerdings das oben beschriebene intuitive Merge-Verfahren für Modelle erfolgreich automatisiert werden. Die Kriterien für diese Zerlegung sollen in diesem Abschnitt detaillierter erläutert werden.

Im letzten Abschnitt wurde von einem intuitiven Merge-Verfahren und von der Integration logischer Modelländerungen gesprochen. Doch was bedeuten logische Modelländerungen konkret? Dieser Frage soll zunächst nachgegangen werden. Dazu sei folgende Definition gegeben:

Definition III.1. Logische Modelländerung

Eine *generische logische Modelländerung* bezeichnet das Löschen oder Hinzufügen einer Modellinformation zu einem Modell.

Da Modelländerungen unabhängig von einer spezifischen Modellierungssprache verarbeitet werden sollen, wird in der Definition von sehr abstrakten (generischen) Änderungen ausgegangen.

Sollen logische Modelländerungen maschinell verarbeitet werden, so ist deren Identifizierung in der Versionshistorie eines Modells der erste Schritt. Dazu ist es notwendig, die Hinzufügungen und Löschungen von Modellinformationen aufeinanderfolgender Versionen zu erkennen.

Zustandsbasierte Merge-Verfahren (vgl. Seite 48) vergleichen dazu die entsprechenden Dateiversionen. Diese enthalten die digitale Repräsentation der Modellinformationen. Zur Durchführung des angesprochenen Modellvergleiches müssen die Modellinformationen, die in einer Modelldatei repräsentiert sind, als einzelne maschinell verarbeitbare Dateneinheiten vorliegen.

Das i. F. skizzierte Verfahren bestimmt dazu aus einer Modelldatei alle morphologischen Dateneinheiten. Der Begriff morphologisch leitet sich sinngemäß vom Begriff Morph ab. Als Morph wird eine atomare bedeutungstragende Einheit einer Sprache benannt (Hammarström 1966). Daraus ergibt sich die folgende Definition einer morphologischen Dateneinheit:

Definition III.2. Morphologische Dateneinheit

Eine *morphologische Dateneinheit* (MDU – engl. *morphological data unit*) ist ein maschinenverarbeitbarer Datensatz, der genau einer atomaren Modellinformation (Morph) entspricht.

Die Zerlegung eines Modells in morphologische Dateneinheiten soll anschließend anhand eines Beispiels illustriert werden. Dazu wird auf das Beispiel in Abbildung III.11 zurückgegriffen. Es zeigt, wie ein Modell einer Ausgangversion *VO* in zwei parallelen Weiterentwicklungen *VA* und *VB* jeweils um neue Komponenten und Schnittstellen erweitert wird.

Beim morphologischen Merge-Verfahren werden zunächst *VA* und *VB* jeweils mit *VO* verglichen. Bei diesem Vergleich werden Änderungen an jeder atomaren Modellinformation berücksichtigt. Für die Weiterentwicklung zur Version *VB* bedeutet das, dass zehn atomare Informationseinheiten hinzugefügt wurden. Diese sind in Abbildung III.15 dargestellt und werden anschließend erläutert:

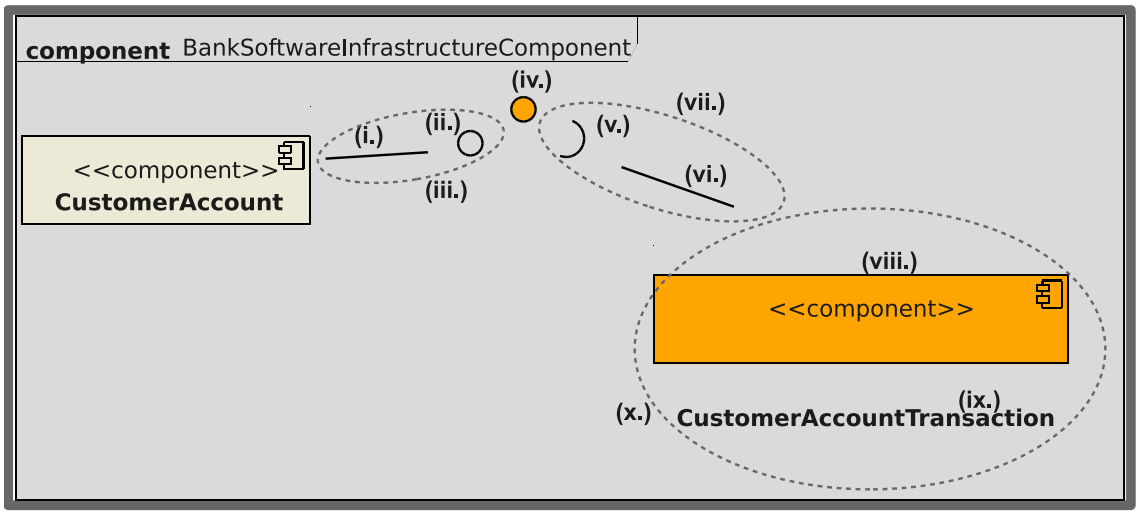


Abbildung III.15. Betrachtung atomarer Informationseinheiten beim morphologischen Merge

Abbildung III.15 zeigt die Zerlegung des dargestellten Modellausschnittes der Version *VB* in folgende morphologische Dateneinheiten:

i. Der Anfang der *Interface-Realization*-Beziehung (iii.) ist mit der Komponente *CustomerAccount* verknüpft.

ii. Das Ziel der *Interface-Realization*-Beziehung (iii.) ist mit dem durch iv. identifizierten Modellelement verknüpft.

iii. Es existiert ein neues Modellelement vom Typ *Interface-Realization* (Identifizierer in der Abbildung: iii.)

iv. Es existiert ein neues Modellelement vom Typ *Interface* (Identifizierer in der Abbildung: iv.)

v. Das Ziel der *Usage*-Beziehung (vii.) ist mit dem durch iv. identifizierten Modellelement verknüpft.

vi. Der Anfang der *Usage*-Beziehung (vii.) ist mit dem durch viii. identifizierten Modellelement verknüpft.

vii. Es existiert ein neues Modellelement vom Typ *Usage* (Identifizierer in der Abbildung vii.)

viii. Es existiert ein neues Modellelement vom Typ *Component* (Identifizierer in der Abbildung viii.)

ix. Es existiert ein neues Datum mit dem Wert *CustomerAccountTransaction* (Identifizierer in der Abbildung ix.)

x. Das Datum ix. wird als Bezeichnung (*name*) dem Modellelement viii. zugewiesen.

Die morphologischen Dateneinheiten i.-x. werden der Modellversion *VO* in der Modellversion *VB* hinzugefügt. Diese Hinzufügungen entsprechen den logischen Modelländerungen zwischen den beiden Versionen *VO* und *VB*. Für die Weiterentwicklung *VA* werden analog die logischen Modelländerungen zwischen *VO* und *VA* bestimmt.

Mit der vorgestellten Zerlegung eines Modells in morphologische Dateneinheiten ist die Grundlage für die Anpassung des Merge-Verfahrens für Modelle geschaffen. Dabei soll der grundsätzliche Charakter des intuitiven und etablierten 3-Wege-Merge-Prinzips erhalten bleiben. Lediglich die Art der Zerlegung von Versionen in Dateneinheiten, die für den Änderungsvergleich notwendig ist, muss angepasst werden. Der dabei entscheidende Aspekt ist, dass Änderungen an logischen Modellinformationen anstatt an Änderungen der Textrepräsentation betrachtet werden.

Die Integration der logischen Modelländerungen kann anschließend nach dem 3-Wege-Merge-Prinzip erfolgen. Da in den beiden Versionen *VA* und *VB* des Beispiels nur Dateneinheiten hinzugefügt wurden, treten bei der Integration dieser Änderungen keine Konflikte auf. Die resultierende Version entspricht damit der intuitiv erwarteten Modellversion *VM*, wie sie in Abbildung III.12 dargestellt ist.

Nicht in jedem Fall ist nach dem Integrieren paralleler Modelländerungen mit dem morphologischen Merge-Verfahren das Resultat frei von Fehlern. Durch die Integration

paralleler Änderungen besteht die Möglichkeit, dass die modellierungssprachspezifische Korrektheit (Syntax incl. statischer Semantik) des Ergebnismodells verletzt ist. Mit der automatischen Erkennung derartiger Fehler beschäftigt sich der folgende Abschnitt.

4.4 Konzept der automatischen Validierung integrierter Modellversionen durch statische Analyse

Das in den letzten Abschnitten skizzierte morphologische Merge-Verfahren verhindert das Auftreten vermeintlicher Merge-Konflikte, die nicht durch Integrationsprobleme logischer Modelländerungen begründet sind. Allerdings können aber durchaus andere Merge-Konflikte bei der Integration paralleler Modelländerungen auftreten. Sie äußern sich durch das Auftreten syntaktischer Fehler oder semantischer Anomalien im Ergebnismodell des morphologischen Merge-Verfahrens. Dazu werden in den beiden folgenden Abschnitten die Herausforderungen der merge-bedingten Validierung von Modellen anhand ihrer Syntax beschrieben.

4.4.1 Merge-bedingte Verletzung der MOF-Syntaxspezifikation von Modellen

Zur Illustration der Problematik bei der Erkennung Merge-bedingter Syntaxfehler in Modellen werden in Abbildung III.16 und Abbildung III.18 zwei typische Szenarien für das Auftreten von Syntaxfehlern bei der Integration paralleler Modelländerungen dargestellt. Dazu werden die oben beschriebenen Modelle weiter durch Alice und Bob geändert.

In Abbildung III.16 wurde einerseits die in der Ausgangsversion *VO* vorhandene Komponente *CustomerAccountTransaction* gelöscht (Version *VB*) und andererseits wurde eine neue Schnittstelle mit entsprechenden *InterfaceRealization*- und *Usage*-Beziehungen zwischen *CustomerAccountTransaction* und *CustomerInternetClientSession* erstellt (Version *VA*).

Durch das Integrieren der logischen Modelländerungen mit dem morphologischen Merge-Verfahren entsteht ein Zwischenergebnis *VM*. Dies ist allerdings nicht syntaktisch korrekt, weil die in *VA* hinzugefügte *InterfaceRealization* in *VM* ausschließlich mit der neuen (in *VA* hinzugefügten) Schnittstelle verbunden ist.

Wegen der Löschung von *CustomerAccountTransaction* referenziert diese *InterfaceRealization*-Beziehung in *VM* eine nicht mehr vorhandene Komponente. Laut Metamodell der Modellierungssprache muss eine *InterfaceRealization* aber genau zu einer Komponente gehören (vgl. Abbildung III.17).

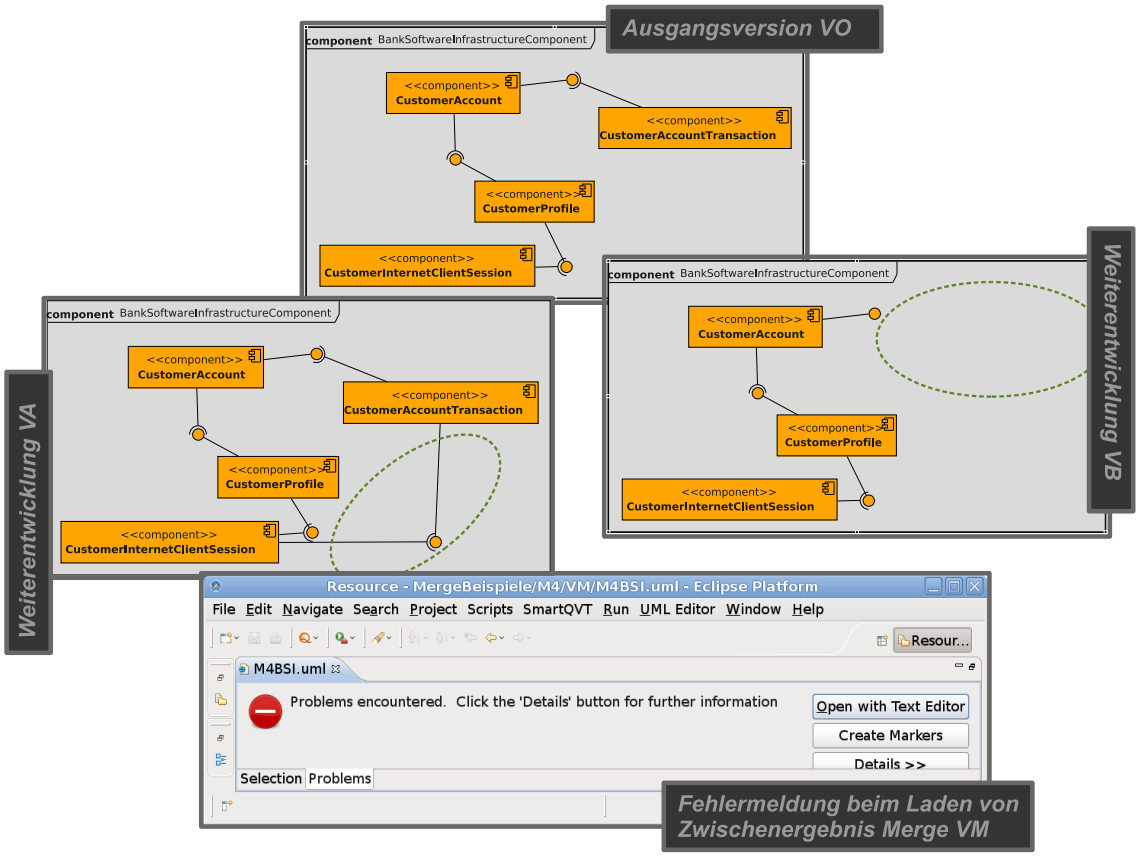


Abbildung III.16. Syntaxfehler nach Anwendung des morphologischen Merge-Verfahrens – Beispiel 1

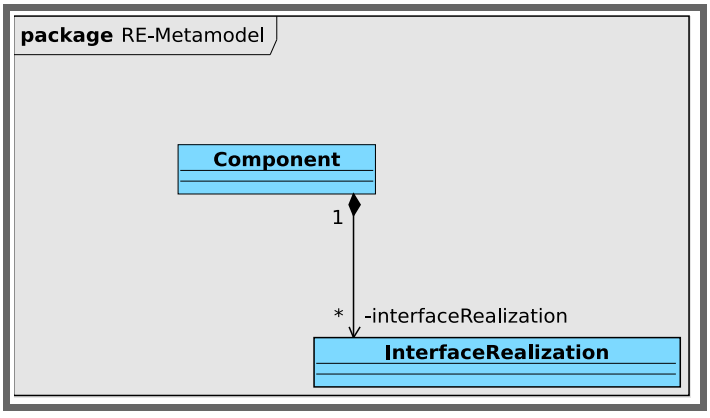


Abbildung III.17. Beziehungstyp zwischen Komponente und InterfaceRealization im RE-Metamodell

Syntaktische Fehler wie in Abbildung III.16, bei denen Beziehungsenden auf gelöschte Modellelemente verweisen sind typisch für Konflikte beim Merge parallel entwickelter Modelle. Eine weitere Konfliktklasse beim Merge, ist die Verletzung von hierarchischen Strukturen wie bspw. der durch die Kompositionsbeziehung erzeugten Strukturen. Ein Beispiel für einen solchen syntaktischen Fehler ist in Abbildung III.18 dargestellt.

In Abbildung III.18 wird wieder ein Ausschnitt aus dem Szenario in Abbildung III.10 dargestellt, wie in Version *VA* und *VB* der Anwendungsfall *PrepareOffer* jeweils mit unterschiedlichen Systemen assoziiert wird. Nach Anwendung des morphologischen Merge-Verfahrens müsste der Anwendungsfall *PrepareOffer* das Verhalten (*behavior*) zweier Systeme *CustomerManagementSystem* und *MarketingSupportSystem* gleichzeitig beschreiben. Diese Zugehörigkeit ist laut der MOF-Syntaxspezifikation syntaktisch nicht korrekt, wie der Metamodellausschnitt in Abbildung III.19 zeigt.

Das als Zwischenversion *VM* bezeichnete Modell in Abbildung III.18 ist manuell als Bildmontage erstellt worden. Es lässt sich nicht mit einem Modellierungswerkzeug anzeigen und noch nicht einmal in der XMI-Repräsentation ausdrücken, da in XMI die Kompositionsbeziehung von Metamodellen zum hierarchischen Aufbau der XML-Struktur nutzt.

Die beschriebenen Syntaxfehler im abstrakten Syntaxgraphen des Modells haben zur Folge, dass Modellierungswerkzeuge u. U. mit einer Fehlermeldung die Verarbeitung komplett abbrechen (vgl. Abbildung III.16 unten), bzw. dass keine Modelldatei mit dem Merge-Ergebnis erstellt werden kann wie im Beispiel in Abbildung III.18.

Eine manuelle Behebung solcher Merge-Konflikte ist in diesen Fällen nicht ohne weitere Unterstützung möglich. Für eine praktikable Unterstützung bei der Erkennung und Behebung Merge-bedingter Syntax-Fehler wird ein automatisches Verfahren benötigt, das die logischen Modelländerungen identifiziert, die für das Auftreten dieser Fehlern verantwortlich sind.

Dazu muss zunächst das Ergebnismodell des morphologischen Merge-Verfahrens hinsichtlich seiner syntaktische Korrektheit maschinell validiert werden. Anschließend können die Modelländerungen bestimmt werden, die für das Auftreten des Fehlers verantwortlich sind.

Eine syntaktische Analyse ist für viele Modellierungssprachen im Software Engineering möglich, da ihre Syntax i. d. R. formal spezifiziert ist. Analog zur syntaktischen Analyse von Quellcode durch Compiler kann so eine automatische Syntaxanalyse von Softwaremodellen konzipiert werden. Das zu entwickelnde Merge-Verfahren soll, wie in den Rahmenbedingungen gefordert (vgl. I.4.1), Modellversionen beliebiger Modellierungssprachen verarbeiten können.

Die dafür benötigte Werkzeugunterstützung muss somit die Syntaxspezifikation (Metamodell) einer beliebigen Modellierungssprache gegen entsprechende Ergebnismodelle des morphologischen Merge-Verfahrens validieren. Dieses Konzept wurde schon in Abbildung II.13 überblicksartig dargestellt und wird in Abschnitt 2 des folgenden Kapitels im Detail erläutert.

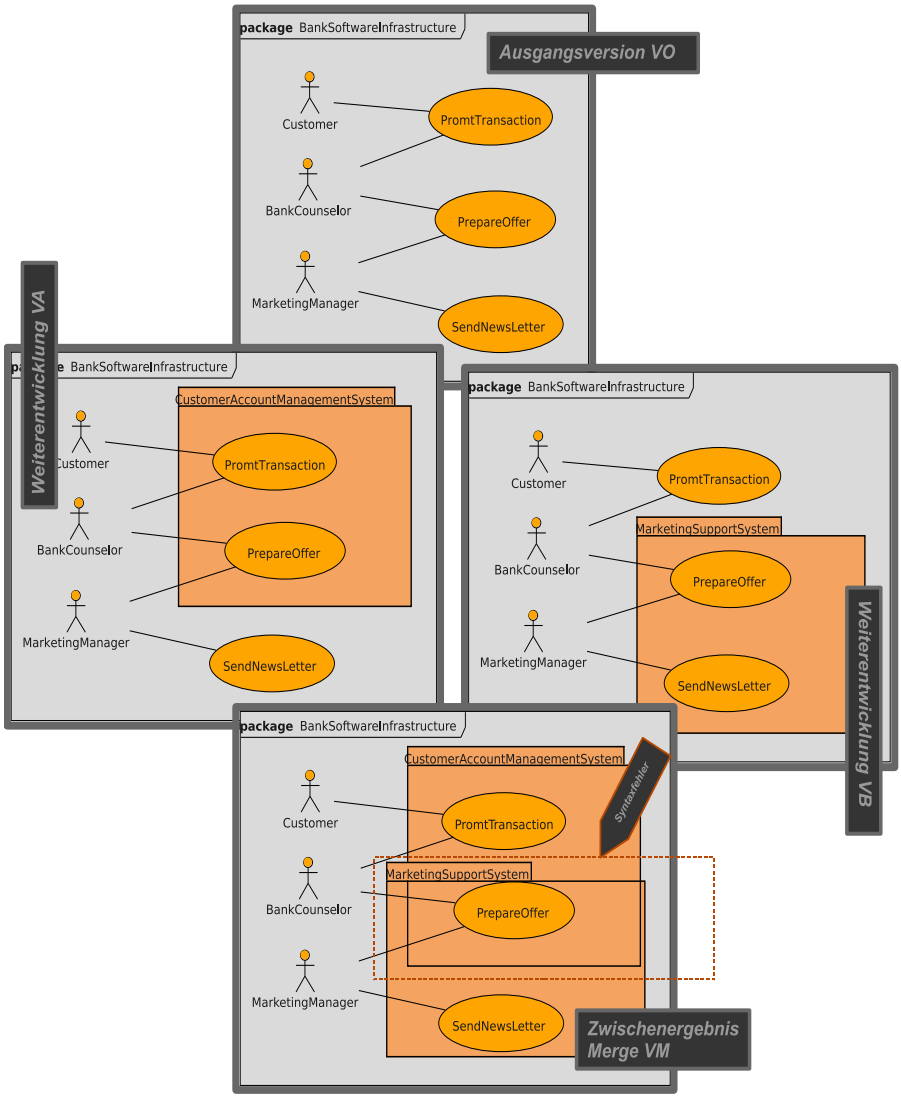


Abbildung III.18. Syntaxfehler nach Anwendung des morphologischen Merge-Verfahrens – Beispiel 2

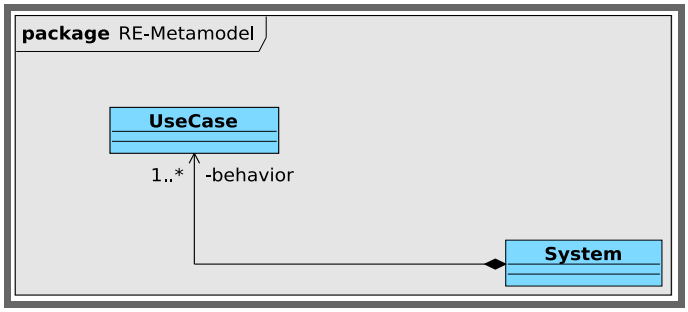


Abbildung III.19. Beziehungstyp zwischen Anwendungsfall und System im RE-Metamodell

Nachdem die Modellelemente identifiziert wurden, deren Vorhandensein oder Fehlen im gemergten Modell Syntaxfehler nach sich ziehen, können die ursächlichen logischen Modelländerungen (Hinzufügung oder Löschung) in der Versionshistorie bestimmt werden. Für das oben eingeführte Beispielszenario stehen die *Hinzufügung* der *InterfaceRealization* in Version *VA* und die *Löschung* der Komponente *CustomerAccountTransaction* in Version *VB* in Konflikt zueinander.

Damit dieser Konflikt behoben werden kann, muss er für die Modellierer verständlich dargestellt werden. Beispielsweise kann der Konflikt durch die Markierung der verursachenden Änderungen in den Versionen *VA* und *VB* visualisiert werden. Die detaillierte Erforschung einer geeigneten bildlichen Darstellung solcher Konflikte ist allerdings nicht Gegenstand dieser Dissertation.

Neben Syntaxfehlern können Ergebnismodelle des morphologischen Merge-Verfahrens Merge-bedingte Fehler in Bezug auf die statische Semantik enthalten (vgl. Abschnitt II.3.2.2). Die automatische Erkennung dieser Fehler wird im folgenden Abschnitt erläutert.

4.4.2 Merge-bedingte Verletzung der statischen Semantik von Modellen außerhalb der MOF-Syntaxspezifikation

Wenn alle syntaktischen Fehler bzgl. der MOF-spezifischen Syntaxspezifikation in einem Softwaremodell behoben worden sind, lässt sich dieses von einem entsprechenden Modellierungswerkzeug verarbeiten und anzeigen. Mit vielen Modellierungswerkzeugen ist es anschließend möglich, weitere kontextsensitive Korrektheitsbedingungen zu prüfen, die in der formalen Spezifikation der sog. statischen Semantik einer Modellierungssprache festgelegt sind. Allerdings werden hierbei nicht immer alle der durchs Metamodell vorgegebenen Bedingungen der statischen Semantik geprüft. Ein Beispielszenario dafür und die Gründe für dieses Verhalten werden in Abbildung III.20 illustriert.

Als Metamodell dient in diesem Beispiel nicht die in Abschnitt 3.1 eingeführte RE-Sprachspezifikation sondern das UML-Metamodell (Object Management Group (OMG) 2010c) und in diesem der Teil zu Klassendiagrammen.

Viele Modellierungswerkzeuge implementieren für UML-Modelle eine zum Teil umfangreiche Modellvalidierung durch statische Analysen, bei der die im Metamodell spezifizierten OCL-Constraints im Modell geprüft werden. Eine der umfangreichsten Prüfungen bietet die Implementierung des UML2-Projektes (Eclipse Foundation 2010) im Rahmen des Eclipse Modeling Projects an. Diese ist in das Modellierungswerkzeug Topcased (Topcased 2010) integriert und dient der folgenden Veranschaulichung.

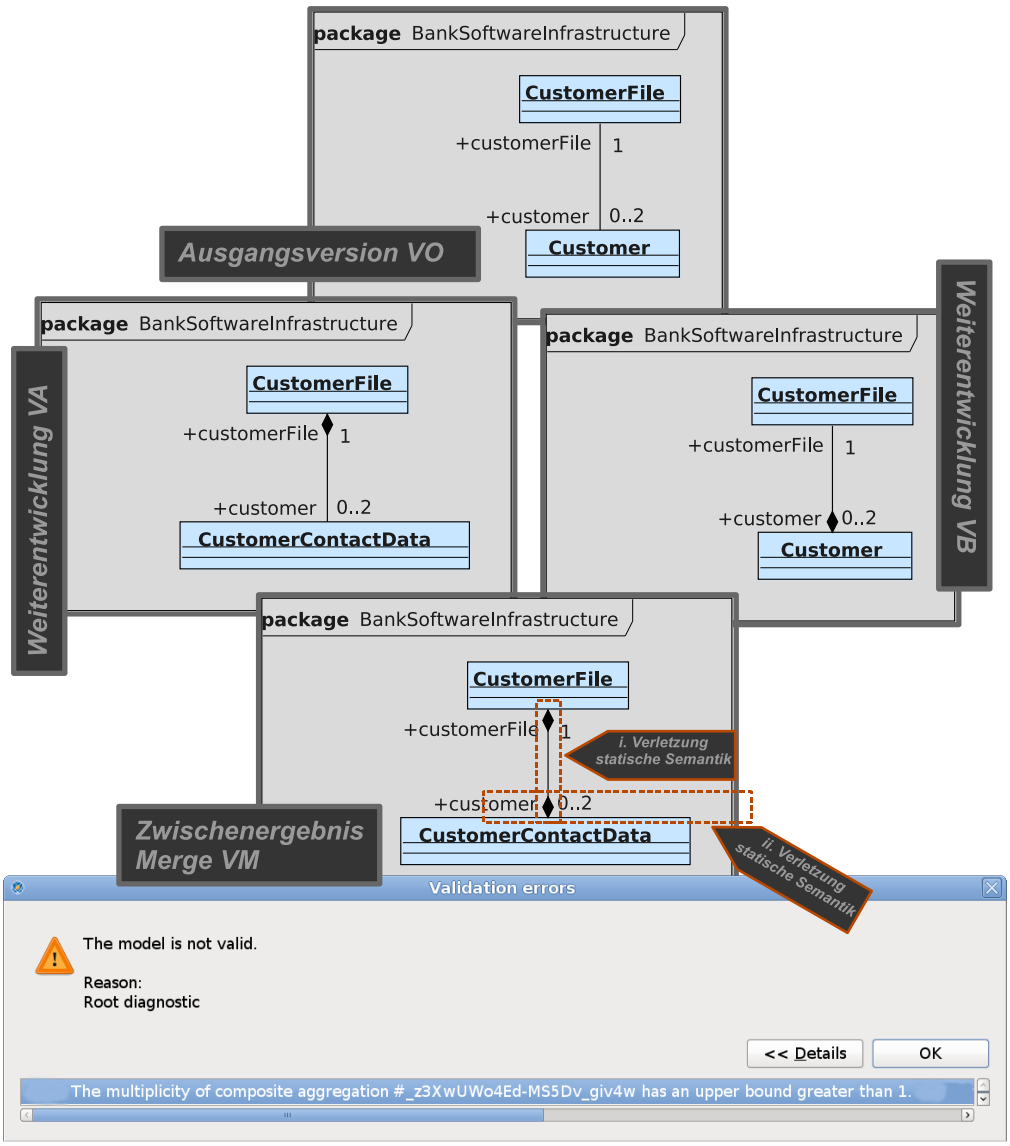


Abbildung III.20. (Nicht-)Erkennung Merge-verursachter Fehler durch Modellierungswerkzeuge

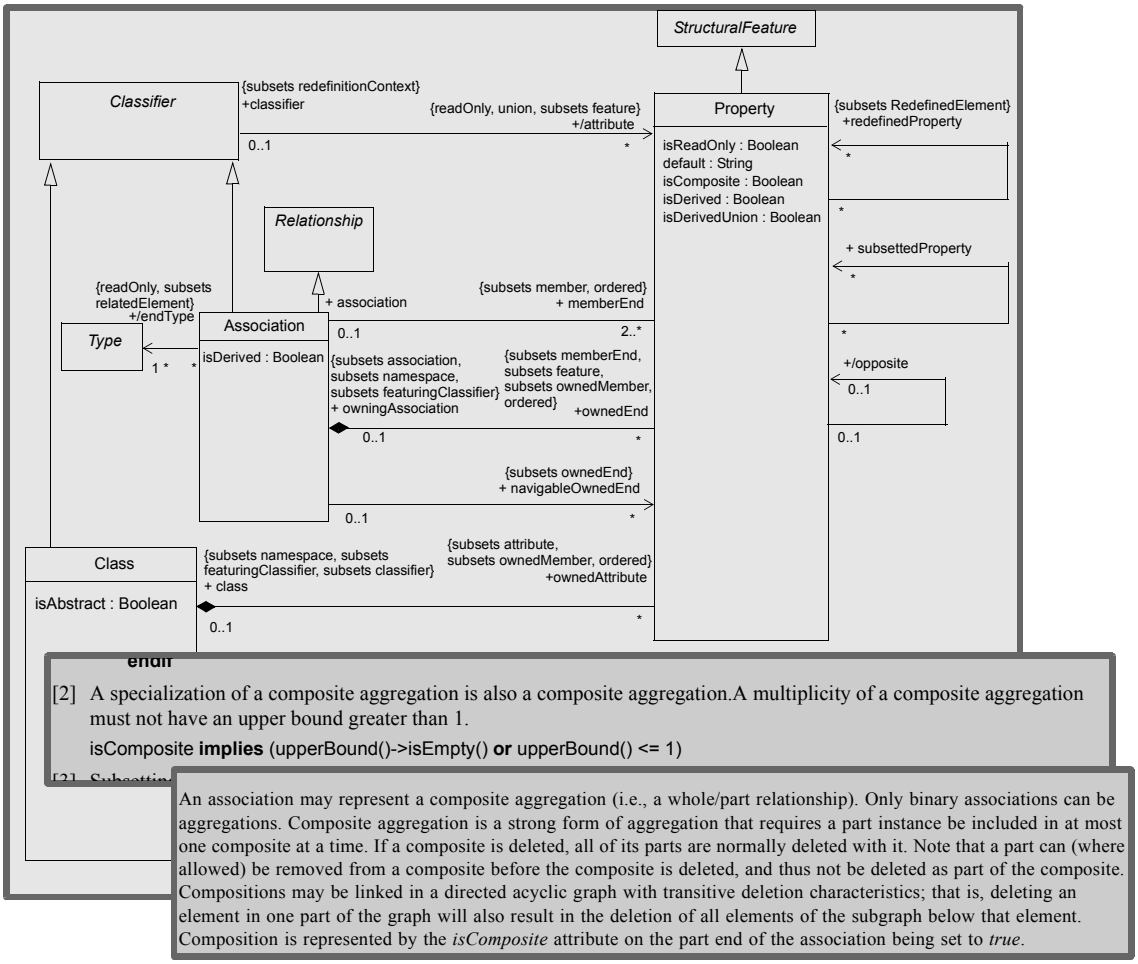


Abbildung III.21. UML-Metamodell - Ausschnitte bzgl. Composite-Eigenschaft
(Quelle : Object Management Group (OMG) 2010c)

Abbildung III.20 stellt ein UML-Modell dar. Es zeigt, wie in den zwei Versionen *VA* und *VB* die Eigenschaft *composite* für beide Enden der Assoziation zwischen *CustomerFile* und *CustomerContactData* auf TRUE gesetzt wurde. Außerdem wurde in der Version *VB* die markierte Multiplizität und der markierte Klassenname geändert. Durch die Integration der logischen Modelländerungen nach dem 3-Wege-Merge-Prinzip werden zwei Bedingungen der statischen Semantik des UML-Metamodells verletzt. Die dabei relevanten Metamodellausschnitte sind Abbildung III.21 dargestellt. Die folgenden Bedingungen werden diesbzgl. verletzt:

i. Die Kompositionsbeziehung ist eine transitive Beziehung. Allerdings darf es keine zyklischen Kompositionsbeziehungen geben. Insofern darf nicht für beide Enden einer Assoziation die *composite*-Eigenschaft auf TRUE gesetzt werden. Andernfalls würde zwischen Objekten, die auf Basis der entsprechenden Assoziation verknüpft sind, eine zyklische Abhängigkeit bestehen.

ii. Ist an einem Assoziationsende die *composite*-Eigenschaft auf TRUE gesetzt, darf die Upper-Bound-Multiplizität an diesem Ende nicht größer als 1 sein. Begründet liegt diese Bedingung in der mit der Kompositionsbeziehung verbundenen Semantik. Diese bestimmt, dass kein Objekt in mehr als einem anderen Objekt kompositional/elementar enthalten sein kann (vgl. (Object Management Group (OMG) 2010c).

Wie in Abbildung III.20 zu sehen, erkennt das Modellierungswerkzeug (im Beispiel: Topcased) ausschließlich den zweiten Fehler. Sofern keine Modelländerungen außerhalb des Modellierungswerkzeugs durchgeführt werden, kommt es bei der nicht-parallelen Entwicklung von UML-Modellen mit Topcased dennoch nicht zu einer Verletzung der Bedingung i.

Der Grund dafür verbirgt sich in der Gestaltung der Benutzungsschnittstelle des in Topcased eingebauten UML-Editors. Dieser verhindert eine inkorrekte Belegung der *composite*-Eigenschaft beim Ändern des Modells. Sofern die *composite*-Eigenschaft für ein Assoziationsende mit dem Editor durch den Modellierer gesetzt wird, setzt der Modelleditor automatisch die *composite*-Eigenschaft des anderen Assoziationsendes zurück. Somit kann bei einer Modellierung ausschließlich per Modelleditor der mit i. markierte Fehler nicht auftreten.

Bei anderen Modellierungswerkzeugen wird die Korrektheit bzgl. Syntax und statischer Semantik ebenfalls schon durch die Benutzungsschnittstelle forciert. Leider ist die Validierung in Modellierungswerkzeugen i. A. nicht auf das Erkennen von Merge-Konflikten ausgerichtet.

Wie das Beispielszenario zeigt, kann ein Fehler der statischen Semantik, der bei der „normalen“ Entwicklungsarbeit eines einzelnen Modellierers durch den Editor verhindert wird, dennoch bei der Integration paralleler Änderungen des Editors auftreten und muss behandelt werden.

In der Regel berücksichtigen Modellierungswerkzeuge bei der Validierung bisher nicht die Synthese von Modellen außerhalb der Anwendung des eigenen Editors und benötigen deswegen nicht immer eine Implementierung aller kontext-sensitiven Bedingungen für die statische Analyse.

Um die Korrektheit im Sinne der Metamodellspezifikation möglichst weitreichend und insbesondere auch nach einem Merge paralleler Versionen zu sichern, müssen auch alle durchs Metamodell formal definierten kontextsensitiven Bedingungen der statischen Semantik automatisch geprüft werden. Erst dadurch werden die Voraussetzungen für eine effiziente Fehlererkennung und Behebung im Rahmen der Integration paralleler Modelländerungen geschaffen.

In Abschnitt IV.2.4 wird ein Verfahren im Detail vorgestellt, das es ermöglicht, Modelle auch hinsichtlich Fehler der statischen Semantik außerhalb bestimmter Modellierungswerkzeuge zu analysieren.

4.4.3 Merge-bedingte Anomalien in Modellen

Auf Seite 78 wurde der Begriff Anomalie, als Bezeichnung für potenziell fehlerhaften Quellcode, eingeführt. Nicht nur in Quellcode können derartige Anomalien bestimmt werden, sondern auch in Softwaremodellen. Ein Beispiel für eine Anomalie in einem Softwaremodell ist in Abbildung III.22 dargestellt. Es zeigt einen Ausschnitt des gemergten Modells aus Abbildung III.10. Schaut man sich die Multiplizitäten zwischen den Komponenten in der Abbildung im Detail an, so wird das Problem in dem Modell der Version *VM* deutlich.

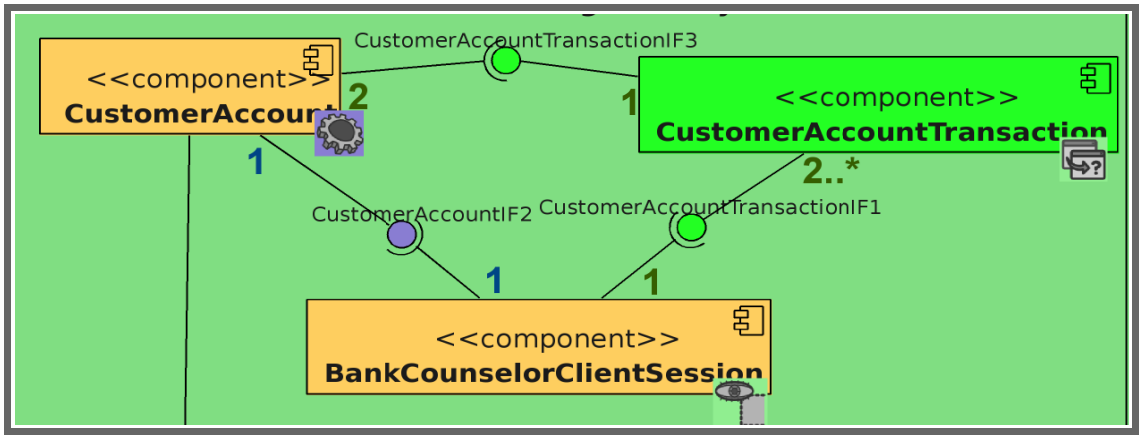


Abbildung III.22. Anomalie in einem Softwaremodell

Nach der Definition des Begriffes Modell in Abschnitt II.3 entspricht die Darstellung in Abbildung III.22 nicht der eines Modells. Der Grund dafür liegt in der Verletzung der Abbildungseigenschaft, die in der Definition für Modelle gefordert wird.

Für die Komponenten, die in Abbildung III.22 dargestellt sind, kann es wegen der Multiplizitätsspezifikation ihrer Nutzungsbeziehungen keine Instanzen geben. Sobald man von der Komponente *BankCounselorClientSession* eine Instanz erzeugen würde, müsste man auch zwei Instanzen von *CustomerAccountTransaction* erzeugen. Diese beiden Instanzen fordern wiederum jeweils die Nutzungsbeziehung mit zwei Instanzen von *CustomerAccount*, sodass von diesen dann mindestens zwei vorhanden sein müssen.

Da aber jede *CustomerAccount*-Komponente mit genau einer *BankCounselorClientService*-Komponente verknüpft sein muss, aber nur eine vorhanden ist, führt dies zum Widerspruch bzgl. der Erfüllung der Multiplizitätsbedingungen. Bei genauerer Betrachtung erkennt man, dass sobald eine Instanz einer beliebigen der drei Komponenten erstellt wird, ein Widerspruch auftritt. Die Modellelemente für alle dargestellten Komponenten können somit keine Entsprechung (Instanz) in der realen Welt (MOF-Ebene M0) haben, was dem formulierten Abbildungskriterium von Modellen zuwiderläuft.

Es ist leicht nachvollziehbar das derartige Anomalien in Modellen leicht bei der parallelen Bearbeitung (hier: bspw. bei parallelen Hinzufügung oder Änderung von Multiplizitäten durch Alice und Bob) auftreten können. Wie bei der Quellcodeanalyse hinsichtlich entsprechender Anomalien ist eine automatische Modellanalyse im Anschluss an einen automatischen Merge hilfreich, um Fehler bei der inhaltlichen Synchronisierung paralleler Arbeit zu finden.

4.5 Fazit: Die werkzeuggestützte Versionsverwaltung von Modellen erfordert neue Lösungsansätze

In den letzten Abschnitten wurde erläutert, dass aktuell verfügbare Ansätze zur werkzeugunterstützten Versionsverwaltung ernsthafte Schwächen bei der Verwaltung von Softwaremodellen während ihrer kollaborativen Entwicklung aufweisen. Bei der vielfach geforderten optimistischen Versionsverwaltung von Modellen verhindern diese Schwächen den praktikablen Einsatz automatischer Softwarewerkzeuge komplett.

Ursache dieser Schwächen ist das durch etablierte Ansätze implementierte Text-Merge-Verfahren zur Integration paralleler Änderungen und die unzureichenden Analysemöglichkeiten Merge-bedingter Konflikte in Modellen. Zur Konstruktion einer adäquaten Versionsverwaltung von Softwaremodellen wurden Lösungsansätze skizziert, die sich durch folgende Punkte zusammenfassen lassen:

- Verwendung eines an der formalen Syntax orientierten Merge-Verfahrens (morphologischer Merge) statt eines Text-Merge-Verfahrens zur Integration paralleler Modelländerungen
- Unterstützung der Integration logischer Modelländerungen durch eine universelle[1] statisch Analyse zur Validierung der syntaktischen Korrektheit von Modellen und weiterer Anomalien.

Ferner wurde angesprochen, dass domänenspezifische Modellierungssprachen eine immer größere Verbreitung neben zahlreichen etablierten Modellierungssprachen erfahren. Um eine breite Anwendbarkeit der zu entwickelnden Modellverwaltung garantieren zu können, sollten entsprechende Lösungsansätze von der Spezifikation einer bestimmten Modellierungssprache abstrahieren und somit Modelle beliebiger Sprachen verwalten können. Im folgenden Kapitel wird für die formulierten Herausforderungen eine Lösung konzipiert.

1 Universell heißt die Korrektheitsanalyse in diesem Kontext, wenn sie für Metamodelle beliebiger Modellierungssprachen anwendbar ist und unabhängig ist von bestimmten Modellierungswerkzeugen.

IV Korrekte Integration asynchroner Modelländerungen

Trotz hoher Nachfrage konnte eine Werkzeugunterstützung zur optimistischen Versionsverwaltung von Softwaremodellen bisher nicht realisiert werden. Im letzten Kapitel wurden dazu die Ursachen für das Scheitern etablierter Implementierungen zur Versionsverwaltung in der kollaborativen Modellierung erläutert. Zudem wurde eine Lösung skizziert, die durch eine automatisierte Versionsverwaltung die asynchrone Entwicklung von Softwaremodellen im Team ermöglicht.

Dabei gliedert sich die Skizze in zwei zu lösende Teilproblemstellungen – Integration von Modelländerungen durch ein morphologisches Merge-Verfahren und anschließende Merge-bedingte Validierung integrierter Modellversionen durch statische Analyse ihrer formalen Syntax. Beide werden in diesem Kapitel im Detail ausgearbeitet und bilden die Grundlage für die Realisierung einer Werkzeugunterstützung, welche in Kapitel V beschrieben wird.

Wie im letzten Kapitel beschrieben, ist das Scheitern des Merge-Verfahrens der Haupthinderungsgrund für den Einsatz etablierter optimistischer Versionsverwaltungswerkzeuge für Modelle. Aus Anwendersicht sind dabei sehr ähnliche Herausforderungen zu bewältigen, wie sie bei der optimistischen Versionsverwaltung von Quellcode auftreten.

Trotz der Ähnlichkeit im grundlegenden Ansatz ist die konkrete Umsetzung der Merge-Phasen aus der Quellcodeverwaltung nicht ohne Weiteres auf die Modellverwaltung übertragbar. Bei der notwendigen Anpassung müssen Teile des Verfahrens, wie bspw. der Änderungsvergleich von Versionen, komplett neu entworfen werden.

Die inhaltliche Gliederung von Kapitel IV orientiert sich an den grundlegenden Merge-Phasen aus der Anwendersicht eines Software-Engineering-Teams. Dabei werden schwerpunktmäßig die Teilkonzepte des Modell-Merge-Verfahrens erläutert, die sich vom Quellcode-Merge-Verfahren wesentlich unterscheiden. Die Umsetzung der Integration asynchron und parallel durchgeführter Modelländerungen muss hierbei grundlegend anders gestaltet werden als die Integration entsprechender Codeänderungen. Das dafür im letzten Kapitel motivierte morphologische Merge-Verfahren für Modelle wird im folgenden Abschnitt

eingehend erörtert. Der daran anschließende Abschnitt widmet sich der Merge-bedingten Validierung von Modellversionen, die auch im Anschluss an ein morphologisches Merge-Verfahren erforderlich ist.

1 Ein morphologisches Merge-Verfahren für Modelle

Das auf Seite 46 beschriebene 3-Wege-Merge-Prinzip hat sich in der werkzeuggestützten Quellcodeverwaltung weitgehend durchgesetzt. Es ist intuitiv verständlich und zielt auf die Anwenderanforderung nach einer optimistischen Versionsverwaltung für Modelle. Aus diesem Grund wird es ebenso wie beim Merge von Quellcode als Kernkonzept des morphologischen Merge-Verfahrens angewendet. Dazu müssen durch das Verfahren zwei Herausforderungen gelöst werden:

- Identifizierung generischer Änderungen zwischen Modellversionen
- Integration paralleler Modelländerungen zu einer Modellversion

Um analog zur Quellcodeverwaltung eine weitgehende Werkzeugunterstützung bieten zu können, müssen beide Schritte so konzipiert werden, dass sie vollständig automatisierbar sind. In den beiden folgenden Abschnitten werden Lösungskonzepte beschrieben, die das morphologische Merge-Verfahren ausmachen.

1.1 Identifizierung generischer Änderungen zwischen Modellversionen

Zur Anwendung des 3-Wege-Merge-Konzeptes auf zwei Modellversionen *VA* und *VB*, die durch parallele Weiterentwicklungen einer gemeinsamen Vorgängerversion *VO* entstanden sind, müssen zunächst die Änderungen zwischen *VO* und *VA* bzw. zwischen *VO* und *VB* bestimmt werden. Dies wird durch einen automatischen Vergleich der betreffenden Versionen erreicht.

Hierbei kommt ein zustandsbasierter Änderungsvergleich zum Einsatz, da dieser im Gegensatz zu operationsbasierten Ansätzen weitgehend unabhängig von den für die Modellbearbeitung genutzten Werkzeugen ist (vgl. II.2.2.3).

Wie in Abschnitt II.2 erörtert wurde, werden beim zustandsbasierten Änderungsvergleich Dokumentversionen auf der Ebene einzelner Dateneinheiten miteinander verglichen. Wie in Abschnitt II.2.2.7 beschrieben, werden für den Vergleich von Quellcode die Textzeilen der entsprechenden Dateien analysiert. Für Modelle kann dieses Verfahren nicht erfolgreich eingesetzt werden. Die Gründe dafür sind in III.4.1 ausführlich dargelegt.

In Abschnitt III.4.2 und III.4.3 wird allerdings eine Alternative zur Betrachtung von Textzeilen im Rahmen des Änderungsvergleiches skizziert. Dabei wird die Verarbeitung logischer Modelländerungen vorgeschlagen. Zur Identifizierung dieser Änderungen müssen die logischen Informationen einzelner Modellversionen miteinander verglichen werden. Durch die Betrachtung von sog. morphologischen Dateneinheiten wird dabei eine möglichst feingranulare Betrachtung von logischen Modelländerungen ermöglicht.

Zur Automatisierung des Versionsvergleiches müssen die entsprechenden Modelldateien in morphologische Dateneinheiten zerlegt werden. Eine derart intuitive Zerlegung von Modellen

wurde in Abschnitt III.4.3 schon beschrieben. Im nächsten Abschnitt wird nun erläutert, wie eine solche Zerlegung automatisiert werden kann.

1.1.1 Morphologische Zerlegung von Modelldaten

Morphologische Dateneinheiten (i. F. auch MDU – engl. morphological data unit) wurden zur Repräsentation atomarer Modellinformationen sog. Morphe eingeführt. Lässt sich nun jedes beliebige Modell in solche Einheiten zerlegen? Wie anschließend gezeigt wird, lautet die Antwort auf diese Frage: Ja, wenn alle zu verwaltenden Modellinformationen in einer standardisierten maschinenverarbeitbaren Repräsentation vorliegen und diese unabhängig von einer bestimmten Modellierungssprache ist.

Im Software Engineering hat sich dafür XMI (Object Management Group (OMG) 2010a) als universeller Repräsentationsstandard zur Speicherung von Modellinformationen bei fast allen Anbietern von Modellierungssoftware durchgesetzt. Außerdem lässt sich die XMI-Syntax jedes Modells in eine Menge von MDUs transformieren.

Ist eine Modellversion derart transformiert, dann wird von einer morphologischen Modellrepräsentation gesprochen. Die Umwandlung von Modelldateien in diese Repräsentationsform ist notwendig, um für das morphologische Merge-Verfahren den 3-Wege-Änderungsvergleich anzuwenden. Die dabei identifizierten logischen Modelländerungen bilden die Grundlage für die Anwendung eines 3-Wege-Merge-Verfahrens für Modellversionen. Abbildung IV.1 stellt das beschriebene Verfahren grafisch dar. Als Alternative zum XMI-Parser/Lexer bei der Verarbeitung von EMF-Modellen (vgl. Abbildung II.12) wird i. F. ein Verfahren vorgestellt, das vergleichbar mit der Funktionalität eines Lexers für XMI-Dateien ist. Dieses Verfahren zerlegt die in einer XMI-Datei gespeicherten Modellinformationen so, dass bei der Synchronisation paralleler Modelländerungen beliebig dekomponierte syntaktische Einheiten betrachtet werden können.

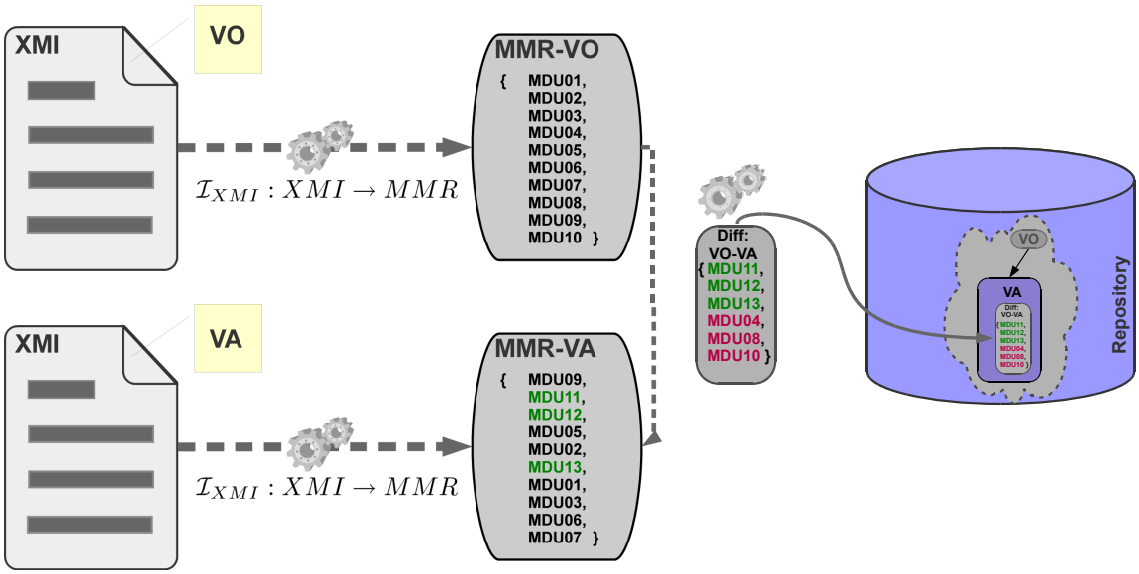


Abbildung IV.1. Automatische Erzeugung einer morphologischen Modellrepräsentation (MMR) aus XMI-Modelldateien

Die Abbildung stellt die beiden Modellversionen VO und VA dar, wie sie auf dem Rechner als XMI-Dateien gespeichert sind. Dabei ist VA eine Weiterentwicklung der Modellversion VO. Um den für einen zustandsbasierten 3-Wege-Merge notwendigen Änderungsvergleich durchführen zu können, wird die XMI-Repräsentation beider Versionen zunächst automatisch in zwei Mengen von MDUs als morphologische Modellrepräsentationen durch $\mathcal{I}_{XMI}$ transformiert. Dieser Vorgang entspricht dem Vorgehen eines Lexers/Parsers (vgl. II.3.2.2).

Bevor die Transformation von XMI-Dateien in MDUs beschrieben wird, soll zunächst erörtert werden, wie eine morphologische Modellrepräsentation (i. F. auch MMR genannt) aufgebaut ist.

1.1.2 Schema der morphologischen Modellrepräsentation

Das Schema der morphologischen Modellrepräsentation (i. F. auch MMR-Schema genannt) stellt genau vier Typen von MDUs zur Verfügung. Diese sind in Abbildung IV.2 dargestellt und werden bezeichnet mit *TypeAssertion*, *ObjectPropertyAssertion*, *DataPropertyAssertion* und *PropertyOrder*. Alle Modelle, die, wie im modellbasierten Software Engineering üblich, nach einem modellierungssprachspezifischen Metamodell aufgebaut sein müssen, lassen sich als eine Menge dieser MDUs ausdrücken.

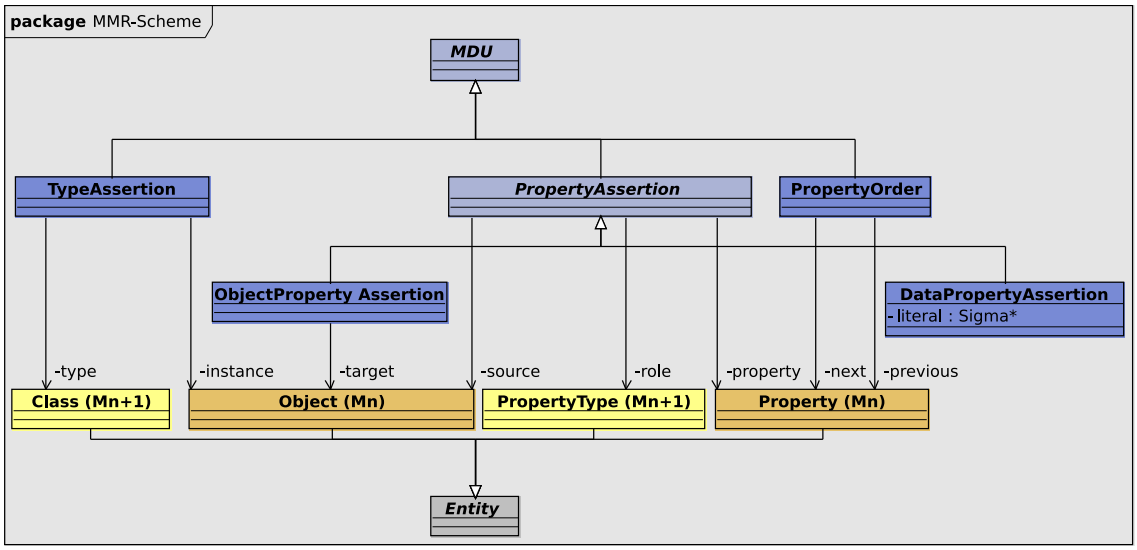


Abbildung IV.2. Schema der morphologischen Modellrepräsentation (MMR)

Die Repräsentation beliebiger Modelle auf Basis des Schemas in Abbildung IV.2 soll anschließend im Detail erläutert werden. Die durch die Syntax beschriebenen Bestandteile eines Modells werden i. F. als Entitäten bezeichnet, welche sich wiederum in Objekte und Eigenschaftsausprägungen unterscheiden lassen. Dazu seien folgende Definitionen gegeben:

Definition IV.1 (Entität, Original Modell)

Eine Entität (engl. *entity*) $e \in ENTITY$ ist ein Teil der Wirklichkeit, der als Einheit wahrgenommen wird und benannt werden kann. Die Menge $ENTITY$ wird als Universum aller Entitäten eingeführt. Entitäten können aus anderen Entitäten zusammengesetzt sein. Die nicht-totale Abbildung $\mathcal{I}_{Composite} : ENTITY \rightarrow \wp(ENTITY)$ weist jeder Entität, für die eine Zerlegung in Teile möglich ist, die Menge an Entitäten zu, aus der sie zusammengesetzt ist.

Mit Original $Orig \subseteq ENTITY$ wird eine Menge von Entitäten bezeichnet.

Seien $Orig_1 \subseteq ENTITY$ und $Orig_2 \subseteq ENTITY$ zwei Originale mit $Orig_1$ endlich und sei $\mathcal{I} : \wp(ENTITY) \rightarrow \wp(ENTITY)$ mit $\mathcal{I}(Orig_1) \subsetneq Orig_2 \rightarrow \mathcal{I}(Orig_1) \neq \emptyset$ eine Interpretationsabbildung (vgl. Verkürzungsmerkmal Stachowiak 1973), dann ist $Orig_1$ im Sinne der Interpretation $\mathcal{I}$ ein Modell für $Orig_2$.

Zur syntaktischen Spezifikation von Modellen im Software Engineering werden wie bereits erwähnt Metamodelle eingesetzt. Diese beschreiben eine Menge korrekter abstrakter Syntaxgraphen von Modellen.

Ein Metamodell ist ein Modell für eine Menge von Modellen, die auf dessen Basis erstellt werden können; es wird folgendermaßen definiert:

Definition IV.2 (Metamodell)

Seien $\hat{m} \subseteq ENTITY$ und $m \subseteq ENTITY$ zwei Modelle und sei $\mathcal{I}$ eine Interpretationsabbildung für $\hat{m}$. Dann ist m ein Modell des Metamodells $\hat{m}$, wenn gilt:

$$\exists e_m \in \mathcal{I}(\hat{m}) : \mathcal{I}_{Composite}(e_m) = m$$

Die Entitäten jedes Modells lassen sich in Objekte und Eigenschaften unterteilen. Das Metamodell eines Modells spezifiziert dabei mit seinen Objekten und Eigenschaften die Klassen und Eigenschaftstypen für die Entitäten des Modells:

Definition IV.3 (Objekt, Eigenschaft)

Sei $m \subseteq ENTITY$ ein Modell. Die Entitäten von m lassen sich in Objekte und Eigenschaften unterscheiden:

Ein Objekt (engl. *object*) $o \in Object^m \subseteq m$ ist eine perzeptorisch identifizierbare Entität. Dieser werden konkrete Eigenschaften beigemessen. Durch die Festlegung der Objekte in einem Modell wird die Frage beantwortet: Was existiert im Modell?

Eine Eigenschaft (engl. *property*) $p \in Property^m \subseteq m$ eines Objektes o charakterisiert dessen Gestalt. Durch die Festlegung von Eigenschaften für ein Objekt wird die Frage beantwortet: Wie unterscheiden sich Objekte voneinander und wie stehen Objekte mit anderen Objekten in Beziehung?

Es gilt: $Object^m \dot{\cup} Property^m = m$.

Definition IV.4 (Klasse, Eigenschaftstyp, Typ, Instanz, Rolle)

Sei $m \subseteq ENTITY$ ein Modell und sei $\hat{m} \subseteq ENTITY$ ein Metamodell für m.

Die Objekte des Metamodells $\hat{m}$ lassen sich Bezug nehmend zum Modell m in Klassen und Eigenschaftstypen aufteilen: $Object^{\hat{m}} = Class_m^{\hat{m}} \dot{\cup} PropertyType_m^{\hat{m}}$.

Eine *Klasse* (engl. *class*) legt fest, in welcher Art und Weise die Eigenschaften eines Objektes von m ausgeprägt sein müssen, damit dieses als *Instanz* (engl. *instance*) von o bezeichnet werden darf. Eine Klasse c selbst wird dabei als *Typ* (engl. *type*) des jeweiligen Objektes aus m bezeichnet.

Ein *Eigenschaftstyp* (engl. property type) legt fest, für welche Eigenschaften eines Objektes im Modell m eine Ausprägung bestimmt werden kann oder muss und wie der Wertebereich dieser Ausprägung definiert ist. Der im Metamodell spezifizierte Eigenschaftstyp einer Eigenschaft wird als deren *Rolle* (engl. *role*) bezeichnet.

Um die automatische Verarbeitung von Modellinformationen zu ermöglichen, müssen Objekte und Eigenschaften als maschinell verarbeitbare Daten repräsentiert sein. Der Standard XMI, bietet die Möglichkeit, Modelle beliebiger Modellierungssprachen als XMI-Dateien zu speichern.

Das Datenformat XMI zu Speicherung von Modellversionen bildet für die weiteren Erläuterungen die Grundlage zur Erstellung einer morphologischen Modellrepräsentation. Grund ist die weite Verbreitung des XMI-Datenformates bei den Herstellern von Modellierungssoftware. Der vorgestellte Ansatz zur Zerlegung von Modellinformationen ließe sich aber auch auf andere Datenformate zur Modellspeicherung übertragen.

Abbildung IV.3 zeigt die Repräsentation von Objekten und Eigenschaften im XMI-Format. Wie in Abbildung IV.3 zu sehen, existieren drei Alternativen zur Ausprägung der Objekteigenschaften:

i. Zuweisung eines Typs (engl. *type assertion*)

ii. Zuweisung anderer Objekte (engl. *object property assertion*)

iii. Zuweisung von Zeichenfolgen (engl. *data property assertion*)

Jede konkrete Zuweisung einer dieser drei Arten entspricht einer Information, die inhaltlich nicht mehr in Teilinformationen zerlegt werden kann. Besteht für ein Objekt die Ausprägung einer Eigenschaft aus einer geordneten Menge einzelner Eigenschaftsausprägungen desselben Eigenschaftstyps, so ergeben sich aus der Ordnung der Listenelemente weitere zu speichernde Informationen:

iv. Konkrete Ordnungsrelation zwischen Eigenschaftsausprägungen

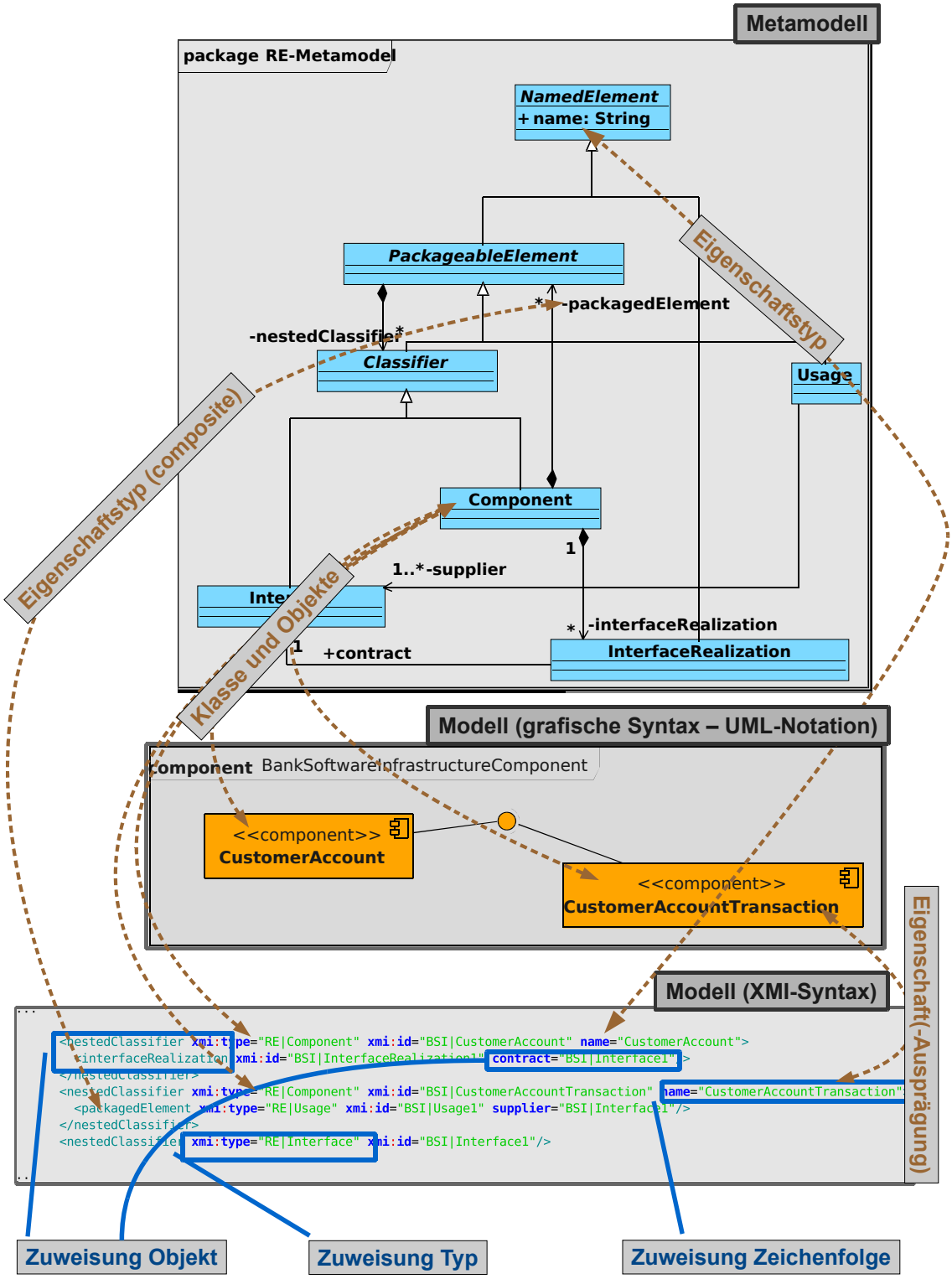


Abbildung IV.3. Definition IV.1 bis Definition IV.4 am Beispiel

Die explizite Repräsentation der genannten Zuweisungen und Ordnungsrelationen als Datensätze auf Basis des Schemas in Abbildung IV.2 entspricht den anfangs motivierten morphologischen Dateneinheiten (MDU). Die i. F. definierte morphologische Modellrepräsentation besteht aus einer Menge der oben benannten MDUs.

Definition IV.5 (Morphologische Modellrepräsentation)

Sei m_{Data} ein Modell repräsentiert im Datenformat $Data$ und sei $Data \subseteq \wp(ENTITY)$ die Menge aller Modelle, die sich in dem gleichnamigen Datenformat erstellen lassen.

Eine morphologische Modellrepräsentation $mmr^m \in MMR \subseteq \wp(ENTITY)$ des Modells m_{Data} ist ein Modell, dessen Objekte sich in Typzuweisungen, Eigenschaftszuweisungen und Ordnungsrelationen für Eigenschaftsausprägungen unterteilen lassen. Es gilt somit:
$TypeAssertion^{m_{Data}} \dot{\cup} PropertyAssertion^{m_{Data}} \dot{\cup} PropertyOrder^{m_{Data}} = mmr^{m_{Data}}$

Ferner muss eine totale Interpretationsabbildung $\mathcal{I}_{Data} : Data \rightarrow MMR$ existierten, die jedes Modell, das im Format $Data$ repräsentiert ist, eindeutig in ein Modell in MMR transformiert.

Durch Zuweisung eines Typs werden einerseits bestimmte Eigenschaften für ein Modellobjekt festgelegt und andererseits wird die Ausprägung weiterer Eigenschaften des Modellobjektes eingeschränkt. Die Entität, die diese Charakteristika definiert, ist ein Objekt des Metamodells, das mit der Klasse des Modellobjekts bezeichnet wird. Dazu sei folgende Definition gegeben:

Definition IV.6 (Typzuweisung)

Sei $o \in m_{Data}$ ein Objekt eines Modells m_{Data} im Format $Data$ und sei $c \in \hat{m}_{Data}$ die Klasse von o im Metamodell. Sei ferner $id_{Data} : \left(\bigcup_{m \in Data} m \right) \rightarrow \Sigma_{ID}^*$ eine bijektive Abbildung, die jeder Entität eines Modells im Format $Data$ einen Identifizier zuweist, wobei Identifizierer aus einer Zeichenfolge auf Basis eines Alphabetes Σ_{ID} bestehen.

Eine Typzuweisung (engl. *type assertion*) für ein Objekt o ist eine morphologische Dateneinheit, die dem Tupel $(\overbrace{id_{Data}(c)}^{\text{type}}, \overbrace{id_{Data}(o)}^{\text{instance}}) \in mmr^m$ mit $mmr^m \in MMR$ entspricht.

Analog zur Charakterisierung eines Objektes durch einen Typ, spezifiziert ein Eigenschaftstyp eine Eigenschaft genauer. Darüber hinaus verknüpft eine Eigenschaft zwei Objekte miteinander. Das ist zum einen das Objekt, das die Eigenschaft ausprägt (bez. mit *source*) und zum anderen das Objekt, das die Ausprägung bestimmt (bez. mit *target*). Die Ausprägung einer Eigenschaft kann dabei entweder einem anderen Objekt des Modells oder einem Wert eines Basistyps (Boolean, Integer, String etc.) entsprechen. Zeichenfolgen von Basistypen werden mit Literal bezeichnet:

Definition IV.7 (Literal)

Ein Literal $l \in \Sigma^*_{Literal} \subset ENTITY$ ist eine endliche Zeichenfolge bestehend aus Zeichen des Alphabets $\Sigma_{Literal}$. $\Sigma^*_{Literal} \subsetneq \Sigma^*_{ID}$ ist die Menge aller Literale. Beispiele für Literale sind Wahrheitswerte, Ganzzahlen und Zeichenketten: $Boolean, Integer, String \subseteq \Sigma^*_{Literal}$

Für Literale gilt i. F.: $l \in \Sigma_{Literal} \leftrightarrow id_{Data}(l) = l$

Die Zuweisung eines Eigenschaftstyps zu einer Eigenschaft und die Verknüpfung von Objekten im Rahmen der MMR wird durch folgende Definition formal eingeführt.

Definition IV.8 (Eigenschaftszuweisung)

Seien $o_1, o_2 \in m$ zwei Objekte eines Modells m, sei $p \in m$ eine Eigenschaft des Objektes o_1, deren Ausprägung durch o_2 bestimmt wird. Sei ferner $pt \in \hat{m}$ der Eigenschaftstyp im Metamodell von m der die Rolle von p bestimmt.

Eine Eigenschaftszuweisung (engl. *property assertion*) für ein Objekt o_1 ist eine morphologische Dateneinheit, die dem Tupel
$(\overbrace{id_{Data}(p)}^{\text{property}}, \overbrace{id_{Data}(pt)}^{\text{role}}, \overbrace{id_{Data}(o_1)}^{\text{source}}, \overbrace{id_{Data}(o_2)}^{\text{target}}) \in mmr^m$ mit $mmr^m \in MMR$ entspricht.

Handelt es sich bei o_2 um ein Literal ($o_2 \in \Sigma^*_{Literal}$), so wird die Eigenschaftszuweisung mit *data property assertion* bezeichnet ansonsten mit *object property assertion*. Es gilt:
$ObjectPropertyAssertion^m \dot{\cup} DataPropertyAssertion^m = PropertyAssertion^m$

Zur Darstellung des in Abbildung IV.3 beschriebenen Beispielmodells lässt sich eine MMR mit den eingeführten Typ- und Eigenschaftszuweisungen erzeugen. Diese ist in Abbildung IV.4 illustriert:

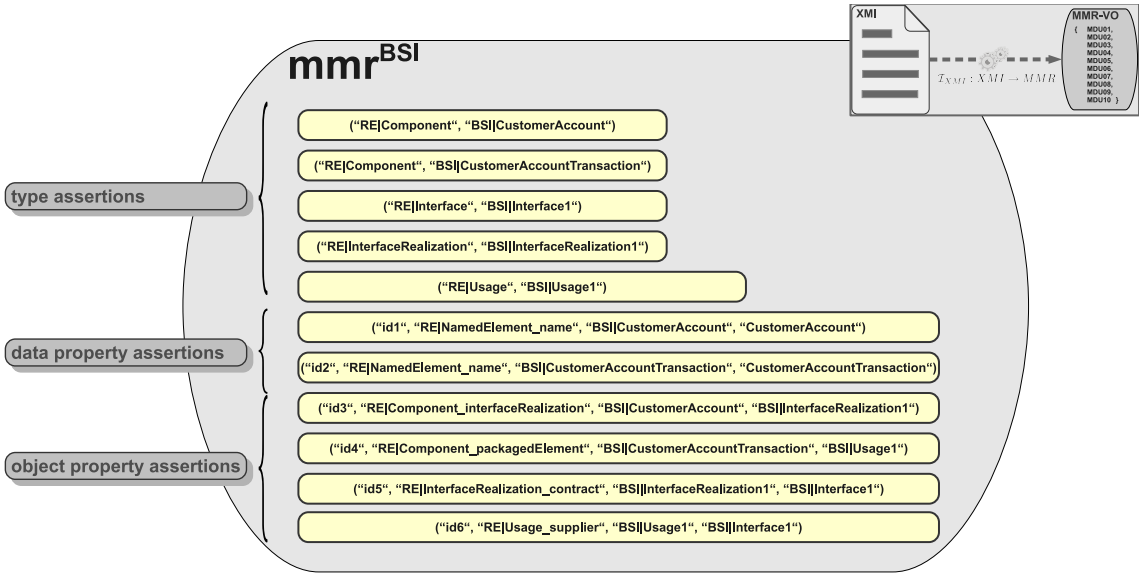


Abbildung IV.4. Beispielmodell aus Abbildung IV.3 in MMR dargestellt

Oft ist eine Eigenschaft nicht einzeln sondern als Liste von Objekten ausgeprägt (Multiplizität des Eigenschaftstyps ist größer als 1). Wenn darüber hinaus der Eigenschaftstyp festlegt, dass die Ordnung der Eigenschaftsausprägungen für die Interpretation des Modells ausschlaggebend ist (Meta-Attribut: „ordered=True“ vgl. Abbildung II.15), dann müssen die konkreten Ordnungsbeziehungen zwischen den Eigenschaftsausprägungen ebenfalls als Informationen durch die MMR festgehalten werden. Grundlage dafür sind MDUs vom Typ PropertyOrder:

Definition IV.9 (Ordnungsrelation für Eigenschaftsausprägungen)

Seien $p1, p2 \in m_{Data}$ zwei Ausprägungen ein und des selben Eigenschaftstyps, wobei die Reihenfolge der Ausprägungen für die Interpretation des Modells relevant ist.

Eine konkrete Ordnungsrelation für Eigenschaftszuweisungen (engl. *property order*) ist eine morphologische Dateneinheit, die dem Tupel $(\overbrace{id_{Data}(p_1)}^{previous}, \overbrace{id_{Data}(p_2)}^{next}) \in mmr^m$ entspricht.

Um die Verwendung dieser MDUs genauer zu beschreiben, wird die im Modell in Abbildung III.7 spezifizierte Drei-Schichten-Architektur als MMR in der folgenden Abbildung IV.5 dargestellt. In der MMR-Darstellung unten links sieht man die beiden *PropertyOrder*-MDUs, mit denen die Ordnung der Schichten innerhalb des Musters festgehalten wird.

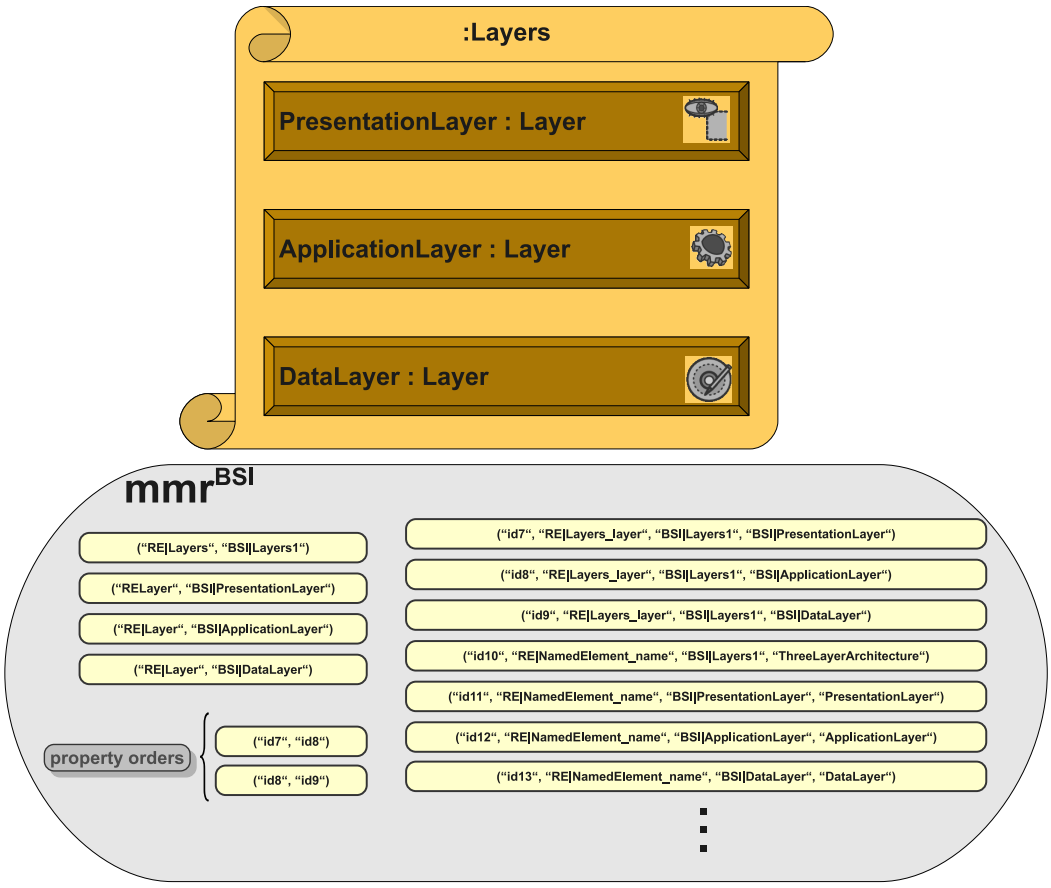


Abbildung IV.5. Spezifizierte Schichtenarchitektur als MMR ausgedrückt – Ausschnitt

1.1.3 Transformation von Modelldaten im XMI-Format in eine morphologische Modellrepräsentation

Im letzten Abschnitt wurde das Schema für die MMR eingeführt. Zur Verwaltung von Modellversionen muss, wie in Definition IV.5 verlangt, eine Abbildung vom Modelldatenformat (hier: nur XMI betrachtet) in die MMR vorhanden sein. Diese Abbildung wird i. F. in drei logischen Schritten informell beschrieben. Auf eine ausführlichere formale Beschreibung wird an dieser Stelle verzichtet, da die beschriebene Abbildung leicht nachvollziehbar ist. Sie wurde im Softwarewerkzeug Team.Mode (vgl. Kapitel V) implementiert.

Eingabe: XMI-Modelldatei **Ausgabe:** Modellrepräsentation $mmr^{BSI} \in MMR$

Schritt 1 (Erzeugung der type assertions):

Erzeuge für jedes XML-Tag der XMI-Datei eine type assertion in der mmr^{BSI}. Dabei sind die IDs des Tupels folgendermaßen zu belegen:

- ID für Instanz der MDU: Belegung des Attributs „xmi:id“ (vgl. Abbildung IV.3) des XML-Tags.
- ID für Typ der MDU: Belegung des Attributs “xmi:type“ des XML-Tags .

Beispiel: Erzeuge morphologische Dateneinheit $mdu = (\text{"RE|Usage"}, \text{"BSI|Usage1"})$ aus „`<packagedElement xmi:type="RE|Usage" xmi:id="BSI|Usage1" supplier="BSI|Interface1"/>`“ (Abbildung IV.3, Zeile 5)

Schritt 2 (Erzeugung der property assertions):

Erzeuge property assertion für jedes XML-Tag. Dabei sind die IDs des Tupels folgendermaßen zu belegen:

- ID für property der MDU wird eineindeutig aus den Belegungen für role, source und target generiert
- ID für role der MDU:
  - Belegung des Attributs „xmi:type“ des Vater-Tags (bspw. „RE:Component“) und „_“,
  - Tag-Name (bspw. „interfaceRealization“)
- ID für source der MDU:
  - XMI-ID des Vater-Tags
- ID bzw. Literal für target der MDU:
  - XMI-ID des Tags bzw. Wert

Beispiel: Erzeuge morphologische Dateneinheit
$mdu = (\text{"id3"}, \text{"RE|Component_interfaceRealization"}, \text{"BSI|CustomerAccount"}, \text{"BSI|InterfaceRealization1"})$
aus

```
<nestedClassifier xmi:type="RE|Component" xmi:id="BSI|CustomerAccount" name="CustomerAccount">
  <interfaceRealization xmi:id="BSI|InterfaceRealization1" contract="BSI|Interface1"/>
```

(Abbildung IV.3, Zeile 1 und 2)

(Fortsetzung)

Erzeuge property assertion für jedes Attribut, das in einem XML-Tag definiert wird und nicht im Namensraum „xmi“ liegt. Dabei sind die IDs des Tupels folgendermaßen zu belegen:

- ID für property der MDU wird eineindeutig aus den Belegungen für role, source und target generiert.
- ID für role der MDU:
  - Belegung des Attributs „xmi:type“ des Tags (bspw. „RE|Usage“) und „_“,
  - Attributname (bspw. „supplier“)
- ID für source der MDU:
  - XMI-ID des Tags (bspw. „BSI|Usage1“)
- ID bzw. Literal für target der MDU:
  - Attributbelegung (bspw. „BSI|Interface1“)

Beispiel: Erzeuge morphologische Dateneinheit
$mdu = (\text{”id6”}, \text{”RE|Usage_supplier”}, \text{”BSI|Usage1”}, \text{”BSI|Interface1”})$ aus
„<packagedElement xmi:type="RE|Usage" xmi:id="BSI|Usage1" supplier="BSI|Interface1"/>“
(Abbildung IV.3, Zeile 5)

Schritt 3 (Erzeugung der property orders):

Erzeuge eine Menge von property orders für jedes Attribut eines XML-Tags, das mit einer Liste von Werten belegt ist. Dabei erzeuge für jeweils zwei aufeinanderfolgende Werte der Liste eine property order dessen Tupel-IDs folgendermaßen zu belegen sind:

- ID für previous der MDU:
  - Vorangehende Wertbelegung eines Wertepaares
- ID für target der MDU:
  - Nachfolgende Wertbelegung eines Wertepaares

Die oben dargestellte Abbildung stellt eine Abbildung zwischen dem XMI-Modelldatenformat und der MMR dar. Mit der Abbildung gelingt es nun Modelldaten automatisch in die MMR zu transformieren, um sie dann im Repository verwalten zu können. Außerdem kann die MMR einer Modellversion aus dem Repository wieder zurück das XMI-Format transformiert werden, sofern die durch Metamodell festgeschriebene Modellsyntax hinreichend korrekt ist. Nach einer Rücktransformation kann die Modellbearbeitung wieder mit einschlägiger Modellierungssoftware durchgeführt werden.

Um einen 3-Wege-Merge von Modellversionen durchführen zu können, müssen die Änderungen zwischen aufeinanderfolgenden Modellversionen in der Versionshistorie bekannt sein. Diese werden mithilfe eines statischen Verfahrens (2-Wege-Diff zwischen Modellversionen) identifiziert. Das Vergleichsverfahren wird dabei auf der MMR der zu vergleichenden Modellversionen durchgeführt. Dieser Merge wird im folgenden Abschnitt näher beleuchtet:

1.1.4 2-Wege-Diff für MMR-Versionen

Sind erst einmal morphologische Modellrepräsentationen von den zu verwaltenden Versionen erzeugt, so ist die Anwendung eines 2-Wege-Vergleichsverfahrens trivial. Um das zu demonstrieren, wird die Modellversion aus Abbildung IV.3 als Version *VO* betrachtet. Wie in Abbildung IV.6 zu sehen, wird sie in einer Folgeversion *VA* um eine neue Komponente *CustomerProfile*, und um eine neue Schnittstelle mit den entsprechenden Beziehungen erweitert. Zudem wird die Komponente *CustomerAccountTransaction* in *AccountTransaction* umbenannt.

Der Vergleich zwischen den beiden Versionen *VO* und *VA* kann jetzt auf Basis der MMRs denkbar einfach durchgeführt werden. Dazu müssen lediglich die einzelnen MDUs beider Versionen verglichen werden. Ist eine MDU sowohl in *VO* als auch in *VA* vorhanden, so wurde die entsprechende Modellinformation nicht geändert (in der Abbildung beige in *VA* eingefärbt). Ist eine MDU in *VA* vorhanden aber nicht in *VO*, so wurde die entsprechende Modellinformation zum Modell hinzugefügt (grün in *VA* und im Diff eingefärbt). Ist eine MDU in *VO* aber nicht in *VA* vorhanden, so wurde die entsprechende Modellinformation aus dem Modell gelöscht (rot im Diff eingefärbt).

Auf Basis des vorgestellten Vergleichsverfahrens lässt sich die Evolution eines Modells durch Versionen und Diffs darstellen. Die Bestimmung der Diffs ist dabei die Voraussetzung für das i. F. vorgestellte 3-Wege-Merge-Verfahren.

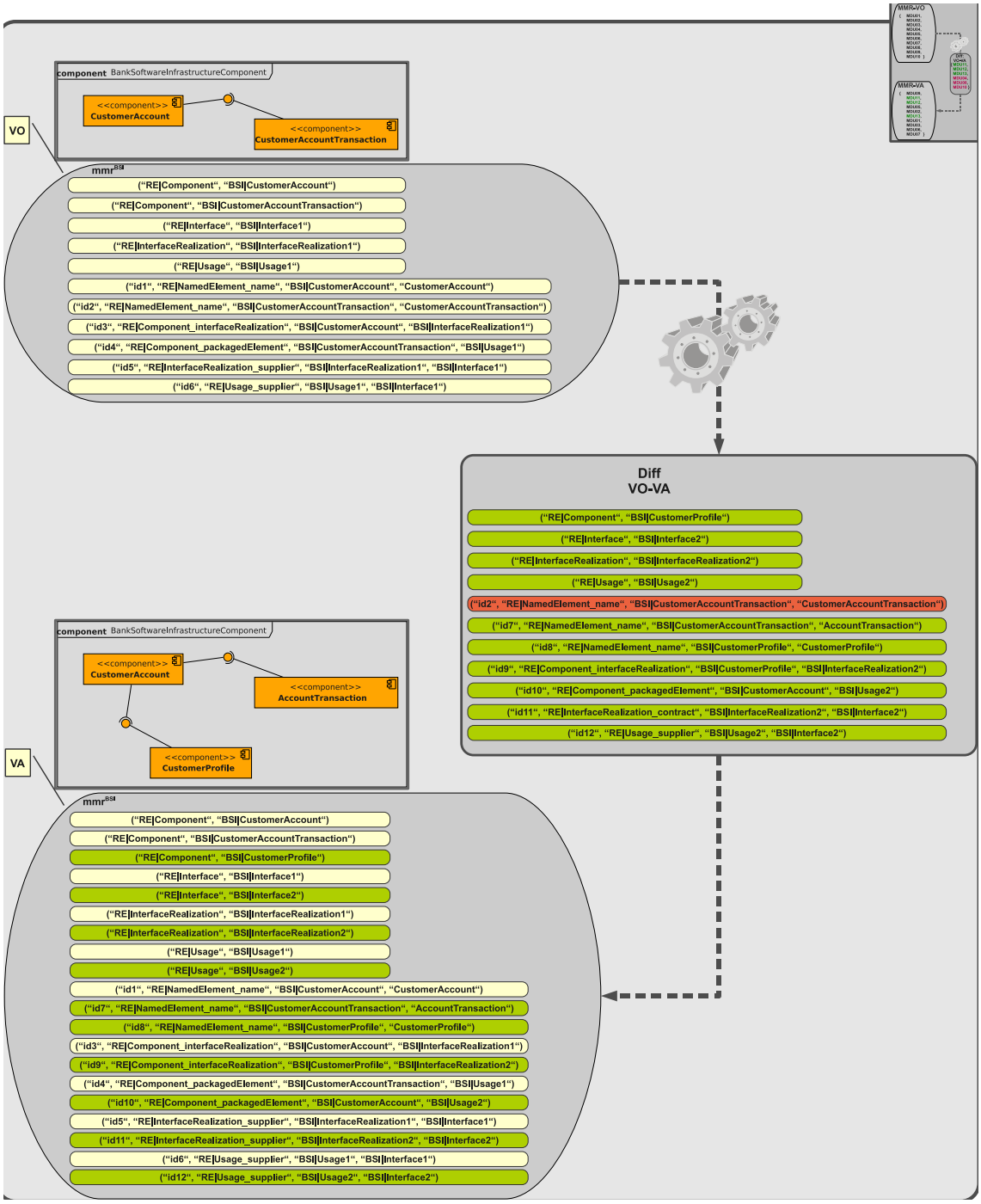


Abbildung IV.6. Erweiterung des Beispiels aus Abbildung IV.3 um 2-Wege-Diff zwischen Folgeversionen

1.2 3-Wege-Merge-Prinzip für MMR-Versionen

Wie in Abschnitt II2.2.3 beschrieben, können mithilfe eines zustandsbasierten Vergleichsverfahrens die Änderungen an Dokumenten bestimmt werden, die durch ein 3-Wege-Merge-Verfahren integriert werden sollen.

Bevor im Abschnitt 1.2.2 die Funktionalität eines 3-Wege-Merge-Verfahrens vorgestellt wird, das für die Verwaltung von MMRs geeignet ist, wird im folgenden Abschnitt ein Datenschema zur Abbildung einer Versionshistorie von Dokumenten erläutert. Dieses Datenschema unterstützt insbesondere die Abbildung der Evolution von Modelldaten in MMR und erlaubt damit eine feingranulare Dokumentation der Modellevolution auf der Ebene atomarer Informationsänderungen. Dabei wird insbesondere die asynchrone Entwicklung von Modellversionen berücksichtigt.

1.2.1 Ein Schema zur Repräsentation der asynchronen Evolution von Dokumenten

Im kollaborativen Entwicklungsprozess eines Dokumentes werden einzelne Ausschnitte fortlaufend durch verschiedene Personen geändert, neu hinzugefügt oder gelöscht. Um den Entwicklungsprozess zu dokumentieren bzw. um auf ältere Entwicklungsstände dauerhaft zugreifen zu können, werden von Zeit zu Zeit Versionen eines Dokumentes gespeichert. Arbeiten mehrere Bearbeiter in der Entwicklung eines Dokumentes zusammen, dienen die Versionen darüber hinaus als inhaltliche Synchronisationspunkte ihrer Arbeit (vgl. Abschnitt II.2).

Zur Repräsentation der Evolution von Dokumenten wird in diesem Abschnitt ein Datenschema beschrieben. Mit diesem können Änderungen an Dateneinheiten (i. F. *Datum*), in die ein Dokument zerlegt werden kann, erfasst werden. Dokumente, wie bspw. natürlichsprachlicher Text, Programmcode oder auch die MMR eines Modells, bestehen aus einer Zusammenstellung (engl. Composition) syntaktischer Einheiten (Wörter, Zeilen, MDUs etc.).

Die Frage, nach welchen Kriterien Modelldaten in sinnvolle Einheiten zerlegt werden können, wurde in Abschnitt 1.1 beantwortet. Bei der Erläuterung des folgenden Schemas wird nun ein Schritt zurückgegangen; das Schema betrachtet i. F. die Dateneinheiten von Versionen im Allgemeinen. Morphologische Dateneinheiten (MDUs), wie sie im letzten Abschnitt eingeführt wurden, stellen eine Spezialisierung der hier betrachteten Dateneinheiten dar.

Steht ein geeigneter Änderungsvergleich zur Verfügung, wie er für MMRs von Modellen beschrieben wurde, so kann die Versionshistorie der entsprechenden Dokumente mit dem in Abbildung IV.7 dargestellten Schema beschrieben werden.

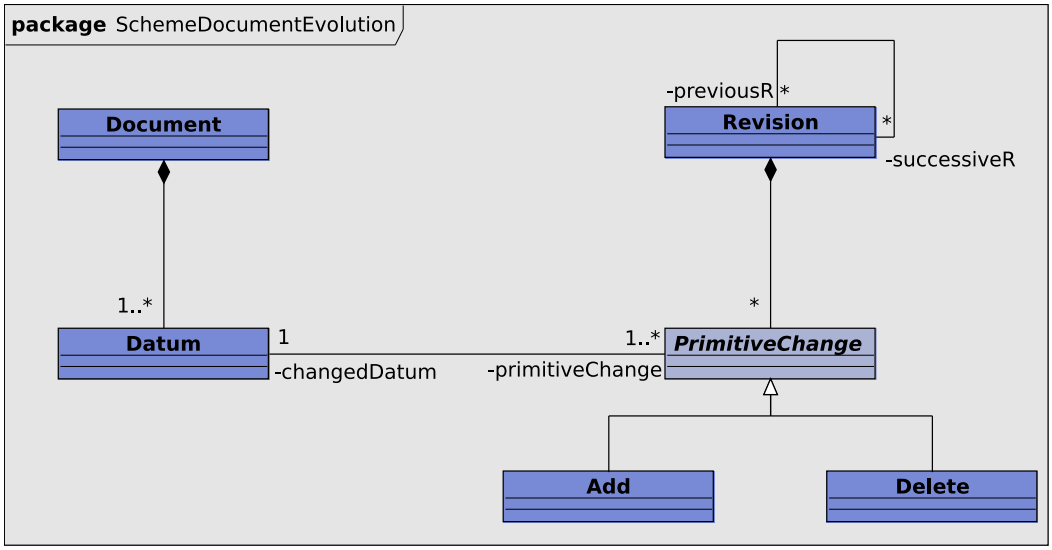


Abbildung IV.7. Datenschema zur Speicherung von Versions- und Änderungsinformationen

Die Granularität der Dokumentation von Änderungen an einem Dokument wird dabei durch dessen Zerlegung in Dateneinheiten bestimmt. Hierzu werden zunächst die Begriffe Dokument und Dateneinheit definiert:

Definition IV.10 (*Dateneinheit*)

Eine *Dateneinheit* (engl. *datum*) $a \in A$ ist eine syntaktische Einheit, deren Informationsgehalt nicht weiter spezifiziert wird. Die Menge $A \subsetneq ENTITY$ wird als Universum aller Dateneinheiten definiert.

Definition IV.11 (*Dokument*)

Ein *Dokument* (engl. *document*) $d \in D$ ist eine endliche Menge von Dateneinheiten, wobei $D = \{d \mid d \subset A, \mathrm{d\ endlich}\}$ als die Menge aller konstruierbaren Dokumente definiert ist.

Wie oben motiviert, sollen die Änderungen eines Dokumentes und damit seiner Dateneinheiten für eine später notwendige Analyse der Entwicklungshistorie aufgezeichnet werden. Dazu werden zwei weitere Konzepte eingeführt: *Primitive* (generische) *Änderung* in Definition IV.12 zur Beschreibung der Änderung einer Dateneinheit (vgl. Definition III.2) und *Revision* in Definition IV.14 zur Bestimmung einer Dokumentversion auf Basis der Änderungen an einer oder mehreren Vorgängerversionen.

Definition IV.12 (*Primitive Änderung*)

Eine *primitive Änderung* (engl. *primitive change*) $pc \in PC$ ist eine atomare (generische) Änderungsoperation, die auf genau eine Dateneinheit angewendet werden kann. Es werden zwei Arten von primitiven Änderungen unterschieden. Hinzufügungen $ADD \subseteq PC$ und Löschungen $DELETE \subseteq PC$ mit $ADD \dot{\cup} DELETE = PC$. $PC \subsetneq ENTITY$ ist da Universum aller primitiver Änderungen.

Zur Definition des Begriffs Revision muss zusätzlich die folgende Abbildung zur Bestimmung geänderter Dateneinheiten definiert werden:

Definition IV.13 (*geänderte Dateneinheit*)

Die surjektive Abbildung $changedDatum : PC \rightarrow A$ ordnet jeder primitiven Änderung die zu ändernde Dateneinheit zu.

Definition IV.14 (*Revision*)

Eine *Revision* (engl. *revision*) $r \in R \subseteq \wp(PC)$ beschreibt Änderungen an einem Dokument in Bezug zu einer Vorgängerrevision. Für Revisionen gelten folgende Bedingungen:

- $\forall r \in R : r$ ist endliche Menge,
- $\forall r \in R, \forall pc_1, pc_2 \in r : changedDatum(pc_1) = changedDatum(pc_2) \rightarrow pc_1 = pc_2$,
- $\forall r_1, r_2 \in \wp(PC) : \ (r_1 \neq r_2) \quad \rightarrow \quad (pc \in r_1 \rightarrow pc \notin r_2)$

$R \subset \wp(PC), \forall r \in R : r\ endlich$ ist die Menge aller Revisionen.

Bestimmt man die Beziehungen zwischen den Revisionen, so kann eine vollständige Versionshistorie inklusive der Änderungen auf Basis von Dateneinheiten dargestellt werden. Die Bestimmung der einzelnen Dokumentversionen, die sich aus den in den Revisionen enthaltenen primitiven Änderungen ergeben, wird im nachfolgenden Abschnitt beschrieben.

Mit Ausnahme der Wurzelrevision (zur Erstellung der initialen Dokumentversion) beziehen sich alle Revisionen eines Dokumentes immer auf mindestens eine Vorgängerrevision und ermöglichen die Darstellung eines Versionsgraphen. Ein solcher Graph ist azyklisch. Dies muss durch die Verknüpfung von Revisionen durch die Beziehungen $previousR$ und $successiveR$ berücksichtigt werden.

Das bedeutet, dass jede Revision beliebig viele Nachfolger hat, aber nicht in der durch die transitive Beziehung $previousR$ dargestellten Entwicklungshistorie enthalten ist:

Definition IV.15 (successiveR/previousR)

Die Abbildungen $successiveR$ und $previousR : R \to \wp(R)$ repräsentieren die Abhängigkeiten der durch die Revisionen dargestellten Dokumentbearbeitungen. Dabei beschreibt $previousR$ die Beziehung einer Revision zu ihren Vorgängerrevisionen und $successiveR$ die Beziehung einer Revision zu ihren Nachfolgern. Für die Abbildungen gelten folgende Bedingungen:

- $\forall r \in R : r_{prev} \in previousR(r) \leftrightarrow r \in successiveR(r_{prev})$,
- $\forall r \in R : r \notin previousR^{+}(r)$, wobei $previousR^{+}$ die transitive Hülle von $previousR$ darstellt (garantiert die Azyklizität des Versionsgraphen),
- $(successiveR^{+}(r_1) = successiveR^{+}(r_2) \ \wedge \ previousR(r_1) = previousR(r_2) = \emptyset) \quad \to \quad r_1 = r_2$, r_1 wird i. F. auch als Wurzelelement eines Versionsgraphen bezeichnet.

Mengen aus den Revisionen als Knoten und aus ihren Beziehungen untereinander (vgl. Definition IV.15) als Kanten entsprechen gerichteten, azyklischen Graphen. Diese werden in der Literatur als Versionsgraphen (vgl. (Conradi und Westfechtel 1998)) bezeichnet. Hat eine Revision mehrere Vorgängerrevisionen, so müssen die parallelen Änderungen der Dateneinheiten in der Versionshistorie integriert werden. Das dazu nötige Verfahren wird im nachfolgenden Abschnitt genauer beschrieben.

Abschließend wird noch eine Abbildung eingeführt, die jeder Dateneinheit die Menge an primitiven Änderungen zuordnet, durch die sie in der Versionshistorie geändert wurde. Diese Abbildung wird im nachfolgenden Abschnitt zur Beschreibung der 3-Wege-Merge-Semantik auf Basis des hier vorgestellten Schemas verwendet.

Definition IV.16 (primitiveChanges)

Die bijektive Abbildung $primitiveChanges : A \to \wp(PC) \setminus \emptyset$ mit:

$\forall pc \in PC, \forall a \in A : \quad pc \in primitiveChanges(a) \leftrightarrow changedDatum(pc) = a$

ordnet jeder Dateneinheit, die primitiven Änderungen zu, die die betreffende Dateneinheit ändern. Beachte: Nach Definition IV.14 kann eine Dateneinheit nur durch maximal eine Änderung in einer Revision geändert werden.

Auf Grundlage des vorgestellten Schemas lässt sich nun der Inhalt jeder Dokumentversion in der Versionshistorie bestimmen. Dazu müssen die primitiven Änderungen der Dateneinheiten, die eine Revision kapselt, auf die jeweilige Vorgängerversion angewendet werden. Wird dieses Verfahren rekursiv bis zum Wurzelelement des Versionsgraphen angewendet, so kann für jede Revision der Inhalt der zugehörigen Dokumentversion automatisch berechnet werden. Dieses Verfahren wird im folgenden Abschnitt im Detail beleuchtet.

1.2.2 Integration asynchroner durchgeführter Änderungen – Semantik eines 3-Wege-Merge-Verfahrens

Wie im letzten Abschnitt erörtert, können Dokumente als Menge von Dateneinheiten repräsentiert werden und ihre Veränderung in (möglicherweise parallelen) Entwicklungszweigen kann mit Versionsgraphen beschrieben werden. Die Zusammensetzung des Dokumentinhaltes, die sich aus den Änderungen der Dateneinheiten in den Revisionen,

wurde hingegen noch nicht erläutert. Insbesondere die Bestimmung einer Dokumentversion, die sich aus einer Revision mit mehreren Vorgängerrevisionen ergibt, ist nicht trivial. In diesem Fall müssen parallele Entwicklungszweige des Dokumentes zusammengeführt werden.

In diesem Abschnitt wird erläutert, wie aus den Revisionen (des in Abschnitt IV.1.2.1 eingeführten Schemas) die einzelnen (ggf. zusammengeführten) Versionen eines Dokumentes bestimmt werden. Abbildung IV.8 stellt den i. F. fokussierten Ausschnitt dar.

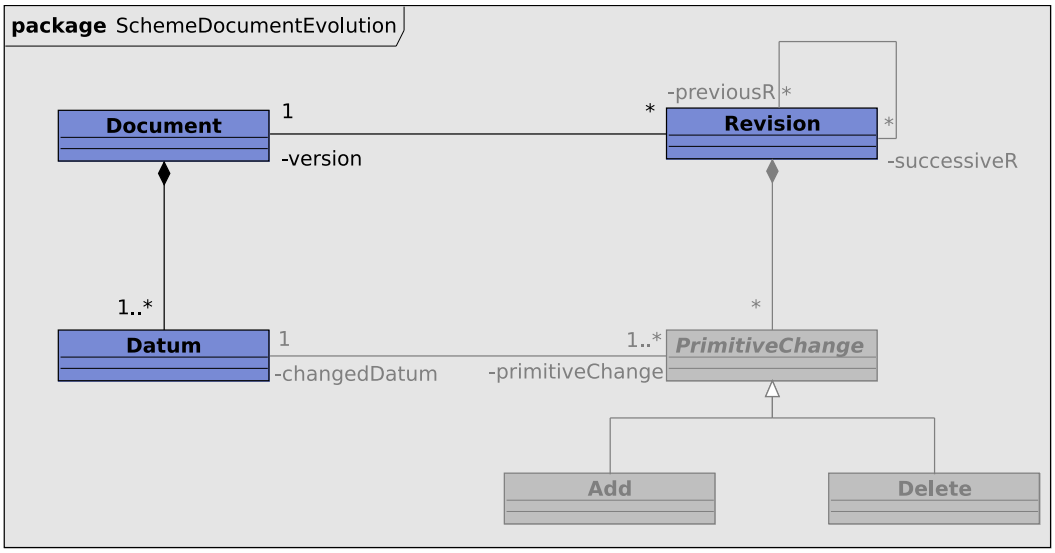


Abbildung IV.8: Herausforderung beim Zusammenführen - Bestimmung der Versionen

Zur Bestimmung einer Modellversion müssen Historie einer Revision und insbesondere deren Änderungen an Dateneinheiten vollständig betrachtet werden. Dafür werden zunächst für jede Dateneinheit die jüngsten primitiven Änderungen in der Historie einer Revision bestimmt. Diese Bestimmung wird durch die Abbildung *lastChanges*, die in Definition IV.17 eingeführt wird, beschrieben.

Definition IV.17 (lastChanges)

Die Abbildung $lastChanges : R \times A \rightarrow \wp(PC)$ bestimmt die letzten primitiven Änderungen einer bestimmten Dateneinheit in der Historie einer bestimmten Revision. Sie ist wie folgt definiert:

$$lastChanges(r,a) = \begin{cases} r \cap primitiveChanges(a) & : \quad r \cap primitiveChanges(a) \neq \emptyset \\ \bigcup\limits_{r_{prev} \in previousR(r)} lastChanges(r_{prev}, a) & : \quad else \end{cases}$$

Zur Bestimmung aller Revisionen in der Historie einer bestimmten Revision wird die Abbildung *allPreviousR* in Definition IV.18 eingeführt. Darüber hinaus wird zur Bestimmung aller primitiven Änderungen einer bestimmten Dateneinheit in der Historie einer bestimmten Revision wird die Abbildung *changeHistory* in Definition IV.19 eingeführt.

Definition IV.18 (allPreviousR)

Die Abbildung $allPreviousR : R \rightarrow \wp(R)$ ordnet jeder Revision $r \in R$ die Menge der Revisionen zu, die transitiv über die Abbildung $previousR$ mit r verbunden sind. Sie ist wie folgt definiert:

$$allPreviousR(r) = r \cup \left(\bigcup_{r_{prev} \in previousR(r)} allPreviousR(r_{prev}) \right)$$

Definition IV.19 (changeHistory)

Die Abbildung $changeHistory : PC \times A \rightarrow \wp(PC)$ bestimmt für eine Dateneinheit $a \in A$ und eine Revision $r \in R$ alle primitiven Änderungen von a, die in der Historie der Revision r an a vorgenommen wurden.

$$changeHistory(r, a) = primitiveChanges(a) \cap \left(\bigcup_{r_{prev} \in previousR(r)} \left(\bigcup_{r_{allPrev} \in allPreviousR(r_{prev})} r_{allPrev} \right) \right)$$

Mit den eingeführten Abbildungen lässt sich nun ermitteln, ob eine bestimmte Dateneinheit in ihrer Entwicklungshistorie parallel geändert wurde. Die entsprechenden primitiven Änderungen werden durch die Abbildung *concurrentChanges*, die in Definition IV.20 beschrieben ist, bestimmt.

Definition IV.20 (concurrentChanges)

Die Abbildung $concurrentChanges : R \times A \rightarrow \wp(PC)$ bestimmt die Menge aller unabhängigen primitiven Änderungen einer Dateneinheit $a \in A$, die in der Historie der Revision $r \in R$ enthaltenen sind. Sie ist wie folgt definiert:

$$concurrentChanges(r, a) = lastChanges(r, a) \setminus \left(\bigcup_{r_{pc} \in \{r_l | r_l \in R,\ r_l \cap lastChanges(r,a) \neq \emptyset\}} changeHistory(r_{pc}, a) \right)$$

Seien $r \in R$ eine Revision und $a \in A$ eine Dateneinheit und $Pc := concurrentChanges(r, a)$. Dann ergeben sich aus der Definition IV.20 drei Arten der Struktur der Menge Pc:

i. $\forall pc \in Pc : pc \in ADD$: Die letzte Änderung der Dateneinheit a in allen (möglicherweise parallelen) Entwicklungszweigen in der Historie von r war eine Hinzufügung.

ii. $\forall pc \in Pc : pc \in DELETE$: Die letzte Änderung der Dateneinheit a in allen (möglicherweise parallelen) Entwicklungszweigen in der Historie von r war eine Löschung.

iii. $\exists pc_1, pc_2 \in Pc, pc_1 \neq pc_2 : pc_1 \in ADD \ \wedge \ pc_2 \in DELETE$: Die letzten Änderungen der Dateneinheit a in mindestens zwei parallelen Entwicklungszweigen in der Historie von r sind unterschiedlichen Typs.

Im Fall (a) und (b) gibt es keine Widersprüche bzgl. des Änderungsstatus' der zusammengeführten Version – (a) → ADD, (b) → DELETE. Der Fall (c) entspräche einer jeweils unterschiedlichen Änderung ein und derselben Dateneinheit in mindestens zwei parallelen Entwicklungszweigen der Historie.

Durch die Verwendung von Unified Universal IDs (UUIDs) (Network Working Group 2010) beim Hinzufügen neuer Dateneinheiten wird das Eintreten dieses Falles vermieden. Werden zur Vermeidung von Dubletten bei der Erstellung neuer Dateneinheiten keine UUIDs sondern aus der Syntax abgeleitete IDs verwendet (bspw. Bezeichner für syntaktische Elemente wie UML-Klassennamen), so können durch *concurrentChanges* die Änderungen identifiziert werden, die widersprüchlich bzgl. ihres Änderungszustandes wie in Fall iii. sind. Somit kann der Fall iii. wahlweise vermieden bzw. behandelt werden und wird deswegen i. F. nicht ausführlicher diskutiert.

Nach einer möglicherweise notwendigen Behandlung widersprüchlicher Änderungen, kann eine zusammengeführte Version eines Dokumentes – *revisedVersion* – für jede Revision bestimmt werden. Die bestimmende Abbildung ist in Definition IV.21 angegeben.

Definition IV.21 (revisedVersion)

Die Abbildung $revisedVersion : R \rightarrow D$ bestimmt für jede Revision, die Dateneinheiten, die zur zugehörigen Dokumentversion gehören. Sie ist wie folgt definiert:

$$revisedVersion(r) = \left\{ a | a \in A \text{ with } \begin{array}{l} concurrentChanges(r, a) \neq \emptyset \quad \text{and} \\ \forall pc \in concurrentChanges(r, a) : pc \in ADD \end{array} \right\}$$

Auf Grundlage von Definition IV.21 lassen sich alle Versionen in der Historie einer Dokumententwicklung eindeutig bestimmen, incl. derer, die aus mehreren Entwicklungszweigen zusammengeführt wurden. Die Abbildung *revisedVersion* setzt somit das 3-Wege-Merge-Prinzip für Mengen von Dateneinheiten um.

In Abschnitt 1.1.1 wurde beschrieben, wie Modellinformationen in morphologische Dateneinheiten zerlegt werden können und in Abschnitt 1.1.4 wurde beschrieben, wie primitive (generische) Änderungen an MDUs automatisch berechnet werden. Mit dem oben vorgestellten Verfahren können nun MDUs und Änderungen dieser nach dem 3-Wege-Merge-Prinzip automatisch verarbeitet werden. Damit gelingt es, Änderungen in möglicherweise asynchron erzeugten Modellversionen zu einer neuen Version auf Basis der MMR zu integrieren.

Die Verarbeitung als morphologische Modellrepräsentation vermeidet damit Integrationsprobleme asynchroner Änderungen, wie sie in Abbildung III.13 illustriert wurden. Integrationskonflikte, wie sie in Abbildung III.13, Abbildung III.16, Abbildung III.18 und Abbildung III.20 dargestellt wurden, können nach der oben beschriebenen Änderungsintegration immer noch auftreten.

Die Probleme, die in Abbildung III.13 und Abbildung III.16 dargestellt wurden verhindern sogar die Transformation der MMR nach XMI. Mit einer Automatisierung der merge-bedingten Validierung von Modellen, deren MMR ein Resultat des oben beschriebenen 3-Wege-Merge-Verfahrens ist, beschäftigt sich der folgende Abschnitt.

2 Automatisierung statischer Analysen zur merge-bedingten Modellvalidierung

Durch den Merge-Ansatz, der in den vorangegangenen Abschnitten beschrieben wurde, können Änderungen an Modellen, die asynchron von verschiedenen Entwicklern durchgeführt wurden, wieder zu einer Version auf Basis der MMR zusammengeführt werden. Die dabei automatisch erzeugten Modellversionen können allerdings inkorrekt hinsichtlich des Metamodells sein, das die Modellierungssprache vorgibt.

Dies ist leicht nachzuvollziehen, wenn eine der Versionen, die automatisch gemergt werden, ohnehin inkorrekt ist. Dieser triviale Fall steht i. F. nicht im Zentrum der Betrachtung, da analog zur Codeverwaltung vorausgesetzt wird, dass alle Modellversionen, die in ein Repository übertragen werden, eine umfassende Validierung erfolgreich bestehen (vgl. Abschnitt III.2.2 Grundsatz in der Codeverwaltung – Was nicht kompiliert, darf nicht eingecheckt werden!).

Im Zentrum der Betrachtung dieses Kapitels steht die Erweiterung des vorgestellten Merge-Verfahrens um die Erkennung und Analyse von modellierungssprachspezifischen Fehlern und allgemeiner Anomalien, die entstehen, obwohl die zu integrierenden Modellversionen jeweils korrekt sind. Ziel der Analyse ist, diese Fehler „vollautomatisch“ im Ergebnismodell zu lokalisieren und die ursächlichen asynchronen Änderungen zu bestimmen. Sind diese Fehler lokalisiert und klassifiziert, müssen sie manuell von den beteiligten Modellierern behoben werden.

Abbildung IV.9 zeigt überblicksartig die Einbettung der Modellvalidierung (Schritt 6 und 7) in das gesamte Merge-Verfahren. In Schritt 5 sieht man ein mit dem morphologischen Merge-Verfahren (MMV) erzeugtes Modell *VM*. Dieses wird in Schritt 6 hinsichtlich Korrektheit bzgl. dem zugehörigen Metamodell validiert. Diese Prüfung kommt zu dem Ergebnis, dass *VM* inkorrekt ist. Eine entsprechende Beschreibung der Fehler (Log) wird automatisch erzeugt. Mithilfe dieser Beschreibung muss die Modelliererin *Alice* die Fehler korrigieren, bevor sie die als korrekt geprüfte Version *VM′* ins Repository überträgt. Die dabei eingesetzte automatische Validierung von Modellen ist Gegenstand des vorliegenden Abschnitts.

Dieser gliedert sich in vier Unterabschnitte. Zunächst wird ein allgemeiner Überblick zum konkreten Lösungsansatz der statischen Analyse vorgestellt. Dieser basiert auf der Transformation von Modellen und Metamodellen in eine beschreibungslogische Wissensbasis und eine Bibliothek von Anfragen der Anfragesprache SPARQL-DL. Abschnitt 2.2 stellt die beiden verwendeten Sprachen vor. In den Abschnitten 2.2 bis 2.4 werden zwei Alternativen zur Validierung von Modellen auf Basis von Beschreibungslogiken und SPARQL durch statische Analysen erörtert. Abschnitt 2.5 beschäftigt sich mit der Erkennung von Anomalien ebenfalls durch die Transformation von Modellen in beschreibungslogische Ausdrücke.

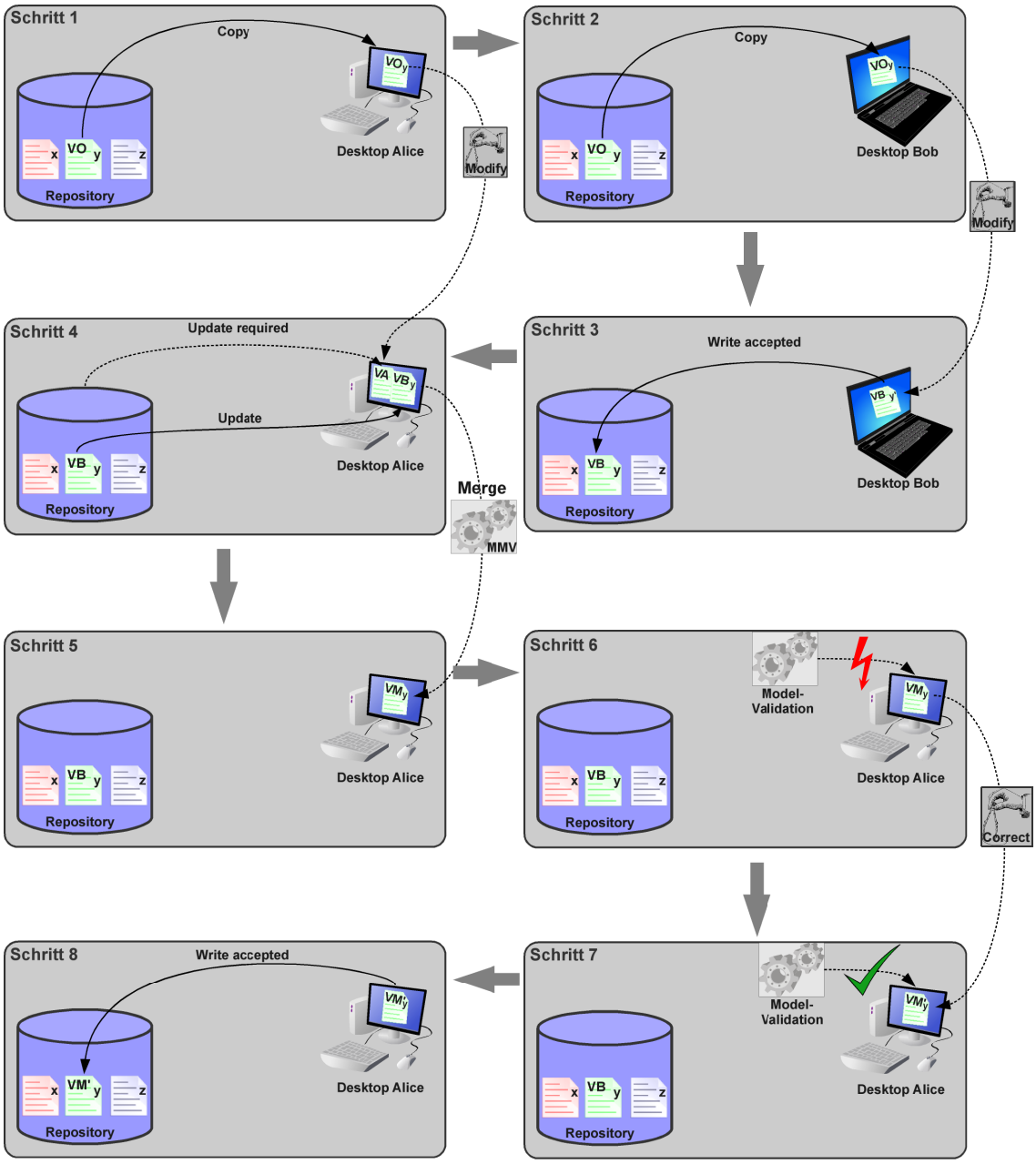


Abbildung IV.9. Einbettung der merge-bedingten Validierung ins Merge-Verfahren

2.1 Überblick: Merge-bedingte Validierung von Modellen in MMR

Ausgangspunkt für eine automatische Korrektheitsanalyse von Modellen im Anschluss an das morphologische Merge-Verfahren (MMV) ist eine maschinenverarbeitbare Beschreibung vom Modell einerseits und dem Metamodell der entsprechenden Modellierungssprache anderseits.

Beides ist bei der werkzeuggestützten Modellierung im Software Engineering vorhanden. Metamodelle von Softwaremodellierungssprachen sind i. d. R. als MOF- bzw. ECore-Modelle spezifiziert und können als XMI-Dateien von fast allen Modellierungswerkzeugen verarbeitet werden.

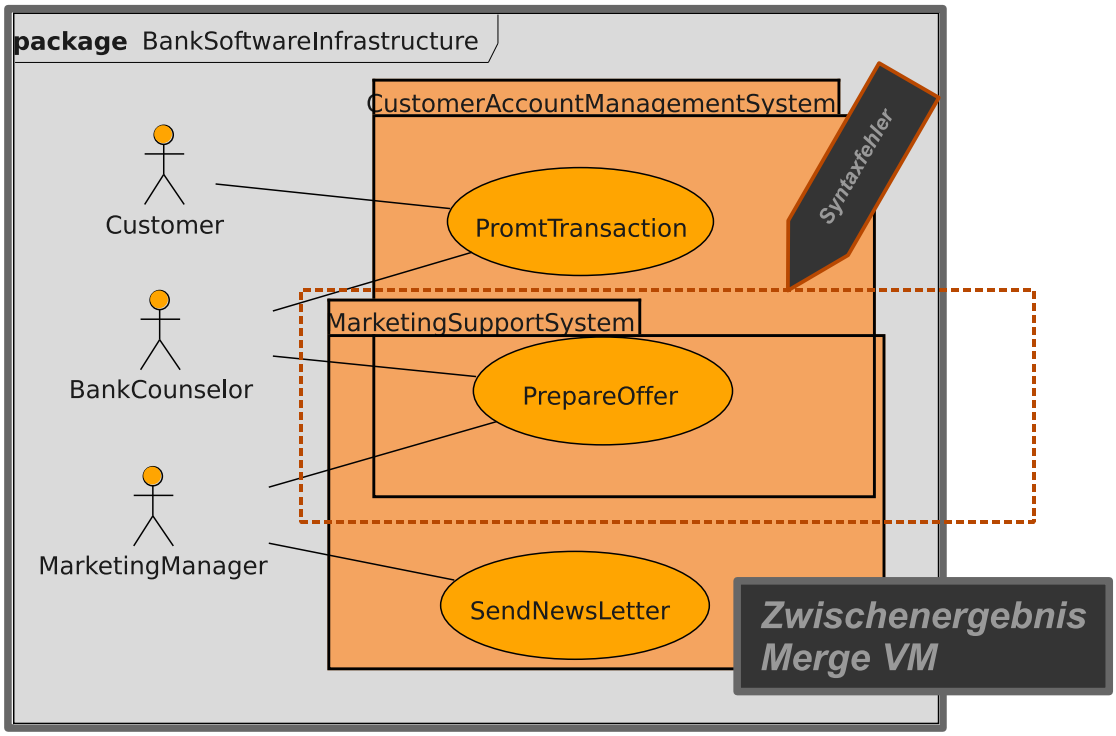


Abbildung IV.10. Merge-bedingter Fehler im Modell nach Änderungsintegration (vgl. Abbildung III.18)

Integrierte Modellversionen stehen im Anschluss an das oben beschriebene Merge-Verfahren in der MMR zur Verfügung. Aufgrund merge-bedingter syntaktischer Inkorrektheit kann nicht jede MMR eines Modells in eine XMI-Repräsentation transformiert werden. Abbildung IV.10 greift dafür das Beispiel aus Abbildung III.18 auf. Nach der dort dargestellten Änderungsintegration müsste der Anwendungsfall *PrepareOffer* in zwei verschiedenen Systemen enthalten sein – *CustomerAccountManagementSystem* und *MarketingSupportSystem*. Die Enthaltensein-Beziehung (engl. *Composite*) wird in XMI durch die Hierarchie zwischen Vater- und Kind-XML-Knoten ausgedrückt. Danach kann ein Objekt (wie in diesem Fall *PrepareOffer*) nicht gleichzeitig als ein XML-Kind-Knoten zweier Vaterknoten ausgedrückt werden.

Derartige Konflikte beim Merge führen zu unüberwindlichen Schwierigkeiten bei der Transformation von MMR-Merge-Ergebnissen in die XMI-Repräsentation. Wegen dieses Umstandes kann eine sich anschließende Validierung nicht immer anhand von XMI-Dateien erfolgen. Eine Lösung dieses Problems ist die Validierung der abstrakten Modellsyntax auf Basis der MMR. Dazu wird die MMR von Modell und Metamodell in eine semantisch fundierte Beschreibungssprache transformiert.

Hierbei kommen eine beschreibungslogische Sprache wie OWL(2) und eine entsprechende Anfragesprache wie SPARQL(-DL) zum Einsatz. Der i. F. beschriebene Ansatz zur Validierung

wurde in Abbildung III.14 überblicksartig beschrieben und eingeordnet. Er wird nun im Detail erläutert. Wie in Abbildung IV.11 beschrieben, besteht die Validierung aus *Stage 1* und *2*. In *Stage 1* wird die MMR eines Modells als sog. ABox einer beschreibungslogische Wissensbasis ausgedrückt. Zur statischen Analyse der durch Metamodell spezifizierten Syntax wird dieses in *Stage 2* automatisch in eine sog. TBox umgewandelt; darüber hinaus wird eine Bibliothek mit SPARQL-Anfragen aus dem Metamodell generiert. Mit diesen kann gezielt nach Modellstrukturen gesucht werden, die die Metamodellvorgaben verletzen.

Nachdem die erwähnten Wissensbasen und die Query-Bibliothek erzeugt wurden, kann mithilfe sog. Reasoner bzw. Query-Engines nach merge-bedingten Fehlern gesucht werden. Zur Fehlersuche werden dafür die vom Reasoner angebotenen ABox- bzw. TBox-Inferenzservices zur Prüfung der Konsistenz (engl. consistency) der Wissensbasis bzw. zur Prüfung der Klassenkonsistenz (engl. concept satisfiability) benutzt. Mithilfe einer Query Engine werden die Anfragen der Query-Bibliothek ausgewertet. Beispielsweise identifiziert die folgende Anfrage den Fehler in Abbildung IV.10:

`Enthält das Modell ein Objekt, dass durch mehrere Ausprägungen der Kompositionsbeziehung referenziert wird?`

Die Antwort in diesem Fall wäre: **`PrepareOffer`**, da die in der Anfrage formulierte Eigenschaft für dieses Objekt zutrifft. Die oben motivierte und in den folgenden Abschnitten im Detail erörterte statische Analyse von Modellen auf Basis von Beschreibungslogiken wird in Abbildung IV.11 überblicksartig illustriert. Voraussetzung für die statische Analyse ist die in *Stage 1* dargestellte Erstellung einer MMR aus der Modellversion im Repository. Vergleichbar ist dieser Schritt mit der Aufgabe eines Lexers (vgl. auch Abbildung II.11 und Abbildung II.12). Nachdem die MMR einer Modellversion erzeugt wurde, kann diese mithilfe sogenannter Interpretationsabbildungen als sogenannte beschreibungslogische ABox ($\mathcal{I}^{ABox}_{MMR^{\hat{m}}}$) bzw. TBox ($\mathcal{I}^{TBox}_{MMR^{\hat{m}}}$) ausgedrückt werden.

Die Modellversion ausgedrückt als ABox dient dabei als Ausgangspunkt für die statischen Analysen zur Validierung der durch das MOF-Metamodell spezifizierten Syntax und weiterer (kontextsensitiver) Nebenbedingungen (bspw. in OCL ausgedrückt) in *Stage 2*. Für diese statischen Analysen der Korrektheit bzgl. einer Sprachspezifikation werden zwei Alternativen – ABox-Reasoning und Query-Processing – in Abschnitt 2.3 bzw. Abschnitt 2.4 vorgestellt. Für die beiden Analysealternativen muss die Syntaxspezifikation der jeweiligen Modellierungssprache in eine beschreibungslogische TBox bzw. in eine Bibliothek von entsprechenden Anfragen ausgedrückt werden. Dazu dienen die beiden Interpretationsabbildungen $\mathcal{I}^{TBox}_{MMR^{ECore}}$ bzw. $\mathcal{I}^{SPARQL\,DL}_{MMR^{ECore}}$.

Die Modellversion ausgedrückt als TBox dient darüber hinaus als Ausgangspunkt für die statische Analyse zur Erkennung von Anomalien, wie sie in Abschnitt III.4.4.3 erörtert wurden. Zur Erkennung der entsprechenden Anomalien wird das TBox-Reasoning eingesetzt (vgl. Abschnitt 2.5).

Bevor das Konzept der Modellvalidierung durch statischen Analysen der Syntax im Detail erläutert wird, stellt der nächste Abschnitt die Grundlagen zu beschreibungslogischen Sprachen wie OWL DL und zu Anfragesprachen wie SPARQL vor. Die dann anschließenden Abschnitte erörtern die in Abbildung IV.11 dargestellte Transformations- und Analysephasen. Dazu wird insbesondere auf die vier in *Stage 1* und 2 (Abbildung IV.11) dargestellten Interpretationsabbildungen eingegangen. Zur Illustration werden die in Kapitel III vorgestellten Beispiele von Merge-Konflikten herangezogen.

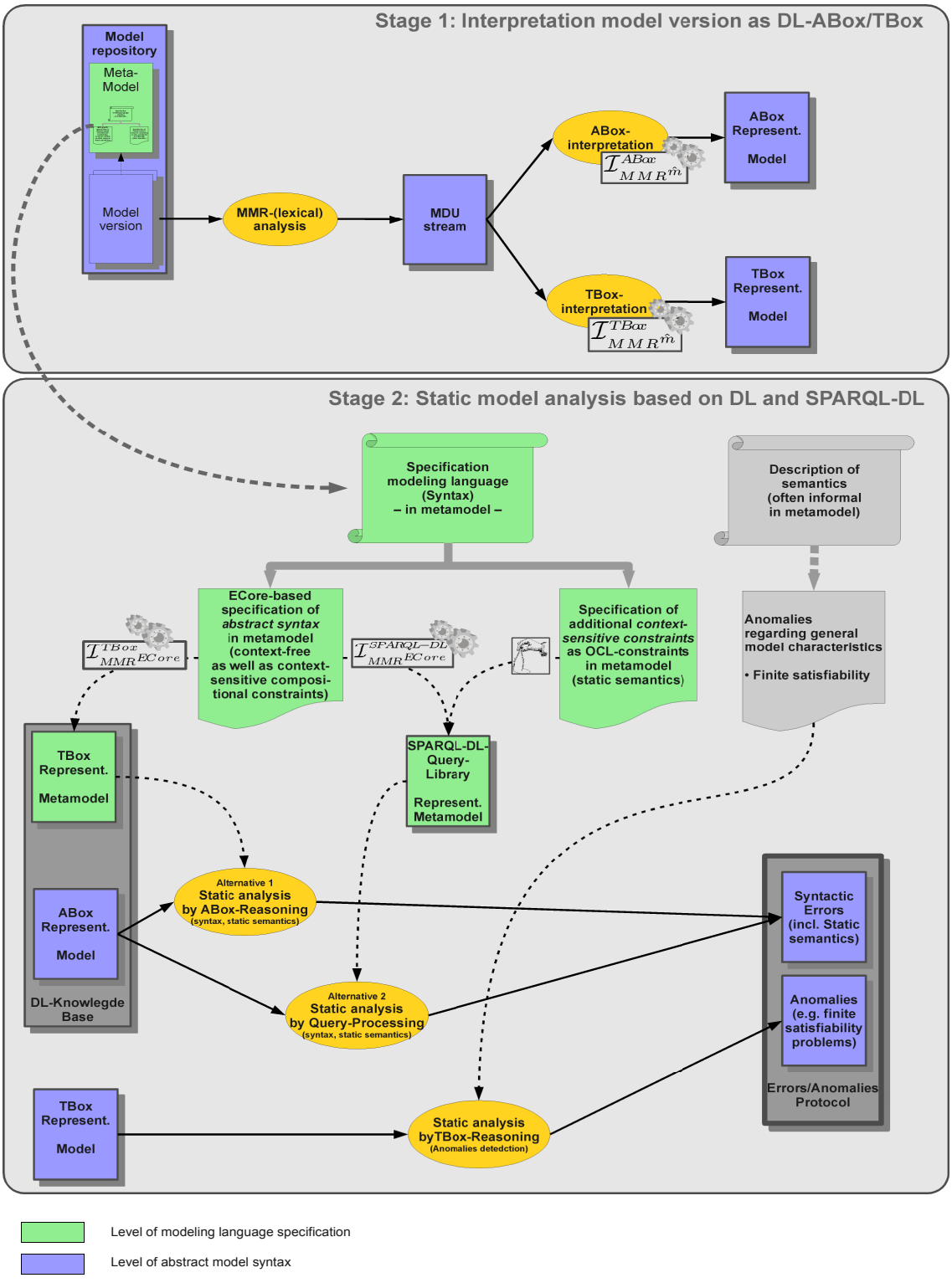


Abbildung IV.11. Merge-bedingte Validierung durch statische Analyse abstrakter Syntaxgraphen von Modellen – Überblick (vgl. 2. Validierung in Abbildung III.14)

2.2 Beschreibungslogiken, OWL und die Anfragesprache SPARQL

Beschreibungslogiken (*description logics – DL*) sind eine Familie formaler Sprachen zur Repräsentation von Wissen. Sie sind eine „Untermenge“ der Prädikatenlogik erster Stufe. Dabei ist die Prüfung der Ableitbarkeit von Aussagen auf Basis von DL im Gegensatz zur Prädikatenlogik immer entscheidbar.

Die gängigsten Inferenzsysteme sind als sog. Theorembeweiser auf Basis der *Prädikatenlogik erster Stufe* (*first order logic FOL*) realisiert. Sie implementieren i. d. R. einen korrekten, vollständigen Kalkül (i. d. R. den Tableaukalkül). Da die FOL nicht entscheidbar ist, können Theorembeweiser auf deren Basis i. A. nicht die Widerspruchsfreiheit eines beliebigen logischen Systems ohne zusätzliche Experteninteraktion beweisen.

Die Ausdrucksmöglichkeiten von Beschreibungslogiken sind im Vergleich zur Prädikatenlogik erster Stufe eingeschränkt. Grund für diese Einschränkung ist die Gewährleistung der Entscheidbarkeit beim logischen Schlussfolgern, die bei Prädikatenlogik erster Stufe (semi-entscheidbar) nicht gegeben ist. Das ermöglicht beschreibungslogischen Theorembeweisern, die Widerspruchsfreiheit eines axiomatischen Systems „vollautomatisch“ nachzuweisen.

So kann bspw. die sog. globale Konsistenz (Frage: Ist eine Wissensbasis widerspruchsfrei?) als ABox-Reasoning-Services geprüft werden (vgl. Abbildung IV.11). Ferner lässt sich die sog. Klassenkonsistenz (engl. concept satisfiability) (Frage: Kann es prinzipiell zu jeder TBox-Klasse min. ein Individuum in der ABox geben?) durch TBox-Reasoning-Services prüfen. Beide Prüfungen sind hingegen für die Prädikatenlogik erster Stufe im Allgemeinen nicht entscheidbar.

Eine zentrale Motivation bei der Entwicklung dieser Art von Logiken war, eine Beschreibungstechnik für Domänenwissen bereitzustellen, die es erlaubt, korrekt und vollständig alle wahren Aussagen aus einer Domänenbeschreibung automatisch entscheidbar abzuleiten.

Damit diese Logiken für den Einsatz in der Praxis tauglich sind, wurde insbesondere darauf geachtet, dass ihre Ausdrucksstärke im Rahmen der Entscheidbarkeit möglichst maximal ist. Da der Berechnungsaufwand wegen der Komplexität des Schlussfolgerns bei ausdrucksstarken Beschreibungslogiken i. A. unpraktikabel hoch sein kann (bis zu $NEXPTIME$), muss die Inferenzimplementierung eines Reasoners gut skalieren und hochoptimiert sein.

Eine Beschreibungslogik besteht aus einer sog. *TBox* (*terminological knowledge*) die das Domänenwissen als Menge von logischen Ausdrücken enthält und einer *ABox* (*assertional knowledge*), die die Individuen (Instanzen der Domänenentitäten) und ihre Beziehungen beschreibt.

Eine in der Wissensmodellierung vielfach eingesetzte Beschreibungslogik ist $\mathcal{SHOIN}$ (Hitzler u. a. 2008). Ihre Popularität verdankt sie zu einem großen Teil der Standardisierung der Web Ontology Language (OWL) (W3C 2004). OWL DL bietet eine maschinenverarbeitbare Repräsentationsform für $\mathcal{SHOIN}$ an. Außerdem existieren hochoptimierte Reasoner für diese Beschreibungslogik.

2.2.1 Die Beschreibungslogik $\mathcal{SHOIN}$

Die Beschreibungslogik $\mathcal{SHOIN}$ ist eine der populärsten Beschreibungslogiken im Bereich der Wissensrepräsentation und insbesondere im Bereich des Semantic Web. Sie ist die theoretische Grundlage für die Web Ontology Language (OWL) und wurde vom W3C als Syntax zur Wissensrepräsentation standardisiert.

In Tabelle 1 wird die Syntax und modelltheoretische Semantik von $\mathcal{SHOIN}$ vorgestellt.

Tabelle 1: Modelltheoretische Semantik von $\mathcal{SHOIN}$ (Ausschnitt vgl. (Hitzler u. a. 2008))

	Name	**Syntax**	**Intuitive Semantik / Modelltheoretische Semantik**
Klassen- und Rollenausdrücke	Klassennamen	$\mathbf{A}, \mathbf{B}$	Bezeichner für atomare Klassen
			$\mathbf{A}, \mathbf{B} \in \mathcal{C}$
	Leere Klasse	$\bot$	Bezeichnet die Klasse, die keine Individuen enthält.
			$(\bot)^{\mathcal{I}} = \emptyset$
	Universelle Klasse	$\top$	Bezeichnet die Klasse, die alle Individuen der Domäne Δ enthält.
			$(\top)^{\mathcal{I}} = \Delta$
	Konjunktion	$\mathbf{C} \sqcap \mathbf{D}$	Bezeichnet die Klasse aller zu $\mathbf{C}$ wie auch $\mathbf{D}$ zu gehörenden Individuen
			$(\mathbf{C} \sqcap \mathbf{D})^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
	Disjunktion	$\mathbf{C} \sqcup \mathbf{D}$	Bezeichnet die Klasse aller zu mindestens einer der Klassen $\mathbf{C}$ bzw. $\mathbf{D}$ gehörenden Individuen
			$(\mathbf{C} \sqcup \mathbf{D})^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
	Negation	$\neg\mathbf{C}$	Bezeichnet die Klasse aller nicht zur Klasse $\mathbf{C}$ gehörenden Individuen aus der gesamten Domäne Δ
			$(\neg\mathbf{C})^{\mathcal{I}} = \Delta \setminus \mathbf{C}^{\mathcal{I}}$
	Existiert-Rollenrestriktion	$\exists R.\mathbf{C}$	Repräsentiert die Klasse all der Individuen, die über die Rolle R mit einem Individuum verbunden sind, welches seinerseits der Klasse $\mathbf{C}$ zugehört.
			$(\exists R.\mathbf{C}.)^{\mathcal{I}} = \{x \mid \text{es gibt ein } y \text{ mit } (x,y) \in R^{\mathcal{I}} \text{ und } y \in \mathbf{C}^{\mathcal{I}}\}$
	Universal-Rollenrestriktion	$\forall R.\mathbf{C}$	Repräsentiert die Klasse, die ein Individuum x genau dann enthält, wenn *jedes* der mit x über die Rolle R verbundene Individuum zu $\mathbf{C}$ gehört.
			$(\forall R.\mathbf{C})^{\mathcal{I}} = \{x \mid \text{aus } (x,y) \in R^{\mathcal{I}} \text{ folgt stets } y \in \mathbf{C}^{\mathcal{I}}\}$
	Größer-gleich-Rollenrestriktion	$\leq n\, R$	Repräsentiert die Klasse, die all jene Individuen enthält, welche mit nicht mehr als n anderen Individuen über die Rolle R verbunden sind.
			$(\leq n\, R.\mathbf{C})^{\mathcal{I}} = \{x \mid \#(x,y) \in R^{\mathcal{I}} \leq n\}$
	Kleiner-gleich-Rollenrestriktion	$\geq n\, R$	Repräsentiert die Klasse, die all jene Individuen enthält, welche mit mindestens n anderen Individuen über die Rolle R verbunden sind.

			$(\geq n\, R.\mathbf{C})^{\mathcal{I}} = \{x \mid \#(x,y) \in R^{\mathcal{I}} \geq n\}$
	Nominale	$\{a\}$	Bezeichnet eine Menge von konkreten Individuen.
	Rollennamen	R, S	Bezeichner für Rollen.
			$R, S \in \mathcal{R}$
	Inverse Rollen	R^-	Repräsentiert die inverse Rolle.
			$(R^-)^{\mathcal{I}} = \{(b,a) \mid (a,b) \in R^{\mathcal{I}}\}$
TBox (Axiome)	Klasseninklusion	$\mathbf{C} \sqsubseteq \mathbf{D}$	Wird genau dann *wahr*, wenn jedes Individuum aus der mit $\mathbf{C}$ bezeichneten Klasse auch zur mit $\mathbf{D}$ bezeichneten Klasse gehört.
			$(\mathbf{C} \sqsubseteq \mathbf{D})^{\mathcal{I}} = wahr$ gdw. $\mathbf{C}^{\mathcal{I}} \subseteq \mathbf{D}^{\mathcal{I}}$
	Äquivalenz	$\mathbf{C} \equiv \mathbf{D}$	Wird genau dann *wahr*, wenn jedes Individuum aus der mit $\mathbf{C}$ bezeichneten Klasse auch zur mit $\mathbf{D}$ bezeichneten Klasse gehört und umgekehrt.
			$(\mathbf{C} \equiv \mathbf{D})^{\mathcal{I}} = wahr$ gdw. $\mathbf{C}^{\mathcal{I}} = \mathbf{D}^{\mathcal{I}}$
	Rolleninklusion	$R \sqsubseteq S$	Wird genau dann *wahr*, wenn jedes durch die Rolle R verbundene Individuenpaar auch durch die Rolle S verbunden wird.
			$(R \sqsubseteq S)^{\mathcal{I}} = wahr$ gdw. $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
	Transitivität	$\mathrm{Tra}(R)$	Wird dann zu *wahr* ausgewertet, wenn die durch R bezeichnete Rolle transitiv ist.
			$(\mathrm{Tra}(R))^{\mathcal{I}} = wahr$ gdw. für alle $(x,y),(y,z) \in R^{\mathcal{I}}$ gilt:$(x,z) \in R^{\mathcal{I}}$
	Symmetrie	$\mathrm{Sym}(R)$	Wird dann zu *wahr* ausgewertet, wenn die durch R bezeichnete Rolle symmetrisch ist.
			$(\mathrm{Sym}(R))^{\mathcal{I}} = wahr$ gdw. für alle $(x,y) \in R^{\mathcal{I}}$ gilt: $(y,x) \in R^{\mathcal{I}}$
ABox (Fakten)	Klassenzugehörigkeit	$\mathbf{C}(a)$	Wird genau dann zu *wahr* ausgewertet, wenn das mit a bezeichnete Individuum zur mit $\mathbf{C}$ bezeichneten Klasse gehört
			$(\mathbf{C}(a))^{\mathcal{I}} = wahr$ gdw. $a^{\mathcal{I}} \in \mathbf{C}^{\mathcal{I}}$
	Rollenbeziehung	$R(a,b)$	Wird genau dann zu wahr ausgewertet, wenn die mit R bezeichnete Rolle das mit a bezeichnete Individuum mit dem durch b bezeichneten Individuum verbindet.
			$(R(a,b))^{\mathcal{I}} = wahr$ gdw. $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$

2.2.2 Inferenzservices

Beschreibungslogische Reasoner bieten i. d. R. eine Reihe von Inferenzservices an. Mit diesen können Fragen zu Eigenschaften der Wissensbasis oder Eigenschaften bestimmter Elemente in der Wissensbasis beantwortet werden. Zu den typischsten Inferenzservices gehören nach (Hitzler u. a. 2008):

- *Klassenäquivalenz*. Entscheidet, ob in einer Wissensbasis für eine Klasse C gilt, dass sie äquivalent zu einer mit D bezeichneten Klasse ist: $C \stackrel{?}{\equiv} D$

- *Unterklassenbeziehung.* Entscheidet, ob in einer Wissensbasis für eine Klasse C gilt, dass sie einer Teilmenge der Individuen einer mit D bezeichneten Klasse entspricht: $C \stackrel{?}{\sqsubseteq} D$
- *Klassendisjunktheit.* Entscheidet, ob in einer Wissensbasis zwei Klassen C und D disjunkt sind: $C \sqcap D \stackrel{?}{\equiv} \bot$
- *Globale Konsistenz (engl. consistency).* Entscheidet, ob in einer Wissensbasis im Sinne der Prädikatenlogik widerspruchsfrei ist.
- *Klassenkonsistenz* (auch: *Klassen-* oder *Konzepterfüllbarkeit* engl. *concept satisfiability*). Entscheidet, ob für eine Klasse C einer Wissensbasis gefolgert werden kann, dass C keine Individuen zugeordnet werden können: $C \stackrel{?}{\equiv} \bot$
- *Instanzprüfung.* Entscheidet, ob ein Individuum a zur Klasse C gehört: Folgt $C(a)$ aus der Wissensbasis?

Für die in den Folgeabschnitten vorgestellte Korrektheitsanalyse gemergter Modelle werden die Inferenzservices zur Prüfung der globalen Konsistenz und der Klassenkonsistenz eingesetzt. Dabei wird das Problem, ob ein Modell korrekt hinsichtlich eines bestimmten Metamodells ist, auf die Frage nach der globalen bzw. der Klassenkonsistenz der entsprechend generierten Wissensbasis zurückgeführt. Weiterführende Erläuterungen zu den anderen Inferenzservices können (Hitzler u. a. 2008) entnommen werden. Ein wichtiger Aspekt beim beschreibungslogischen Schlussfolgern ist die Wahl zwischen Offener- bzw. Geschlossener-Welt-Semantik. Diese soll im nächsten Abschnitt erläutert werden.

2.2.3 Offene- versus Geschlossene-Welt-Semantik

Die sog. Geschlossene-Welt-Annahme (*closed world assumption – CWA*) ist das vorherrschende Paradigma im Bereich des modellbasierten Software Engineerings. Sie setzt voraus, dass jede Aussage, die nicht als wahr geschlussfolgert werden kann, falsch ist.

Dem gegenüber steht die sog. Offene-Welt-Annahme (*open world assumtion – OWA*), sie ist im Bereich der Wissensrepräsentation weit verbreitet. Die OWA setzt voraus, dass eine Aussage nur genau dann falsch ist, wenn sie als falsch geschlussfolgert werden kann. Im Bereich des logischen Schlussfolgerns auf Basis von Beschreibungslogiken wird meist von der Offenen-Welt-Semantik ausgegangen.

Für die statische Analyse der Syntax gemergter Modelle können situationsabhängig beide Paradigmen hilfreich eingesetzt werden. Um auch das Geschlossene-Welt-Paradigma für das Schlussfolgern mit Beschreibungslogiken einzusetzen, existieren mehrere Ansätze, angefangen von der Einführung epistemischer K-Operatoren in der Sprache $\mathcal{ALCK}$ (Grimm und Motik 2005) bis hin zum Vervollständigen logischer Wissensbasen (Gries 2009).

Zur Anwendung einer Geschlossenen-Welt-Semantik für bestimmte Aspekte der generierten Wissensbasis wurde in den Beispielen ab Abschnitt 2.3 das Vervollständigen der Wissensbasis im Rahmen der automatischen T- und ABox-Generierung angewendet.

2.2.4 Syntax für Beschreibungslogiken – Die Web Ontology Language (OWL)

Ähnlich den Beschreibungslogiken ist die *Web Ontology Language (OWL)* (W3C 2004) eine Familie von Sprachen aus dem Bereich der Wissensrepräsentation. Zielsetzung bei der Spezifikation von OWL war es, eine maschinenverarbeitbare Syntax zu definieren, auf deren Grundlage Reasoning-Software-Werkzeuge arbeiten können. Am weitesten verbreitet ist dabei die OWL DL bzw. OWL2 DL (W3C 2009). OWL2 DL liegt als Entwurf bei beim *World Wide Web Consortium (W3C)* als Nachfolgestandard der Ontologiesprache OWL DL vor. OWL2 DL basiert auf der Beschreibungslogik $\mathcal{SROIQ}(D)$ und ist noch ausdrucksstärker als OWL DL. Ihre Ausdrucksmächtigkeit ist eine echte Obermenge von OWL DL, welche die oben beschriebene $\mathcal{SHOIN}(D)$ als Grundlage hat.

Eine zentrale Motivation für die Entwicklung von OWL(2) DL war es, eine semantisch formal fundierte Beschreibungssprache zur Wissensrepräsentation zu entwerfen, die ein Maximum an Ausdruckstärke bietet aber dennoch logisch entscheidbar ist. Eine weitere Anforderung ist die Existenz „praktisch“ einsetzbarer Schlussfolgerungsalgorithmen für diese Sprachen.

Praktisch bedeutet in diesem Zusammenhang, dass einfache oder vielfach benötigte logische Inferenzservices (bspw. (Klassen-)Konsistenz) in für die Praxis hinreichend großen Wissensbasen sehr effizient verarbeitet werden können. Für viele Anwendungsfälle erfüllt OWL(2) unter Nutzung entsprechender Inferenzmaschinen diese Anforderungen (Hitzler u. a. 2008).

2.2.5 Inferenzwerkzeuge für Beschreibungslogiken

Die Entwicklung von Werkzeugen zum logischen Schließen in Beschreibungslogiken hat eine lange Tradition. Mit KL-ONE (Brachman und Schmolze 1985) wurde 1985 das erste System veröffentlicht. Seither entstanden eine Reihe weiterer Systeme wie LOOM (MacGregor 1991), CLASSIC (Borgida u. a. 1989), FaCT (Horrocks 1998), Racer (Haarslev und Möller 2001), die logisches Schlussfolgern in jeweils ausdruckstärkeren Beschreibungslogiken ermöglichten und hinsichtlich ihrer Performance stark optimiert wurden.

Die populärsten aktuellen Systeme in diesem Bereich sind RacerPro 2.0, KAON2 (AIFB Karlsruhe 2010), FaCT++, QuOnto (DASI-lab group - University of Rome und IBM Center for Advanced Studies (CAS) 2010) und Pellet (Parsia und Sirin 2004). Sie erlauben ein stark performanceoptimiertes Schlussfolgern auf Basis von OWL(2), wobei Pellet als einziger Reasoner OWL DL vollständig unterstützt.

Alle diese Systeme ermöglichen das automatische Schlussfolgern ausschließlich im Sinne der Offenen-Welt-Annahme (OWA). Eine umfassende Unterstützung der Geschlossenen-Welt-Annahme (CWA) im Rahmen des automatischen Schlussfolgerns ist zurzeit noch ein aktuelles Forschungsthema.

Für Ansätze wie den auf Basis des K-Operators zum Schlussfolgern mit der CWA stehen Erweiterungen für die entsprechenden Reasoning-Werkzeuge bereit (Grimm und Motik 2005). Daneben können Ansätze zum „Vervollständigen“ einer Wissensbasis eingesetzt werden, um ein Schlussfolgern nach CWA durch populäre Reasoner zu ermöglichen.

Neben der Inferenzunterstützung bieten die meisten der genannten Systeme die Auswertung von Anfragen (engl. Queries) auf Basis unterschiedlicher Anfragesprachen (SPARQL-DL, OWL-QL (Fikes und Horrocks 2003), nRQL (Haarslev, Möller und Wessel 2004) etc.) an.

2.2.6 SPARQL und Konjunktive Anfragen zur Informationsanalyse in Wissensbasen

Die Abkürzung SPARQL steht für *SPARQL Protocol and RDF Query Language*. Sie benennt eine graph-basierte Anfragesprache zur Recherche von Informationen, die in RDF repräsentiert sind und zur Darstellung der Anfrageresultate. SPARQL ist im Forschungsbereich des Semantic Web sehr populär.

Das grundlegende Konzept hinter einer sog. SPARQL-Anfrage ist die Formulierung eines (potenziell sehr komplexen) Graph-Musters, dass in einem RDF-Graphen gesucht werden soll. Darüber hinaus bietet die SPARQL-Syntax Konstrukte zur Filterung und Darstellung der Anfrageergebnisse.

SPARQL unterstützt allerdings nicht die beschreibungslogische Semantik in einer OWL-Wissensbasis sondern nur die Suche von Mustern in Graphen. Zur Verdeutlichung wird folgendes Beispiel betrachtet.

$$
\begin{aligned}
&Tra(vorfahre),\\
&kind \sqsubseteq vorfahre\\
&kind(Gustav, Stefan)\\
&kind(Stefan, Joris)\\
&kind(Stefan, Lucas)
\end{aligned}
$$

Die Wissensbasis enthält die Beziehung *kind* zwischen Eltern und ihren Kindern und die Beziehung *vorfahre* zwischen Personen und ihren Ahnen, wobei die vorfahre-Beziehung eine transitive Beziehung ist (vgl. erste Zeile). *Gustav* ist hier der Vater von *Stefan* und *Stefan* der Vater von *Joris*. Eine „einfache“ SPARQL-Anfrage

```
SELECT ?Spross WHERE {
     <http://stammbaum/Stefan> ex:kind      ?Spross
}
```

nach den Kindern von *Stefan* liefert wie erwartet *Joris* und *Lucas*. Allerdings liefert eine SPARQL-Anfrage nach den Vorfahren von *Joris*

```
SELECT ?Ahne WHERE {
     ?Ahne  ex:vorfahre  <http://stammbaum/Joris>
}
```

keine Ergebnisse, da die *vorfahre*-Beziehung nicht explizit im RDF-Graphen enthalten ist. Würde man die Semantik der Transitivität und der Subrollenbeziehung auswerten, so wären zumindestens *Stefan* und *Gustav* bekannte Vorfahren von *Joris*.

OWL-Wissensbasen beschreiben eine Menge möglicher Interpretationen. Jede dieser Interpretationen in einer Domäne kann allerdings auch als Graph aus Objekten und Beziehungen betrachtet werden. Dies eröffnet einen gewissen Spielraum zur Untersuchung beschreibungslogischer Wissensbasen nach bestimmten Informationen durch Anfragen, wie das Beispiel oben zeigt.

Um Anfragen an beschreibungslogische Wissensbasen stellen zu können, die die Semantik der spezifizierten TBox berücksichtigen, implementieren viele Reasoning-Werkzeuge das Konzept der konjunktiven Anfragen. Konjunktive Anfragen entsprechen konjunktiv verknüpften Prädikaten. Das Resultat einer konjunktiven Anfrage entspricht jeder in der Wissensbasis bekannten Belegung durch Individuen und Rollenbeziehungen, die für das Prädikat als wahr ausgewertet wird.

Für die Auswertung konjunktiver Anfragen gibt es zurzeit keine standardisierte Semantik. Bei der Benutzung entsprechender Anfragen zur Informationsgewinnung in Wissensbasen muss deswegen die jeweilig genutzte Implementierung der Query-Engine betrachtet werden.

Beachtet man die erläuterten Unterschiede in der Anwendung von SPARQL und konjunktiven Anfragen, so bieten die Anfragesprachen ein mächtiges Mittel zur Informationsgewinnung aus OWL-Wissensbasen.

2.2.7 Aktuelle Ansätze zum Einsatz von Beschreibungslogiken zur Validierung von Modellen

Die Transformation von Metamodellen in logische Ausdrücke einer Logiksprache oder als Anfragen einer Anfragesprache ist nicht neu (Deiters u. a. 2009). Insbesondere zur Analyse der Korrektheit von bzw. Konsistenz zwischen UML-Modellen wurden Beschreibungslogiken in (Berardi, Calvanese und Giacomo 2005a; van der Straeten 2005) vorgeschlagen.

In verschiedenen Ansätzen wird das UML-Metamodell als terminologisches Wissen (TBox) einer Wissensbasis ausgedrückt und die zu prüfenden UML-Modelle als Faktenwissen (ABox) wie in Abbildung IV.12 dargestellt. Durch eine derartige Repräsentation von Metamodell- und Modellinformationen kann mithilfe der Konsistenzanalyse eines beschreibungslogischen Reasoners die Frage nach der Existenz von Widersprüchen zwischen dem TBox- und dem ABox-Teil der Wissensbasis entschieden werden.

Werden Widersprüche erkannt, so ist das Modell im Sinne der Spezifikation seiner formalen Syntax im Metamodell nicht korrekt.

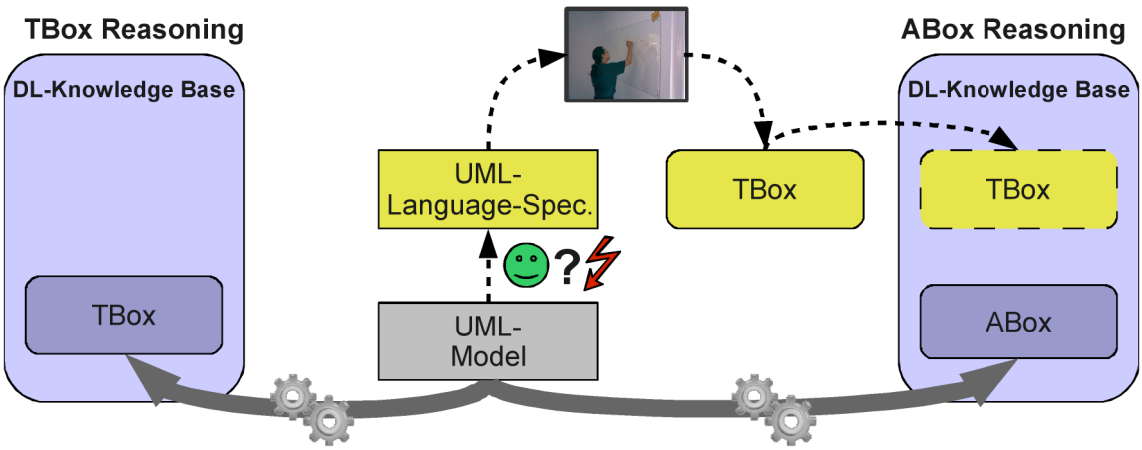


Abbildung IV.12. Validierung von UML-Modellen auf Basis von Beschreibungslogiken

Die vorhandenen Ansätze zum Einsatz von Beschreibungslogiken in der statischen Analyse von Modellen, setzen die *manuelle* Modellierung der Metamodellinformationen einer Modellierungssprache als Domänenwissen (TBox) einer Beschreibungslogik voraus (vgl. Abbildung IV.12). Metamodelle von Modellierungssprachen im Bereich des modellbasierten Software Engineering sind aber i. d. R. auf Basis der MOF spezifiziert und nicht durch beschreibungslogische Domänenbeschreibungen.

Es ist leicht nachzuvollziehen, dass eine manuelle Erstellung einer TBox für jedes per MOF/OCL spezifizierte Metamodell der jeweils zu verwendenden Modellierungssprache nicht praktikabel ist, da das Ausdrücken hinreichend großer Metamodelle durch logische Mengenrelation sehr aufwendig ist. Eine automatische Erstellung einer TBox aus einem durch die MOF/OCL spezifizierten Metamodell wäre zur Lösung dieses Problems hilfreich (vgl. $\mathcal{I}^{TBox}_{MMR^{ECore}}$ in Abbildung IV.11).

Ein Schritt in diese Richtung wurde mit dem Projekt IODT/EODM (IBM alphaWorks 2007) verfolgt. Im Rahmen dieses Projektes wurde ein erstes Entwicklungs-Toolkit zur Transformation von per ECore spezifizierten EMF-Modellen nach OWL entwickelt.

2.3 Alternative 1: Statische Analyse durch ABox-Reasoning mit OWL im Rahmen der Modellvalidierung

In Abbildung IV.11 wurde überblicksartig ein Konzept zur Validierung von Modellversionen vorgestellt, die nach einem Merge als morphologische Modellrepräsentation vorliegen. Grundidee des Verfahrens ist die Transformation von Modell und Metamodell in eine formale Logiksprache (OWL/SPARQL-DL) und eine anschließende automatische Durchführung der statischen Analyse mittels Reasoning-Services bzw. Anfrageauswertung. Dabei können bereits verfügbare Softwarewerkzeuge aus dem Bereich der Wissensverarbeitung eingesetzt werden.

Von den erwähnten Validierungsmöglichkeiten zur statischen Analyse (Abbildung IV.11, Stage 2) wird in diesem Abschnitt die Modellvalidierung basierend auf dem ABox-Reasoning-Service zur Konsistenzprüfung einer Wissensbasis erläutert. Dazu geht der folgende Unterabschnitt auf die Transformationen zur Erzeugung von TBox und ABox aus den entsprechenden MMRs von Modell und Metamodell ein. Anschließend wird die Anwendung eines OWL-Reasoners zur Identifizierung von merge-bedingten Fehlern in einem Modell beschrieben.

Zur Illustration der Analyse wird das Beispiel aus Abbildung IV.10 herangezogen. In diesem Beispiel wird die Kompositionsbeziehung zwischen *UseCase* und *System* im RE-Metamodell (vgl. Abbildung IV.13) durch die Integration paralleler Änderungen verletzt.

2.3.1 Transformation von Metamodellen und Modellen nach OWL

Wie in Abbildung IV.11 gezeigt wird, muss zur Anwendung des ABox-Reasoning aus der MMR des Metamodells mithilfe einer Interpretationsabbildung $\mathcal{I}^{TBOX}_{MMR^{ECore}}$ eine TBox und aus der MMR des Modells durch $\mathcal{I}^{TBOX}_{MMR^{model}}$ eine ABox erzeugt werden. Die Erläuterung dieser beiden Interpretationsabbildungen am Beispiel ist Gegenstand der folgenden Unterabschnitte.

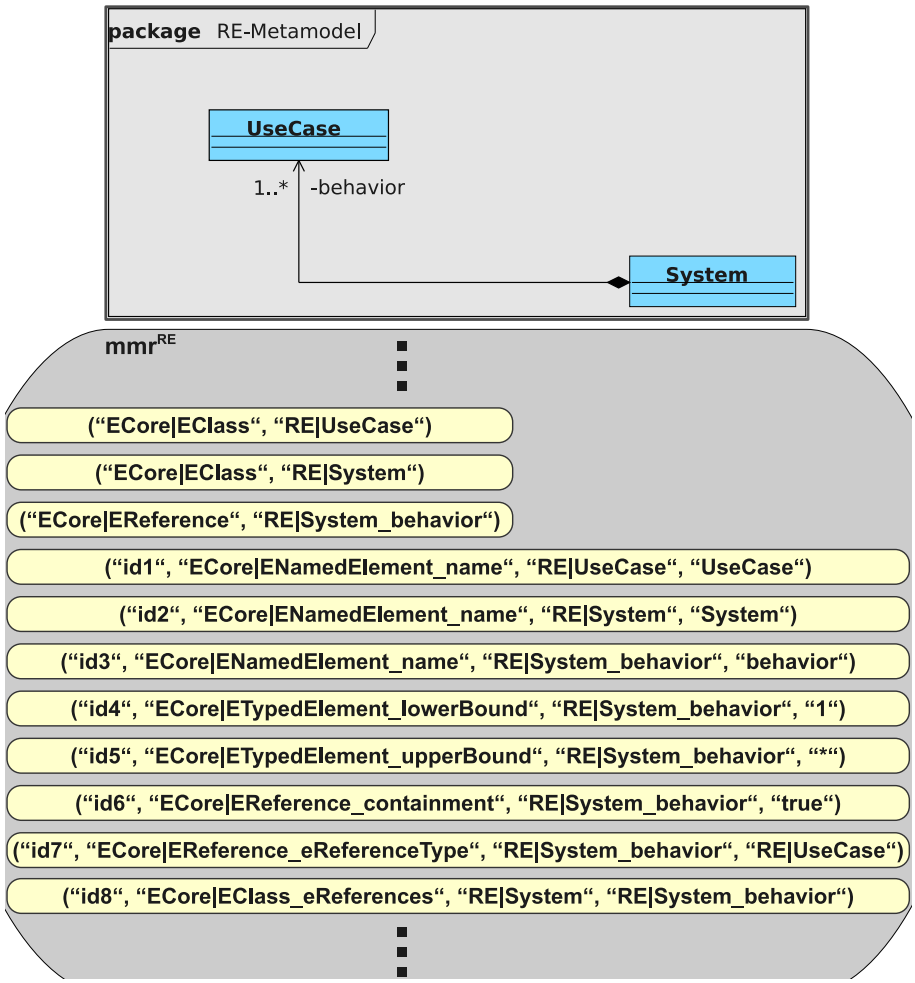


Abbildung IV.13. Ausschnitt aus dem RE-Metamodell (UML und MMR repräsentiert)

Interpretation von Metamodellen als beschreibungslogische TBox

Metamodelle (MOF-Ebene M2) werden im modellbasierten Software Engineering i. d. R. als MOF-Modelle beschrieben. Das MOF-Meta-Metamodell (MOF-Ebene M3) ist somit das Metamodell für Metamodelle von Modellierungssprachen (MOF-Ebene M2) (vgl. Abschnitt II.3.3.2). Da morphologische Dateneinheiten sowohl Modell- als auch Metamodellelemente referenzieren, muss dies bei der Erzeugung der MMR für ein Metamodell berücksichtigt werden.

Zur Illustration der Transformation der MMR eines Metamodells in eine entsprechende OWL-TBox wird anstelle der MOF die *Essential MOF* (Abk. *EMOF*) verwendet. Diese wird durch die Sprache ECore implementiert, deren Metamodell in Abbildung II.15 abgebildet wurde.

Zur Darstellung der Transformation einer MMR in eine beschreibungslogische TBox wird ein Ausschnitt des RE-Metamodells (vgl. Abbildung III.5) in Abbildung IV.13 als MMR dargestellt, der den Fehler in Abbildung IV.10 begründet. Wie man der Abbildung entnehmen kann, ist das RE-Metamodell eine Instanz des ECore-Metamodells aus Abbildung II.15.

Anschließend wird nun die Transformation dieses Metamodellausschnittes anhand der in Interpretation IV.1 dargestellten Abbildung erläutert. Interpretation IV.1 stellt die oben motivierte Abbildung zur Umwandlung der MMR eines Metamodells, das auf Basis von ECore modelliert wurde, in eine beschreibungslogische TBox dar:

Interpretation IV.1: Durch die Abbildung $\mathcal{I}^{TBox}_{MMR^{ECore}} : \wp(MMR^{ECore}) \rightarrow TBox^{\mathcal{SHOIN}}$ werden ECore-Modelle (Metamodelle einer beliebigen Modellierungssprache) in eine beschreibungslogische TBox transformiert. Nachfolgend wird ein zentraler Ausschnitt dieser Abbildung dargestellt:

$$
\begin{array}{rcl}
(\{(\text{"ECore|EClass"}, id^{class})\} \subseteq mmr^m)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & id^{class} \in \mathcal{C} \\
(\{(\text{"ECore|EReference"}, id^{ref})\} \subseteq mmr^m)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & id^{ref} \in \mathcal{R} \\
(\{(\text{"ECore|EAttribute"}, id^{attr})\} \subseteq mmr^m)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & id^{attr} \in \mathcal{R} \\
 & \Rightarrow & \text{containment} \in \mathcal{R} \\
\left(\left\{ \begin{array}{l} (i, \text{"ECore|ETypedElement_upperBound"}, id^{ref}, ub), \\ (i, \text{"ECore|EClass_eReferences"}, id^{class}, id^{ref}) \end{array} \right\} \subseteq mmr^m, ub \neq \text{" *"} \right)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & \text{"}id^{class} \sqsubseteq \leq ub\, id^{ref}\text{"} \in \text{TBox} \\
\left(\left\{ \begin{array}{l} (i, \text{"ECore|TypedElement_lowerBound"}, id^{ref}, lb), \\ (i, \text{"ECore|EClass_eReferences"}, id^{class}, id^{ref}) \end{array} \right\} \subseteq mmr^m \right)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & \text{"}id^{class} \sqsubset \geq lb\, id^{ref}\text{"} \in \text{TBox} \\
(\{(i, \text{"ECore|EReference_eReferenceType"}, id^{ref}, id^{class})\} \subseteq mmr^m)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & \text{"}\top \sqsubseteq \forall id^{ref}.id^{class}\text{"} \in \text{TBox} \\
(\{(i, \text{"ECore|EClass_eReferences"}, id^{class}, id^{ref})\} \subseteq mmr^m)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & \text{"}\exists id^{ref}.\top \sqsubseteq id^{class}\text{"} \in \text{TBox} \\
\left(\left\{ \begin{array}{l} (i, \text{"ECore|EReference_containment"}, id^{ref}, true), \\ (i, \text{"ECore|EReference_eReferenceType"}, id^{ref}, id^{class}) \end{array} \right\} \subseteq mmr^m \right)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & \left\{ \begin{array}{l} \text{container_}id^{class} \in \mathcal{R}, \\ \text{containment_}id^{class} \in \mathcal{R}, \\ \text{"}\top \sqsubseteq \leq 1 \text{container_}id^{class}\text{"}, \\ \text{"}id^{ref^-} \sqsubseteq \text{container_}id^{class}\text{"} \\ \text{"}id^{ref} \sqsubseteq \text{containment_}id^{class}\text{"} \\ \text{containment_}id^{class} \sqsubseteq \text{containment"} \end{array} \right\} \subseteq \text{TBox} \\
(\{(i, \text{"ECore|EClass_eSuperTypes"}, id^{class1}, id^{class2})\} \subseteq mmr^m)^{\mathcal{I}^{TBox}_{MMR^{ECore}}} & \Rightarrow & \text{"}id^{class1} \sqsubseteq id^{class2}\text{"} \in \text{TBox} \\
\vdots & &
\end{array}
$$

Durch die Interpretation IV.1 wird für einen Ausschnitt aus der EMOF, wie sie durch die OMG informell im MOF-Metamodell beschrieben ist, die Umwandlung von M2-Metamodellen in beschreibungslogische Terme ausgedrückt. Die Interpretation IV.1 zeigt dabei die Darstellung von Klassen und Generalisierungen, Referenzen mit optionalen Kompositionseigenschaften und Multiplizitäten.

Die ersten drei Zeilen beschreiben die Abbildung von Typzuweisungen auf Klassen (ID: *ECore|EClass*), Referenzen (ID: *ECore|EReference*) und Attributen (ID: *ECore|EAttribute*) von ECore, in Klassennamen bzw. Rollennamen der erzeugten TBox. Dazu werden die Identifizierer der *instance*-Referenz der Typzuweisung (vgl. Abbildung IV.2 und Definition IV.6) als Namen für die jeweiligen Klassen und Rollen verwendet.

Die Eigenschaftszuweisungen in Zeile vier bis neun bilden die Grundlage zur Bestimmung der logischen Ausdrücke in der zu erstellenden TBox. In der vierten und fünften Zeile wird die Interpretation von Multiplizitäten als TBox-Ausdrücke dargestellt. Zeile 6 und 7 stellt die Interpretation von Eigenschaftszuweisungen der Rollen *ECore|EReference_eReferenceType* und *ECore|EReference_eReferences* als Werte- und Definitionsbereich der eingeführten Relationen dar (vgl. Abbildung IV.2 und Definition IV.8).

Diese entsprechen somit den Referenzen und Attributen des ECore-Modells. Ist im ECore-Modell für eine Referenz das Composite-Attribut mit *TRUE* belegt, so werden die

Eigenschaften der entsprechenden Relation in der TBox angepasst. Dazu wird in Zeile 8 festgelegt, dass Objekte eines dem Metamodell entsprechenden Modells in maximal einem anderen Objekt enthalten sein dürfen. Dazu werden entsprechende Hilfsrollen – $containment_id^{class}$ und $container_id^{class}$ – pro beinhalteter Klasse (id^{class}) eingeführt. In der letzten dargestellten Zeile wird die Vererbungsbeziehung auf die beschreibungslogische Teilmengenbeziehung abgebildet.

Der in Interpretation IV.1 dargestellte Ausschnitt lässt sich analog erweitern, sodass alle Modellelemente als Instanzen des in Abbildung II.15 dargestellten ECore-Metamodells in einer TBox semantisch äquivalent ausgedrückt sind. Zur Darstellung der Modellvalidierung im Hinblick auf das Beispiel in Abbildung IV.10 genügt an dieser Stelle allerdings der dargestellte Ausschnitt der Interpretationsabbildung.

Wendet man die durch Interpretation IV.1 gegebene semantische Abbildung auf den in Abbildung IV.13 dargestellten Ausschnitt aus der MMR des RE-Metamodells an, so ergibt sich eine TBox für das RE-Metamodell, wie sie in Auflistung IV.1 abgebildet ist.

Auflistung IV.1: TBox des RE-Metamodells (Ausschnitt)

$$\mathcal{I}^{TBox}_{MMR^{ECore}}\left(mmr^{RE}\right) = \left\{ \begin{array}{ll} \mathbf{RE|UseCase} & \\ \mathbf{RE|System} & /* \text{ Klassennamen } */ \\ & \\ \mathbf{RE|System_behavior} & \\ \mathbf{container_RE|UseCase} & \\ \mathbf{containment_RE|UseCase} & /* \text{ Rollennamen } */ \\ \mathbf{containment} & \\ & \\ \top \sqsubseteq \forall \mathbf{RE|System_behavior}.\mathbf{RE|UseCase} & \\ \exists \mathbf{RE|System_behavior}.\top \sqsubseteq \mathbf{RE|System} & \\ \mathbf{RE|System} \sqsubseteq \geq 1\, \mathbf{RE|System_behavior} & \\ \top \sqsubseteq \leq 1\, \mathbf{container_RE|UseCase} & \\ \mathbf{RE|System_behavior}^{-} \sqsubseteq \mathbf{container_RE|UseCase} & \\ \mathbf{RE|System_behavior} \sqsubseteq \mathbf{containment_RE|UseCase} & /* \text{ Klassenaxiome } */ \\ \mathbf{containment_RE|UseCase} \sqsubseteq \mathbf{containment} & \\ & \\ \mathbf{RE|UseCase} \sqcap \mathbf{RE|System} \sqsubseteq \bot & \\ \mathbf{RE|UseCase} \sqcup \mathbf{RE|System} \sqcup \ldots \sqsubseteq \top & \\ \vdots & \end{array} \right\}$$

Mit der erzeugten TBox-Repräsentation des RE-Metamodells lassen sich nun die Syntax beliebiger Modelle, die zu diesem Metamodell konform sein muss, im Rahmen der Modellvalidierung statisch analysieren. Dazu kann, wie in Abschnitt 2.2 detaillierter erläutert, ein Reasoner auf Basis von OWL eingesetzt werden. Drückt man die beschreibungslogischen Terme in Auflistung IV.1 in OWL-Syntax aus, so stellt sich das Ergebnis wie in Abbildung IV.14 dar.

Nachdem gezeigt wurde, wie das Metamodell einer Modellierungssprache als TBox in OWL abgebildet werden kann, muss nun das zu prüfende Modell in eine beschreibungslogische ABox transformiert werden. Dieser Vorgang ist Thema des nächsten Unterabschnittes und ermöglicht die anschließende Validierung auf Basis der Konsistenzprüfung einer Wissensbasis.

```
.
.
.
<owl:Class rdf:about="RE#UseCase"/>
<owl:Class rdf:about="RE#System"/>

<owl:ObjectProperty rdf:about="RE#System_behavior"/>
<owl:ObjectProperty rdf:about="RE#containment_REUseCase"/>
<owl:ObjectProperty rdf:about="RE#container_REUseCase"/>


<owl:ObjectProperty rdf:about="RE#System_behavior">
  <rdfs:range rdf:resource="RE#UseCase"/>
  <rdfs:domain rdf:resource="RE#System"/>
  <rdfs:subPropertyOf rdf:resource="RE#containment_REUseCase"/>
</owl:ObjectProperty>


<owl:Class rdf:about="RE#System">
  <rdfs:subClassOf>
    <owl:Restriction>
     <owl:onProperty rdf:resource="RE#System_behavior"/>
     <owl:minCardinality rdf:datatype="xsd:nonNegativeInteger"
     >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="www.w3.org/2002/07/owl#Thing">
  <rdfs:subClassOf>
    <owl:Restriction>
     <owl:onProperty rdf:resource="RE#container_REUseCase"/>
     <owl:maxCardinality rdf:datatype="xsd:nonNegativeInteger"
     >1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty>
  <rdfs:subPropertyOf rdf:resource="RE#container_REUseCase"/>
  <owl:inverseOf rdf:resource="RE#System_behavior"/>
</owl:ObjectProperty>


<owl:Class rdf:about="www.w3.org/2002/07/owl#Nothing">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="RE#UseCase"/>
    <owl:Class rdf:about="RE#System"/>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:about="www.w3.org/2002/07/owl#Thing">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="RE#UseCase"/>
    <owl:Class rdf:about="RE#System"/>
  </owl:unionOf>
</owl:Class>
.
.
.
```

Abbildung IV.14. OWL-Repräsentation – RE-Metamodell-Ausschnitt (vgl. Auflistung IV.1)

Interpretation von Modellen als beschreibungslogische ABox

Die im letzten Abschnitt vorgestellte beschreibungslogische Interpretation der MMR eines Metamodells in terminologisches Wissen einer Wissensbasis (TBox) bildet eine Grundlage für Validierung gemergter Modelle. Um diese durchführen zu können, muss die zu prüfende Modellversion in einer beschreibungslogischen ABox vorliegen. Die Interpretation $\mathcal{I}^{ABox}_{MMR^{model}}$ (vgl. Abbildung IV.11), die ein beliebiges in MMR vorliegendes Modell in eine beschreibungslogische ABox umwandelt, wird in diesem Abschnitt genauer erläutert.

Die motivierte Interpretation ist wie im letzten Abschnitt als Transformationsabbildung auf Basis der MMR beschrieben und in Interpretation IV.2 dargestellt. Sie ist weniger komplex als das Pendant zur TBox-Erzeugung aus Metamodellen. Die Abbildung lässt sich für Modelle beliebiger Modellierungssprachen, die in MMR repräsentiert sind, anwenden.

Interpretation IV.2: Abbildung $\mathcal{I}^{ABox}_{MMR^m}$

$$
\begin{array}{rcl}
\left((i, id^{type}_{\hat{m}}, id^{instance}_{m}) \in TypeAssertion^m\right) & \Rightarrow & id^{type}_{\hat{m}}(id^{instance}_{m}) \in \text{ABox} \\
\left((i, id^{role}_{\hat{m}}, id^{source}_{m}, id^{target}_{m}) \in PropertyAssertion^m\right) & \Rightarrow & id^{role}_{\hat{m}}(id^{source}_{m}, id^{target}_{m}) \in \text{ABox} \\
\left(\begin{array}{l}(i_1, id^{role}_{\hat{m}}, id^{source}_{m}, id^{target_1}_{m}) \in PropertyAssertion^m \\ (i_2, id^{role}_{\hat{m}}, id^{source}_{m}, id^{target_2}_{m}) \in PropertyAssertion^m \\ (i_1, i_2) \in PropertyOrder^m\end{array}\right) & \Rightarrow & \left\{\begin{array}{l}\text{order_}id^{role}_{\hat{m}}\text{_}id^{source}_{m}(id^{target_1}_{m}, id^{target_2}_{m}), \\ \text{order_}id^{role}_{\hat{m}}\text{_}id^{source}_{m}(\text{Pred}, \text{Succ})\end{array}\right\} \subseteq \text{ABox}
\end{array}
$$

Aus jeder Typzuweisung in der MMR eines Modells wird eine entsprechende Zuweisung zwischen Klasse und Individuum in der ABox erzeugt (vgl. Zeile 1, Interpretation IV.2). In der zweiten Zeile wird dargestellt, wie aus jeder Eigenschaftszuweisung der MMR eine Rollenbeziehung in der ABox erzeugt wird.

Die dritte Zeile zeigt die Repräsentation der Ordnungsbeziehungen, falls es mehrere Ausprägungen eines Eigenschaftstyps für ein Objekt gibt. Dazu wird eine neue Rolle (oben Rollenname: "$\text{order_}id^{role}_{\hat{m}}\text{_}id^{source}_{m}$) zur Abbildung der Ordnung spezifisch für Eigenschaftstyp ($id^{role}_{\hat{m}}$) und Objekt ($id^{source}_{\hat{m}}$) erstellt. Zur Illustration der Interpretation IV.2 wird wiederum das Beispiel aus Abbildung IV.10 genutzt. Dazu stellt Abbildung IV.14 die MMR des zu transformierenden Modellausschnitts dar.

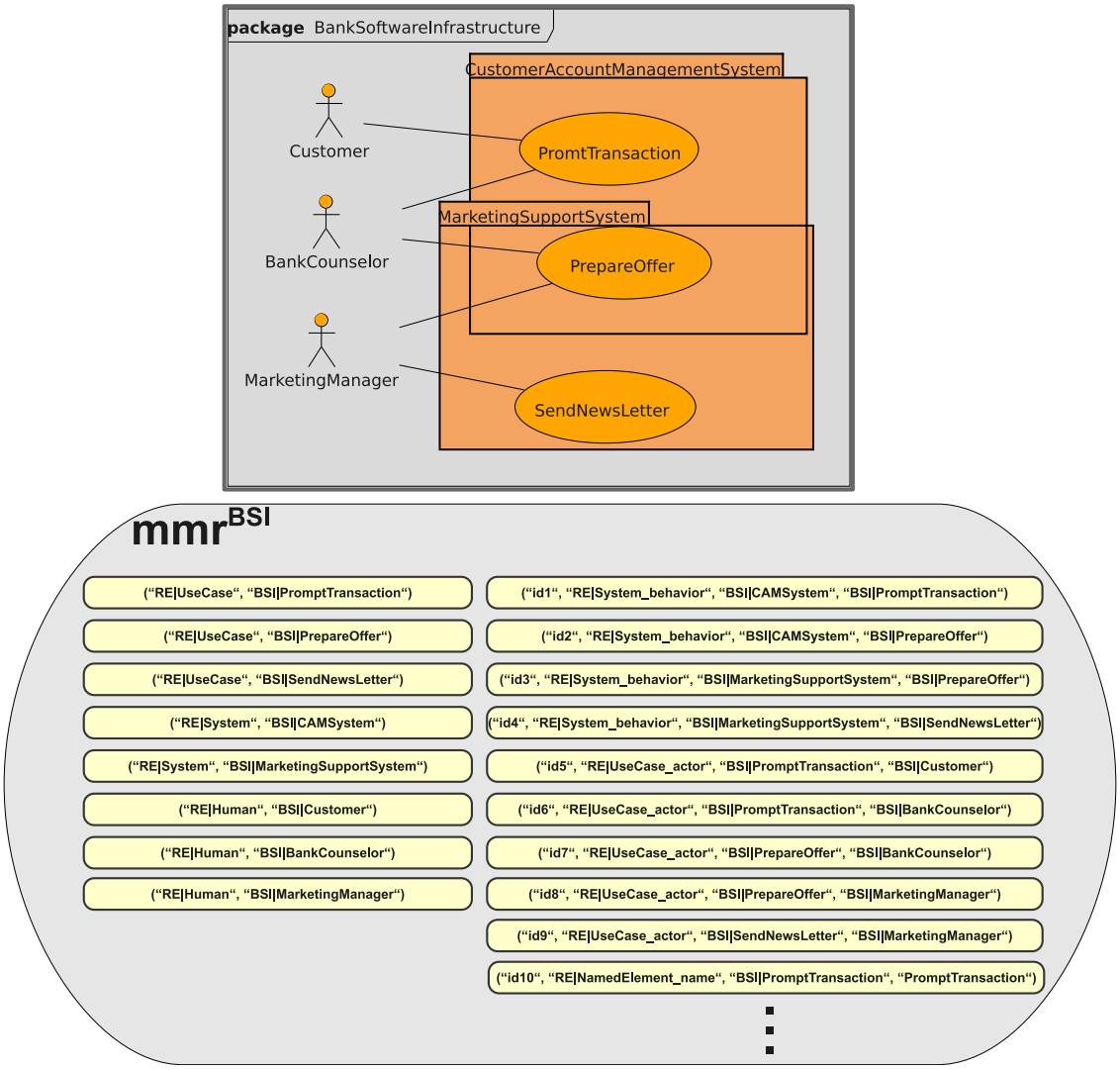


Abbildung IV.15. MMR von Ausschnitt (vgl. Abbildung IV.10) des Modells BankSoftwareInfrastructure

Wendet man die durch Interpretation IV.2 beschriebene Transformation auf die MMR in Abbildung IV.14 an, so erhält man die in Auflistung IV.2 abgebildete ABox. Diese kann wie in Abbildung IV.15 dargestellt in OWL repräsentiert werden. Die ABox aus Auflistung IV.2 und die TBox aus Auflistung IV.1 bilden zusammen eine Wissensbasis, die Aufschluss über die Korrektheit gemergter Modelle geben kann (hinsichtlich Syntax und statischer Semantik). Auf die dafür notwendige, automatisierte Analyse geht der folgende Abschnitt ein.

Auflistung IV.2: ABox des Modells „BankSoftwareInfrastructure – BSI“ (Ausschnitt)

$$
\mathcal{I}^{ABox}_{MMR^m}\,(mmr^{BSI}) = \left\{
\begin{array}{l}
\mathbf{RE|UseCase(BSI|PromptTransaction)},\\
\mathbf{RE|UseCase(BSI|PrepareOffer)},\\
\mathbf{RE|UseCase(BSI|SendNewsLetter)},\\
\mathbf{RE|System(BSI|CAMSystem)},\\
\mathbf{RE|System(BSI|MarketingSupportSystem)},\\
\mathbf{RE|Human(BSI|Customer)},\\
\mathbf{RE|Human(BSI|BankCounselor)},\\
\mathbf{RE|Human(BSI|MarketingManager)}\\
\\
\mathbf{RE|System_behavior(BSI|CAMSystem, BSI|PromptTransaction)},\\
\mathbf{RE|System_behavior(BSI|CAMSystem, BSI|PrepareOffer)},\\
\mathbf{RE|System_behavior(BSI|MarketingSupportSystem, BSI|PrepareOffer)},\\
\mathbf{RE|System_behavior(BSI|MarketingSupportSystem, BSI|SendNewsLetter)}\\
\mathbf{RE|System_actor(BSI|PromptTransaction, BSI|Customer)},\\
\mathbf{RE|System_actor(BSI|PromptTransaction, BSI|BankCounselor)},\\
\mathbf{RE|System_actor(BSI|PrepareOffer, BSI|BankCounselor)},\\
\mathbf{RE|System_actor(BSI|PrepareOffer, BSI|MarketingManager)},\\
\mathbf{RE|System_actor(BSI|SendNewsLetter, BSI|MarketingManager)},\\
\mathbf{RE|NamedElement_name(BSI|PromptTransaction, PromptTransaction)},\\
\vdots
\end{array}
\right.
$$

```
...
<owl:Class rdf:about="http://www.w3.org/2002/07/owl#Thing">
  <owl:oneOf rdf:parseType="Collection">
    <RE:UseCase rdf:about="http://BSI#PromptTransaction"/>
    <RE:UseCase rdf:about="http://BSI#PrepareOffer"/>
    <RE:UseCase rdf:about="http://BSI#SendNewsLetter"/>
    <RE:System rdf:about="http://BSI#CAMSystem"/>
    <RE:System rdf:about="http://BSI#MarketingSupportSystem"/>
    <RE:Human rdf:about="http://BSI#Customer"/>
    <RE:Human rdf:about="http://BSI#BankCounselor"/>
    <RE:Human rdf:about="http://BSI#MarketingManager"/>
 </owl:oneOf>
</owl:Class>

<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <RE:UseCase rdf:about="http://BSI#PromptTransaction"/>
    <RE:UseCase rdf:about="http://BSI#PrepareOffer"/>
    <RE:UseCase rdf:about="http://BSI#SendNewsLetter"/>
    <RE:System rdf:about="http://BSI#CAMSystem"/>
    <RE:System rdf:about="http://BSI#MarketingSupportSystem"/>
    <RE:Human rdf:about="http://BSI#Customer"/>
    <RE:Human rdf:about="http://BSI#BankCounselor"/>
    <RE:Human rdf:about="http://BSI#MarketingManager"/>
  </owl:distinctMembers>
</owl:AllDifferent>

<RE:System rdf:about="http://BSI#CAMSystem">
  <RE:System_behavior rdf:resource="http://BSI#PromptTransaction"/>
</RE:System>
<RE:System rdf:about="http://BSI#CAMSystem">
  <RE:System_behavior rdf:resource="http://BSI#PrepareOffer"/>
</RE:System>
<RE:System rdf:about="http://BSI#MarketingSupportSystem">
  <RE:System_behavior rdf:resource="http://BSI#PrepareOffer"/>
</RE:System>
<RE:System rdf:about="http://BSI#MarketingSupportSystem">
  <RE:System_behavior rdf:resource="http://BSI#SendNewsLetter"/>
</RE:System>
<RE:UseCase rdf:about="http://BSI#PromptTransaction">
  <RE:UseCase_actor rdf:resource="http://BSI#Customer"/>
</RE:UseCase>
<RE:UseCase rdf:about="http://BSI#PromptTransaction">
  <RE:UseCase_actor rdf:resource="http://BSI#BankCounselor"/>
</RE:UseCase>
<RE:UseCase rdf:about="http://BSI#PrepareOffer">
  <RE:UseCase_actor rdf:resource="http://BSI#BankCounselor"/>
</RE:UseCase>
<RE:UseCase rdf:about="http://BSI#PrepareOffer">
  <RE:UseCase_actor rdf:resource="http://BSI#MarketingManager"/>
</RE:UseCase>
<RE:UseCase rdf:about="http://BSI#SendNewsLetter">
  <RE:UseCase_actor rdf:resource="http://BSI#MarketingManager"/>
</RE:UseCase>
<RE:UseCase rdf:about="http://BSI#PromptTransaction">
  <RE:NamedElement_name rdf:datatype="xsd:String">
    PromptTransaction
  </RE:NamedElement_name>
</RE:UseCase>
...
```

Abbildung IV.16. OWL-Repräsentation – BSI-Modell-Ausschnitt (vgl. Auflistung IV.2)

2.3.2 Statische Analyse durch Konsistenzprüfung von OWL-Wissensbasen

Nach Erzeugung der beschreibungslogischen Wissensbasis aus den morphologischen Repräsentationen von Modell und Metamodell kann ein Modell auf Basis seiner Syntax durch Softwarewerkzeuge validiert werden.

Dazu wird die Frage nach der syntaktischen Korrektheit eines Modells (incl. statischer Semantik) auf die Frage nach der Konsistenz einer Wissensbasis reduziert. Um diese zu beantworten, wird der ABox-Reasoning-Service zur Prüfung der globalen Konsistenz (vgl. Abschnitt 2.2.2) eingesetzt. Mit diesem ist automatisch entscheidbar, ob sich ein Widerspruch aus einer Wissensbasis schließen lässt.

Falls das Modell, das als ABox repräsentiert ist, nicht den Anforderungen des als TBox repräsentierten Metamodells entspricht, wird die Inkonsistenz der generierten Wissensbasis festgestellt. Zum automatischen Schlussfolgern von Konsistenz bzw. Inkonsistenz können die in Abschnitt 2.2.5 erwähnten Reasoning-Werkzeuge eingesetzt werden.

Wie in Abschnitt III.4.4.1 schon erwähnt, ist das Merge-Ergebnis, das im Modellausschnitt Abbildung III.18 dargestellt ist, inkorrekt. Begründet wird dies durch die Kompositionsbeziehung zwischen *RE|System* und *RE|UseCase* im Metamodell (vgl. Metamodellausschnitt in Abbildung III.19). Mit dem OWL-Reasoner Pellet kann dieser merge-bedingte Fehler automatisch erkannt werden. Die Ausgabe des Pellet-Reasoners bei der Analyse einer Wissensbasis (engl. knowlegde base, Abk. KB) bestehend aus den Ausdrücken aus Auflistung IV.1 und Auflistung IV.2 ist folgende:

Auflistung IV.3. Ausgabe des Reasoners bei inkonsistenter Wissensbasis → Modellinkorrektheit

```
  17.09.2010 09:34:21 org.mindswap.pellet.KnowledgeBase consistency

  WARNUNG: Inconsistent ontology. Reason: Individual http://PrepareOffer has more than 1
values for property http://anonymous.de#container_REUseCase violating the cardinality
restriction

  Error (KB is inconsistent!): Individual http://PrepareOffer has more than 1 values for
property http://anonymous.de#container_REUseCase violating the cardinality restriction



  fertig!
```

Der Anwendungsfall *PrepareOffer* bestimmt als Verhalten (engl. *behavior*) von zwei *Systemen*, die im Modell spezifiziert sind. Zur Lösung des Problems müssen die an den ursächlichen Änderungen beteiligten Modellierer das inkorrekte Modell dahin gehend ändern, dass jeder Anwendungsfall nur einem System zugeordnet ist. Dabei stellt sich die Frage nach einer geeigneten Darstellung des Fehlers im Modell.

Die Darstellung des Fehlers wie im Zwischenergebnis *VM* in Abbildung III.18 kann nicht automatisch erzeugt werden. Dafür wäre Wissen zur grafischen Repräsentationsform von Modellen der jeweiligen Modellierungssprache und zur intuitiven Darstellbarkeit bestimmter

Fehler notwendig. Eine Möglichkeit den Modellierern trotzdem die Fehlerursache zu verdeutlichen, ist die Markierung der betreffenden parallelen Änderungen in den parallel erzeugten Versionen, wie in Abbildung IV.17 dargestellt (vgl. auch (Bartelt und Schindler 2010)).

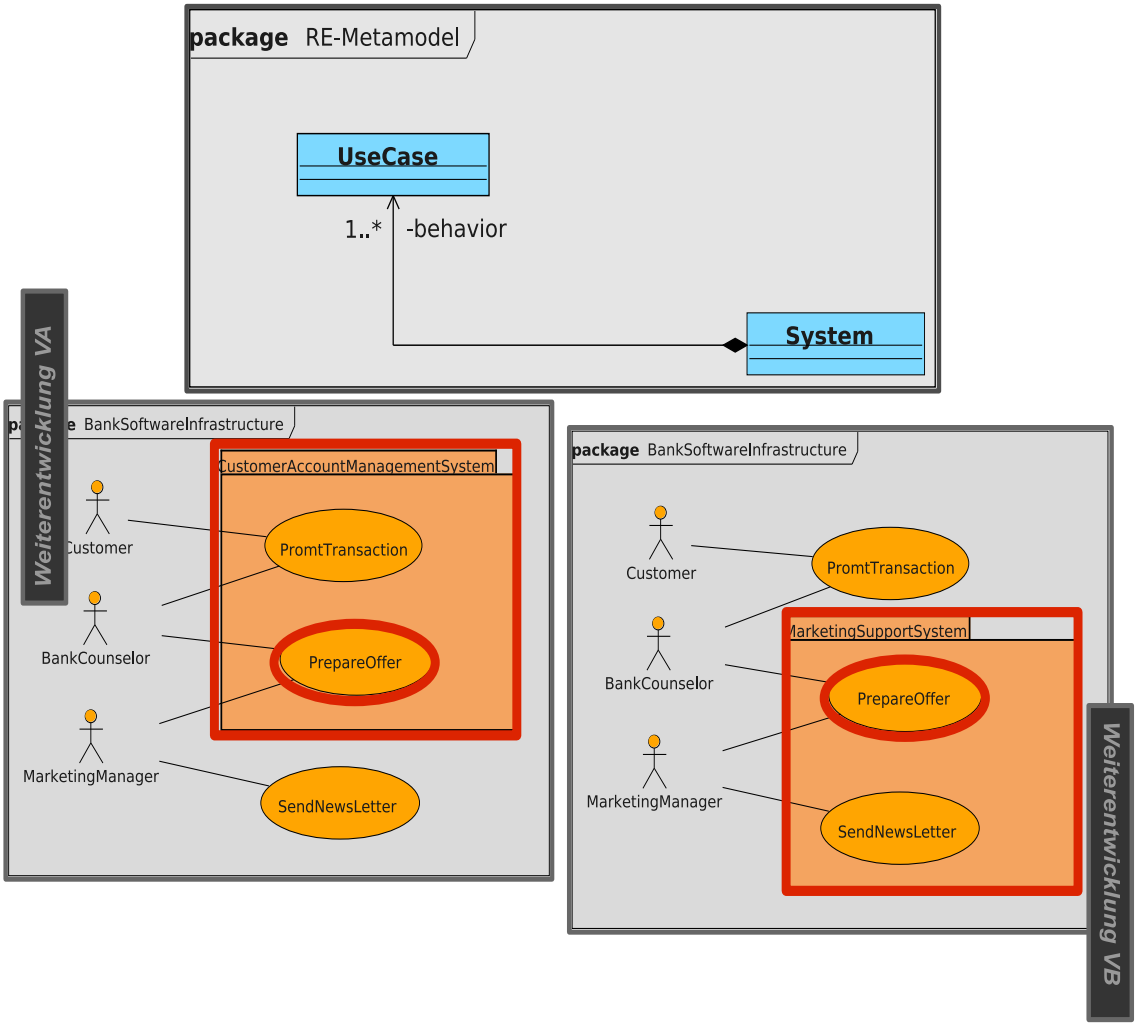


Abbildung IV.17. Darstellung merge-bedingter Fehler in parallelen Versionen

Abbildung IV.17 zeigt die Markierung der Modellelemente, die für den Fehler in Zwischenversion *VM* verantwortlich sind. Beide Modellierer haben in den betreffenden Versionen *VA* und *VB* den *Anwendungsfall PrepareOffer* als Verhalten jeweils unterschiedlicher *Systeme* spezifiziert. Durch die Markierungen in den parallelen Modellversionen und der Anzeige des entsprechend „verletzten“ Metamodellausschnittes ist die Fehlerursache offensichtlich.

Die Fehlerdarstellung in Abbildung IV.17 ist sicherlich nur eine Alternative zur Unterstützung der Modellierer bei der Korrektur merge-bedingter Fehler. Auf Darstellungsalternativen soll an dieser Stelle nicht weiter im Detail eingegangen werden, diese eignen sich als Thema für weitere Forschungsarbeit.

Bei einer Korrektur könnten sich die Modellierer dazu entscheiden, dass der *UseCase PrepareOffer* ausschließlich dem *MarketingSupportSystem* zugeordnet wird wie auch in der Weiterentwicklung der Version *VB*. Die Modellversion *VM* wäre nach dieser Entscheidung korrekt bzgl. des Metamodells. Bei einer wiederholten Prüfung durch den OWL-Reasoner werden dann keine Fehler mehr festgestellt. Die Ausgabe des Pellet-Reasoners ist dementsprechend leer, wenn auch keine weiteren Fehler mehr im Modell vorhanden sind:

Auflistung IV.4. Ausgabe des Reasoners bei konsistenter Wissensbasis

```
<EMPTY>

fertig!
```

2.3.3 Einschränkungen der statischen Modellanalyse durch Konsistenzprüfung im Rahmen des ABox-Reasonings

Bei der Prüfung der globalen Konsistenz bestimmt der Reasoner, ob ein Widerspruch aus den logischen Ausdrücken der Wissensbasis automatisch ableitbar ist. Gelingt das nicht, so ist die Wissensbasis konsistent. Konnte ein Widerspruch geschlossen werden, so bricht der Reasoner mit einer wie im letzten Abschnitt dargestellten Fehlermeldung ab. Da sich aus einem Widerspruch in einem logischen System alles schließen lassen würde, ist dieses Verhalten nur konsequent.

Verletzt ein Modell aber verschiedene Teile der Metamodellspezifikation (sind bspw. mehrere merge-bedingte Fehler vorhanden), so wird durch die Konsistenzanalyse zunächst nur auf den zuerst erkannten Fehler hingewiesen. Um welchen der Fehler es sich dabei handelt, kann nicht bewusst durch die Modellierer beeinflusst werden.

Erst nach Korrektur des zuletzt erkannten Fehlers können durch wiederholte Konsistenzprüfungen weitere Fehler automatisch erkannt werden. Die Reihenfolge, in der Fehler erkannt werden, ist dabei durch das interne Vorgehen bei der Verarbeitung der Wissensbasis durch den Reasoner bestimmt und folgt keiner den Modellierern transparenten Systematik.

Die Festlegung auf eine bestimmte Reihenfolge, in der einzelne Fehler nacheinander in gemergten Modellen erkannt werden, bedeutet eine Einschränkung der Arbeit der Modellierer bei der Fehlerbeseitigung. Sie können somit nicht selbst entscheiden, welche Fehler zuerst behoben werden sollen und welche erst später.

Durch ein sogenanntes Ontology-Debugging kann aber die Wissensbasis gezielt nach verschiedensten Fehlern durchsucht werden. Durch den Ontologie-Editor Swoop (vgl. (Kalyanpur u. a. 2006)) kann dabei die Fehlersuche unterstützt werden. Dennoch ist u. a. die Benutzbarkeit dieser Art der automatischen Fehleranalyse zurzeit noch stark eingeschränkt (Stuckenschmidt 2008). Abhilfe verschafft in diesem Zusammenhang die Modellvalidierung auf Basis einer automatischen Auswertung von Anfragen an die Wissensbasis. Diese Variante der statischen Analyse wird im folgenden Abschnitt erläutert.

2.4 Alternative 2: Statische Analyse durch SPARQL-Anfragenauswertung im Rahmen der Modellvalidierung

Eine zweite in Abbildung IV.11 dargestellte Möglichkeit zur Erkennung merge-bedingter Fehler ist die Nutzung einer Anfragesprache wie SPARQL(-DL), um eine aus einem Modell generierte, beschreibungslogische Wissensbasis zu untersuchen (vgl. Abbildung IV.11).

Dazu wird die Spezifikation des Metamodells einer Modellierungssprache als eine Bibliothek von Anfragen ausgedrückt. Jede einzelne Anfrage dieser Bibliothek dient dabei der Erkennung von Modellstrukturen, die den im Metamodell spezifizierten Bedingungen widersprechen. Liefert eine solche Anfrage an eine Wissensbasis Ergebnisse, so werden damit Verletzungen der Syntax oder der statischen Semantik erkannt. Die Anfrage bestimmt also Modellelemente, deren Beziehungen untereinander nicht den im Metamodell spezifizierten Bedingungen entsprechen.

Als Beispiel für eine solche Prüfung der statischen Semantik eines Modells durch SPARQL-Anfragen soll die Situation im BSI-Modell nach der morphologischen Integration paralleler Änderungen dienen. Abbildung IV.18 zeigt dazu den relevanten Ausschnitt der Beispielmodellierung aus Abschnitt III.3.2. Durch die Integration der Änderungen im Beispiel verletzt die Beziehung zwischen den Komponenten *CustomerAccount* und *CustomerAccountTransaction* die 4-Schichten-Architektur, wie sie nach der Änderungsintegration im Modell festgelegt ist (vgl. Abbildung III.10). *CustomerAccount* ist dem *ApplicationLayer* und *CustomerAccountTransaction* dem *ControlLayer* zugeordnet (erkennbar durch die Zuordnung der Layersymbole zu den Komponenten), obwohl *CustomerAccount* auf *CustomerAccountTransaction* zugreift.

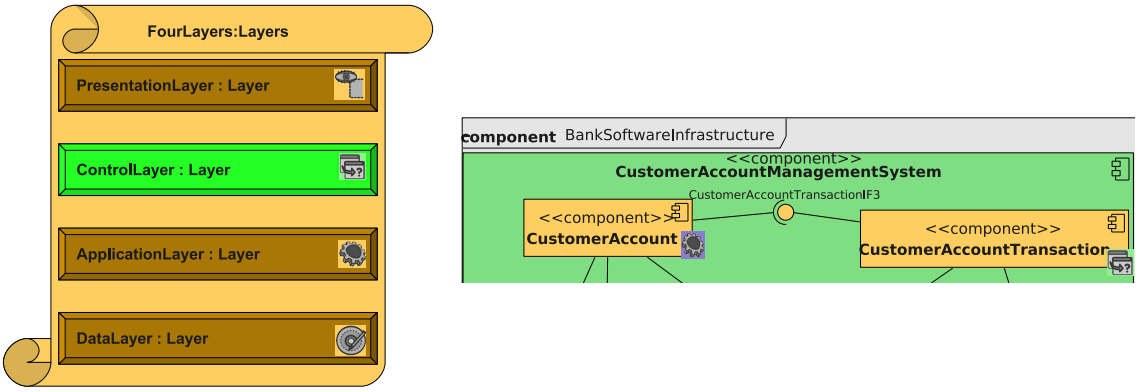


Abbildung IV.18. Prüfung von Schichtenabhängigkeiten bzgl. der Komponentenzuordnung

Als beschreibungslogische ABox lassen sich die relevanten Teile des oben dargestellten Modellausschnitts wie in Auflistung IV.5 beschreiben:

Auflistung IV.5: ABox des Modells „BankSoftwareInfrastructure – BSI“ (Ausschnitt)

$$\mathcal{I}^{ABox}_{MMR^m}(mmr^{BSI}) = \left\{ \begin{array}{l} \mathbf{RE}|\mathbf{Layers}(\mathbf{BSI}|\mathbf{FourLayers}), \\ \mathbf{RE}|\mathbf{Layer}(\mathbf{BSI}|\mathbf{ControlLayer}), \\ \mathbf{RE}|\mathbf{Layer}(\mathbf{BSI}|\mathbf{ApplicationLayer}), \\ \mathbf{RE}|\mathbf{Component}(\mathbf{BSI}|\mathbf{CustomerAccount}), \\ \mathbf{RE}|\mathbf{Component}(\mathbf{BSI}|\mathbf{CustomerAccountTransaction}), \\ \mathbf{RE}|\mathbf{Interface}(\mathbf{BSI}|\mathbf{CustomerAccountTransactionIF3}), \\ \mathbf{RE}|\mathbf{InterfaceRealization}(\mathbf{BSI}|\mathbf{CustomerAccountTransactionIFR3}) \\ \mathbf{RE}|\mathbf{Usage}(\mathbf{BSI}|\mathbf{CustomerAccountUSE3}), \\ \\ \mathbf{RE}|\mathbf{Layers_layer}(\mathbf{BSI}|\mathbf{FourLayers}, \mathbf{BSI}|\mathbf{ControlLayer}), \\ \mathbf{RE}|\mathbf{Layers_layer}(\mathbf{BSI}|\mathbf{FourLayers}, \mathbf{BSI}|\mathbf{ApplicationLayer}), \\ \mathbf{order_RE}|\mathbf{Layers_layer_BSI}|\mathbf{FourLayers}(\mathbf{BSI}|\mathbf{ControlLayer}, \mathbf{BSI}|\mathbf{ApplicationLayer}), \\ \mathbf{RE}|\mathbf{Component_interfaceRealization}(\mathbf{BSI}|\mathbf{CustomerAccountTransaction}, \mathbf{BSI}|\mathbf{CustomerAccountTransactionIFR3}), \\ \mathbf{RE}|\mathbf{Component_packagedElement}(\mathbf{BSI}|\mathbf{CustomerAccount}, \mathbf{BSI}|\mathbf{CustomerAccountUSE3}), \\ \mathbf{RE}|\mathbf{InterfaceRealization_contract}(\mathbf{BSI}|\mathbf{CustomerAccountTransactionIFR3}, \mathbf{BSI}|\mathbf{CustomerAccountTransactionIF3}), \\ \mathbf{RE}|\mathbf{Usage_supplier}(\mathbf{BSI}|\mathbf{CustomerAccountUSE3}, \mathbf{BSI}|\mathbf{CustomerAccountTransactionIF3}), \\ \vdots \\ \mathbf{order_RE}|\mathbf{Layers_layer_BSI}|\mathbf{FourLayers}(\mathbf{Pred}, \mathbf{Succ}), \\ \vdots \end{array} \right\}$$

Wird diese ABox in RDF-Syntax repräsentiert, so kann mit einer SPARQL-Anfrage untersucht werden, ob die Komponentenstruktur die zugrunde gelegte Schichtenarchitektur verletzt oder nicht. Dazu erkennt die in Auflistung IV.6 abgebildete SPARQL-Anfrage alle Komponentenpaare, deren Abhängigkeit bzgl. der Schnittstellennutzung der spezifizierten Schichtenarchitektur widerspricht.

Auflistung IV.6. Anfrage zur Erkennung von Fehlern bzgl. verletzten Schichtenabhängigkeiten in der Komponentenstruktur des BSI-Modells

```
SELECT DISTINCT ?RequiringComponent ?LowerLayer ?ProvidingComponent ?HigherLayer
WHERE {

?HigherLayer           ?PropertyOrder                              ?LowerLayer.
?RequiringComponent    <http://RE#Component_layer>                 ?LowerLayer.
?ProvidingComponent    <http://RE#Component_layer>                 ?HigherLayer.
?RequiringComponent    <http://RE#Component_interfaceRealization> ?InterfaceRealization.
?ProvidingComponent    <http://RE#Component_packagedElement>       ?Usage.
?InterfaceRealization  <http://RE#InterfaceRealization_contract>  ?Interface.
?Usage                 <http://RE#Usage_supplier>                  ?Interface.
<http://help#Pred>     ?PropertyOrder                              <http://help#Succ>
}
```

Wie erwartet kennt die Anfrage aus Auflistung IV.6 fehlerhafte Komponentenstrukturen, was durch die Ausgabe einer SPARQL-Query-Engine in Auflistung IV.7 dargestellt ist.

Auflistung IV.7. Ergebnis der Anfrageausführung aus Auflistung IV.6

```
-------------------------------------------------------------------------------------------
| RequiringComponent | LowerLayer       | ProvidingComponent         | HigherLayer |
===========================================================================================
| CustomerAccount    | ApplicationLayer | CustomerAccountTransaction | ControlLayer |
-------------------------------------------------------------------------------------------
```

Eine solche SPARQL-Anfrage muss entweder manuell erstellt werden – bspw. im Rahmen der Spezifikation von statischer Semantik des RE-Metamodells. Alternativ könnten auch entsprechend vorhandene OCL-Constraints des RE-Metamodells in Anfragen automatisch übersetzt werden. Diese Möglichkeit wurde jedoch nicht weiter im Rahmen dieser Arbeit untersucht.

Um eine Lösung für die in Abschnitt 2.3.3 erörterte Problematik hinsichtlich des fehlenden Überblicks über die gesamte Merge-bedingte Fehlersituation zu erhalten, kann aus einem per MOF spezifizierten Metamodell eine Bibliothek von SPARQL-Anfragen zu Verstößen generiert werden. Die Ausführung aller Anfragen in der Bibliothek auf der mit Interpretation IV.2 generierten ABox ermöglicht es dann, eine ganzheitlichere Übersicht über merge-bedingte Fehler zu erhalten.

Dazu lassen sich ebenso syntaktische Fehler, die durch den MOF-spezifizierten Anteil des Metamodells bestimmt sind, durch SPARQL-Anfragen automatisch erkennen. Für das Beispiel in Abbildung IV.15/Auflistung IV.2 lässt sich ebenfalls eine SPARQL-Anfrage zur Fehlererkennung angeben. Die Anfrage ist mit ihrer entsprechenden Ausgabe der Query-Engine in Auflistung IV.8 abgebildet. Derartige Anfragen können dabei direkt aus dem RE-Metamodell automatisch generiert werden.

Mit den in den Abschnitten 2.3 und 2.4 vorgestellten Alternativen lassen sich Fehler in Bezug auf die Metamodellkonformität eines nach dem morphologischen Merge-Verfahren integrierten Modells automatisch erkennen und analysieren. Dabei kann sowohl die Modellsyntax, welche durch den per MOF spezifizierten Anteil des Metamodells festgelegt wurde, als auch weitere (OCL-)Bedingungen der statischen Semantik geprüft werden. Eine zusätzliche Analyse bzgl. weiterer Fehler (vgl. Abschnitt III.4.4.3) in Modellen diskutiert der folgende Abschnitt.

Auflistung IV.8. Anfrage zur Erkennung syntaktischer Fehler bzgl. Eigenschaften der Containment-Beziehung

SPARQL-Anfrage:

```
-------------------------------------------------------------------------------------
SELECT DISTINCT ?Container1 ?Container2 ?Containment
WHERE {
  ?Container1 <http://anonymous.de#containment> ?Containment.
  ?Container2 <http://anonymous.de#containment> ?Containment.
  FILTER (?Container1 != ?Container2)
}
```

Antwort der Query-Engine:

```
-------------------------------------------------------------------------------------
| Container1                | Container2                | Containment
|
=====================================================================================
| MarketingSupportSystem    | CustomerAccountManagement | CustomerInternetClientSession
|
| CustomerAccountManagement | MarketingSupportSystem    | CustomerInternetClientSession>
|
| MarketingSupportSystem    | CustomerAccountManagement | PrepareOffer
|
| CustomerAccountManagement>| MarketingSupportSystem    | PrepareOffer>
|
-------------------------------------------------------------------------------------
```

2.5 Statische Analyse zur Anomalieerkennung durch TBox-Reasoning

In den letzten beiden Abschnitten wurde vorgestellt, wie geprüft werden kann, ob Modelle hinsichtlich eines vorgegebenen Metamodells korrekt sind. Diese Korrektheit schließt die vollständige Prüfung der syntaktischen Korrektheit (incl. der sog. statischen Semantik) ein.

Durch die Sicherstellung der syntaktischen Korrektheit ist eine inhaltliche Interpretation des Modells möglich. Eine solche Interpretation erlaubt eine weitere statische Analyse der Modellkorrektheit. Zur Erläuterung dieser Analyse wird das beispielhafte Modellierungsszenario aus Abbildung III.22 herangezogen. Dazu stellt Abbildung IV.19 den Modellausschnitt zusammen mit dem relevanten Metamodellausschnitt dar.

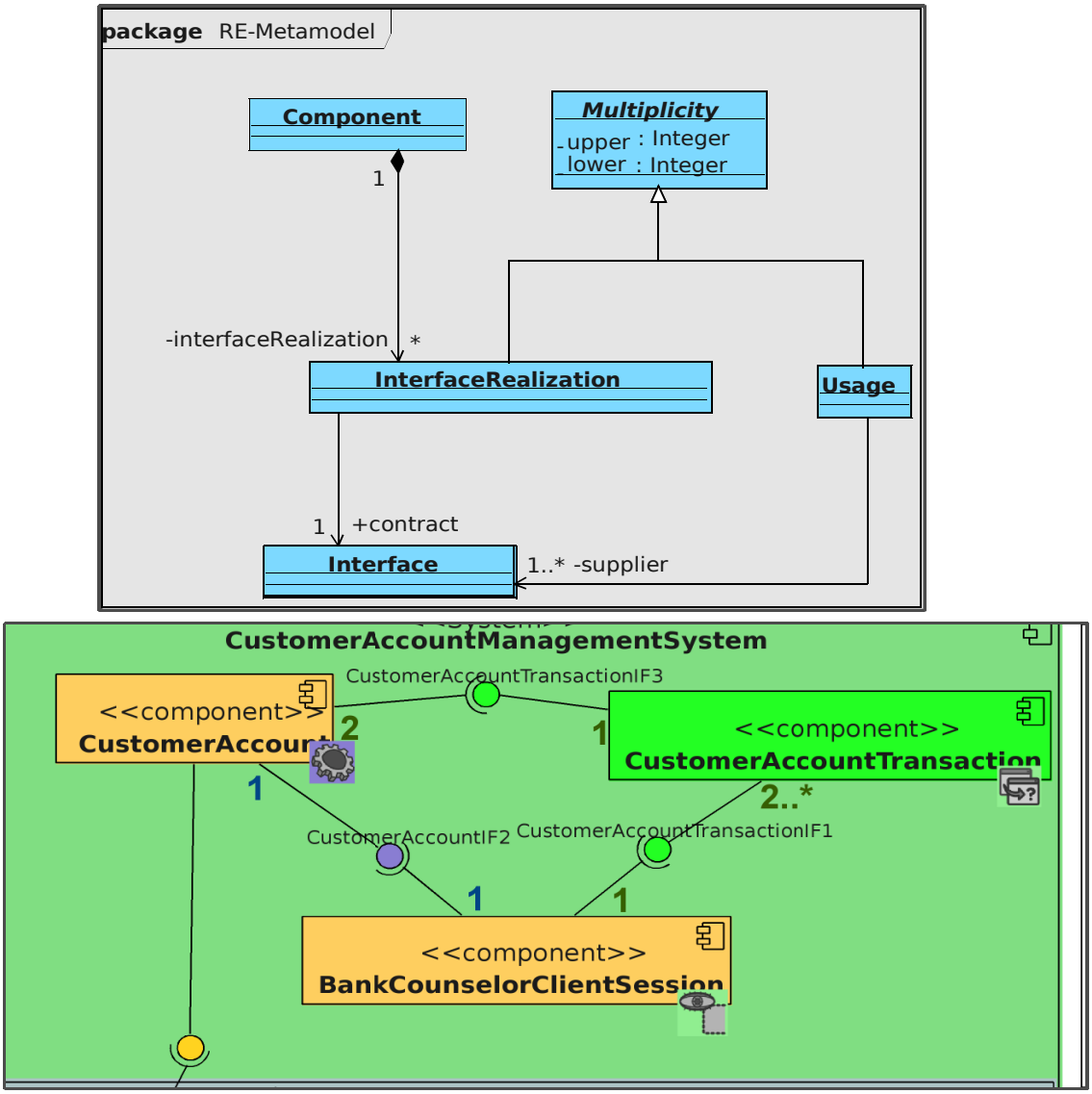


Abbildung IV.19. Beispiel für Anomalie im Ausschnitt des BSI-Modells

Wie in der Abbildung zu sehen, widerspricht der Modellausschnitt des BSI-Modells nicht der Syntaxspezifikation, die das RE-Metamodell vorgibt. Dennoch ist das Modell logisch fehlerhaft, wenn man davon ausgeht, dass jedes Modellelement potenziell instanziierbar sein muss, wenn von insgesamt endlich vielen Instanzen ausgegangen wird (endliche Domäne, engl. *finite domain*).

In diesem Fall kann es für die Komponenten *BankCounselorClientSession*, *CustomerAccountTransaction* und *CustomerAccount* (MOF-Ebene M1) keine Ausprägungen auf der MOF-Ebene M0 geben, die den modellierten Multiplizitäten genügen. Sofern eine Instanz von *BankCounselorClientSession* erstellt wird, muss diese mit mindestens zwei Instanzen von *CustomerAccountTransaction* über InterfaceRealization/Usage verknüpft werden. Diese beiden Instanzen müssen wiederum jeweils mit zwei Instanzen von

CustomerAccount verbunden werden, wobei jede Instanz von *CustomerAccount* mit genau einer *BankCounselorClientSession*-Instanz verknüpft sein muss. Bei genauerer Betrachtung sieht man, dass jede Ausprägung des BSI-Modells mit einer beliebigen Anzahl an *BankCounselorClientSession*-, *CustomerAccountTransaction*- bzw. *CustomerAccount*-Instanzen größer als Null widersprüchlich zu den modellierten Multiplizitätsbedingungen wäre.

Dies verdeutlicht folgende Betrachtung, wobei die Typ-Instanz-Beziehung durch die Menge-Element-Beziehung ausgedrückt wird und von einer endlichen Domäne ausgegangen wird:

$$
\begin{aligned}
&|BankCounselorClientSession| = n, \quad n \in \mathbb{N} \text{ (finite domain)} \\
&\Rightarrow |CustomerAccountTransaction| \geq 2n \\
&\Rightarrow |CustomerAccount| \geq 2n \\
&\Rightarrow |BankCounselorClientSession| \geq 2n \\
&\Rightarrow n = 0 \quad \square
\end{aligned}
$$

Analog zur oben belegten Schlussfolgerung, kann man ebenfalls zeigen, dass es nach dem Modell von Abbildung IV.19 ebenfalls für *CustomerAccountTransaction* und *CustomerAccount* keine Instanzen geben kann. Alle drei Komponententypen lassen sich somit nicht in der Realität (MOF-Ebene M0) durch nicht-leere Instanzmengen abbilden.

Warum bedeutet eine solche Situation nun, dass ein Modell inkorrekt ist? Die Antwort auf diese Frage ergibt sich aus der Modelldefinition nach Stachowiak (Stachowiak 1973) (vgl. Abschnitt II3). Demnach muss für ein Modell das Abbildungskriterium gelten, das verlangt, dass es zu jedem Modell ein entsprechendes Original existieren muss oder in Zukunft erstellt werden könnte. Genau dieses Kriterium trifft aber für Modelle bzw. Modellausschnitte nicht zu, wenn Modellelemente wie bspw. die genannten Komponententypen keine Entsprechung in einer endlichen Domäne haben können.

Analog zum oben angeführten Beispiel zeigt Abbildung III.4 ein Beispiel für einen gleichartigen Fehler im Quellcode. Dadurch, dass in diesem Beispiel die Schleifenbedingung immer zu *false* ausgewertet wird, wird der Schleifenrumpf nie ausgeführt. Auch hier kann es keine Entsprechung der im Schleifenrumpf angegebenen Anweisungen als konkrete Ausführungen zur Laufzeit geben. Die Kontrollflussanalyse im Rahmen der semantischen Analyse bzw. im Rahmen der Zwischencodeoptimierung erkennt in diesen Fällen die Problemsituation (→ Anomalie) und warnt den Entwickler (vgl. Abbildung III.4).

Bei der automatischen Integration von parallelen Änderungen durch 3-Wege-Merge-Verfahren ist das Risiko, dass derartige Anomalien in gemergten Modellen auftreten, sehr hoch. Denn, wenn mehrere Modellierer widersprüchlich Vorstellungen von dem zu modellierenden Sachverhalt haben, kann das Ergebnis eines Merges entsprechend inkorrekt sein, obwohl ihre jeweiligen asynchronen Modellversionen korrekt sind.

Ziel dieses Abschnittes ist es, ein Verfahren vorzustellen, mit dem logische Widersprüche in gemergten Modellen, wie in Abbildung IV.19, automatisch erkannt werden können. Grundlage für dieses Verfahren bildet die Analyse der Klassen- oder auch Konzepterfüllbarkeit (engl. class/concept satisfiability) in TBoxen. Erstmalig wurden diese Verfahren für die Validierung von UML-Klassendiagrammen in (Berardi, Calvanese und De Giacomo 2001) vorgestellt. Weitere Arbeiten sind in (Berardi, Calvanese und Giacomo 2005b; Makarenkov u. a. 2009) veröffentlicht. Nachfolgend soll vorgestellt werden, wie beliebige Modelle auf Basis des genannten Verfahrens automatisch analysiert werden können.

Grundidee des Analyseverfahrens ist die Reduzierung der Frage nach der Existenz von Anomalien in einer Modellversion auf die Frage nach der Klassenerfüllbarkeit (engl. class satisfiability) (vgl. Abschnitt 2.2.2) unter Vorgabe endlicher Modelle (engl. finite model property) einer Wissensbasis. Im Gegensatz zur Analyse der Klassenerfüllbarkeit im Allgemeinen ist das Finite-Model-Reasoning vergleichsweise wenig erforscht. Für die Beschreibungslogik $\mathcal{ALCQI}$ ist allerdings nachgewiesen, dass die Klassenerfüllbarkeit auch unter der finite-model-Annahme entscheidbar geprüft werden kann *(Rosati 2008)*. Dazu wird der TBox-Reasoning-Service einer beschreibungslogischen Inferenzmaschine eingesetzt.

Zur entsprechenden Korrektheitsüberprüfung eines Modells muss dieses als eine TBox von $\mathcal{ALCQI}$ interpretiert werden. $\mathcal{ALCQI}$ erlaubt allerdings nicht den Einsatz aller Sprachmittel von $\mathcal{SHOIN}$. $\mathcal{ALCQI}$ erlaubt ausschließlich den Einsatz folgender Ausdrücke:

$$\mathcal{ALCQI} := \bot \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid C \sqsubseteq D \mid \exists R.C \mid \forall R.C \mid \geq n\ R.C \mid \leq n\ R.C \mid R^{-}$$

Die Semantik der einzelnen Ausdrücke entspricht dabei ihrer Entsprechungen in Abschnitt 2.2.1.

Nachdem nun bestimmt wurde welche Sprachmittel für die Interpretation eines Modells in Beschreibungslogiken zur Verfügung stehen, soll am eingangs erwähnten Beispiel die motivierte Interpretation illustriert werden. Um Modellversionen wie motiviert validieren zu können, muss neben dem Metamodell der Modellierungssprache die Interpretationsabbildung $\mathcal{I}^{TBox}_{MMR^{RE}} : \wp(MMR^{RE}) \rightarrow TBox^{\mathcal{ALCQI}}$ dem Analyseverfahren zur Verfügung gestellt werden. Ein Beispiel für eine solche Abbildung ist in Interpretation IV.3 angegeben.

Interpretation IV.3: Durch die Abbildung $\mathcal{I}^{TBox}_{MMR^{RE}} : \wp(MMR^{RE}) \rightarrow TBox^{\mathcal{ALCQI}}$ werden in MMR repräsentierte RE-Modelle (Ausprägungen des RE-Metamodells) in logische Ausdrücke einer beschreibungslogischen TBox transformiert (vgl. Abbildung IV.11). Nachfolgend wird ein zentraler Ausschnitt dieser Abbildung dargestellt:

$$\begin{array}{rcl}
\vdots & \Rightarrow & \\
(\{(\text{"RE|Component"}, id^{class})\} \subseteq mmr^{\hat{m}})^{\mathcal{I}^{TBox}_{MMR^{RE}}} & \Rightarrow & id^{class} \in \mathcal{C} \\
(\left\{ \begin{array}{l} (i, \text{"RE|Component_packagedElement"}, id^{comp}, id^{usage}) \\ (i, \text{"RE|Usage_supplier"}, id^{usage}, id^{if}) \end{array} \right\} \subseteq mmr^{\hat{m}})^{\mathcal{I}^{TBox}_{MMR^{RE}}} & \Rightarrow & id^{comp}_id^{if} \in \mathcal{R} \\
\left(\left\{ \begin{array}{l} (i, \text{"RE|Multiplicity_upper"}, id^{usage}, ub), \\ (i, \text{"RE|Component_packagedElement"}, id^{comp}, id^{usage}) \\ (i, \text{"RE|Usage_supplier"}, id^{usage}, id^{if}) \end{array} \right\} \subseteq mmr^{\hat{m}}, ub \neq \text{"*"} \right)^{\mathcal{I}^{TBox}_{MMR^{RE}}} & \Rightarrow & \text{"}id^{comp} \sqsubseteq \leq ub\, id^{comp}_id^{if}\text{"} \in \text{TBox} \\
\left(\left\{ \begin{array}{l} (i, \text{"RE|Multiplicity_lower"}, id^{usage}, lb), \\ (i, \text{"RE|Component_packagedElement"}, id^{comp}, id^{usage}) \\ (i, \text{"RE|Usage_supplier"}, id^{usage}, id^{if}) \end{array} \right\} \subseteq mmr^{\hat{m}} \right)^{\mathcal{I}^{TBox}_{MMR^{RE}}} & \Rightarrow & \text{"}id^{comp} \sqsubseteq \geq lb\, id^{comp}_id^{if}\text{"} \in \text{TBox} \\
\left(\left\{ \begin{array}{l} (i, \text{"RE|Multiplicity_upper"}, id^{ifreal}, ub), \\ (i, \text{"RE|Component_packagedElement"}, id^{comp1}, id^{usage}) \\ (i, \text{"RE|Component_interfaceRealization"}, id^{comp2}, id^{ifreal}) \\ (i, \text{"RE|Usage_supplier"}, id^{usage}, id^{if}) \\ (i, \text{"RE|InterfaceRealization_contract"}, id^{ifreal}, id^{if}) \end{array} \right\} \subseteq mmr^{\hat{m}}, ub \neq \text{"*"} \right)^{\mathcal{I}^{TBox}_{MMR^{RE}}} & \Rightarrow & \text{"}id^{comp2} \sqsubseteq \leq ub\, id^{comp1}_id^{if^{-}}\text{"} \in \text{TBox} \\
\left(\left\{ \begin{array}{l} (i, \text{"RE|Multiplicity_lower"}, id^{ifreal}, lb), \\ (i, \text{"RE|Component_packagedElement"}, id^{comp1}, id^{usage}) \\ (i, \text{"RE|Component_interfaceRealization"}, id^{comp2}, id^{ifreal}) \\ (i, \text{"RE|Usage_supplier"}, id^{usage}, id^{if}) \\ (i, \text{"RE|InterfaceRealization_contract"}, id^{ifreal}, id^{if}) \end{array} \right\} \subseteq mmr^{\hat{m}} \right)^{\mathcal{I}^{TBox}_{MMR^{RE}}} & \Rightarrow & \text{"}id^{comp2} \sqsubseteq \geq lb\, id^{comp1}_id^{if^{-}}\text{"} \in \text{TBox} \\
\left(\left\{ \begin{array}{l} (i, \text{"RE|Component_packagedElement"}, id^{comp1}, id^{usage}) \\ (i, \text{"RE|Component_interfaceRealization"}, id^{comp2}, id^{ifreal}) \\ (i, \text{"RE|Usage_supplier"}, id^{usage}, id^{if}) \\ (i, \text{"RE|InterfaceRealization_contract"}, id^{ifreal}, id^{if}) \end{array} \right\} \subseteq mmr^{\hat{m}} \right)^{\mathcal{I}^{TBox}_{MMR^{RE}}} & \Rightarrow & \text{"}\top \sqsubseteq \forall id^{comp1}_id^{if}.id^{comp2}\text{"} \in \text{TBox} \\
\left(\left\{ \begin{array}{l} (i, \text{"RE|Component_packagedElement"}, id^{comp}, id^{usage}) \\ (i, \text{"RE|Usage_supplier"}, id^{usage}, id^{if}) \end{array} \right\} \subseteq mmr^{\hat{m}} \right)^{\mathcal{I}^{TBox}_{MMR^{RE}}} & \Rightarrow & \text{"}\exists id^{comp}_id^{if}.\top \sqsubseteq id^{comp}\text{"} \in \text{TBox} \in \text{TBox}
\end{array}$$

Wendet man die Interpretation IV.3 auf die MMR des Modells in Abbildung IV.19 an so erhält man eine TBox von $\mathcal{ALCQI}$ wie sie in Auflistung IV.9 abgebildet ist. Analysiert man die in Auflistung IV.9 skizzierte TBox mit einem Reasoner, der das Finite-Model-Reasoning berücksichtigen kann (bspw. QuOnto *(DASI-lab group - University of Rome und IBM Center for Advanced Studies (CAS) 2010; Rosati 2008)*) so werden die Services *BankCounselorClientSession*, *CustomerAccountTransaction* und *CustomerAccount* wie erwartet als nicht (endlich) erfüllbar unter dem Finite-Domain-Aspekt identifiziert. Die Identifizierung der betroffenen Elemente erlaubt eine Fehler-Markierung in der betroffenen Modellversion, was die Fehlerbehebung stark vereinfacht.

Neben logischen Fehlern aufgrund von Multiplizitätsbedingungen lassen sich eine Reihe weiterer Anomalien durch das TBox-Reasoning erkennen. Beschränkt werden die Analysemöglichkeiten in endlichen Domänen durch die sprachlichen Ausdrucksmöglichkeiten von $\mathcal{ALCQI}$ oder vergleichbarer Beschreibungslogiken.

Auflistung IV.9: BSI-Modell (Ausschnitt) als beschreibungslogische TBox (vgl. Abbildung IV.19)

$$\mathcal{I}^{TBox}_{MMR^{RE}}\left(mmr^{BSI}\right) = \left\{ \begin{array}{l} \vdots \\ \mathbf{BSI|BCClientSession} \\ \mathbf{BSI|CustomerAccountTransaction} \\ \mathbf{BSI|CustomerAccount} \\ \\ \mathbf{BSI|BCClientSession_catansactionIF1} \\ \mathbf{BSI|CustomerAccount_catransactionIF3} \\ \mathbf{BSI|BCClientSession_caccountIF2} \\ \\ \top \sqsubseteq \forall \mathbf{BSI|BCClientSession_catansactionIF1}.\mathbf{BSI|CustomerAccountTransaction} \\ \exists \mathbf{BSI|BCClientSession_catansactionIF1}.\top \sqsubseteq \mathbf{BSI|BCClientSession} \\ \top \sqsubseteq \forall \mathbf{BSI|CustomerAccount_catransactionIF3}.\mathbf{BSI|CustomerAccountTransaction} \\ \exists \mathbf{BSI|CustomerAccount_catransactionIF3}.\top \sqsubseteq \mathbf{BSI|CustomerAccount} \\ \top \sqsubseteq \forall \mathbf{BSI|BCClientSession_caccountIF2}.\mathbf{BSI|CustomerAccount} \\ \exists \mathbf{BSI|BCClientSession_caccountIF2}.\top \sqsubseteq \mathbf{BSI|BCClientSession} \\ \mathbf{BSI|BCClientSession} \sqsubseteq \geq 2\, \mathbf{BSI|BCClientSession_catansactionIF1} \\ \mathbf{BSI|CustomerAccountTransaction} \sqsubseteq \geq 1\, \mathbf{BSI|BCClientSession_catansactionIF1}^{-} \\ \mathbf{BSI|CustomerAccountTransaction} \sqsubseteq \leq 1\, \mathbf{BSI|BCClientSession_catansactionIF1}^{-} \\ \mathbf{BSI|CustomerAccountTransaction} \sqsubseteq \geq 2\, \mathbf{BSI|CustomerAccount_catransactionIF3}^{-} \\ \mathbf{BSI|BCClientSession} \sqsubseteq \geq 1\, \mathbf{BSI|BCClientSession_caccountIF2} \\ \mathbf{BSI|BCClientSession} \sqsubseteq \leq 1\, \mathbf{BSI|BCClientSession_caccountIF2} \\ \mathbf{BSI|CustomerAccount} \sqsubseteq \geq 1\, \mathbf{BSI|BCClientSession_caccountIF2}^{-} \\ \mathbf{BSI|CustomerAccount} \sqsubseteq \leq 1\, \mathbf{BSI|BCClientSession_caccountIF2}^{-} \\ \\ \mathbf{BSI|BCClientSession} \sqcup \mathbf{BSI|CustomerAccountTransaction} \sqcup \mathbf{BSI|CustomerAccount} \sqcup \ldots \equiv \top \\ \mathbf{BSI|BCClientSession} \sqcap \mathbf{BSI|CustomerAccountTransaction} \equiv \bot \\ \mathbf{BSI|CustomerAccountTransaction} \sqcap \mathbf{BSI|CustomerAccount} \equiv \bot \\ \mathbf{BSI|CustomerAccount} \sqcap \mathbf{BSI|BCClientSession} \equiv \bot \\ \vdots \end{array} \right\}$$

2.6 Auflösung inkonsistenter Änderungen / Wiederherstellung syntaktischer Korrektheit

Mit den in den Abschnitten 2.3, 2.4 und 2.5 vorgestellten Verfahren lassen sich syntaktische Fehler (incl. statischer Semantik) und Anomalien in Modellen erkennen. Ferner identifizieren die Analyseverfahren die in Konflikt stehenden, syntaktischen Modellelemente in den asynchron erzeugten Modellversionen.

Ohne weitere Informationen der Modellierer, die an den in Konflikt stehenden Änderungen beteiligt sind, kann die Konfliktauflösung nicht weiter automatisiert werden. Die Konfliktauflösung hängt insbesondere von der Intention der Modellierer ab, was sie mit ihren

Änderungen im Modell zum Ausdruck bringen wollten. Da diese Informationen nicht maschinenverarbeitbar vorliegen, bleibt nur die Möglichkeit einer manuellen Konfliktlösung.

Dabei kann die manuelle Konfliktauflösung im Rahmen eines interaktiven Verfahrens zur Wiederherstellung der syntaktischen Korrektheit unterstützt werden. In Entwicklungsumgebungen wie Eclipse werden bei Syntaxfehlern im Code heute schon sog. *Quick Tipps* angezeigt. Diese entsprechen Lösungsvorschlägen zur Korrektur syntaktischer Fehler. Die Auswahl einer vorgeschlagenen Lösungsalternative oder die Korrektur auf eine andere Weise bleibt allerdings weiterhin in der Verantwortung des Entwicklers.

Auch im Forschungsfeld zur Versionsverwaltung von Modellen wird an der Unterstützung der Konfliktbehandlung gearbeitet. Weitergehende Informationen dazu finden sich unter (Brosch 2009; Mens, Straeten und Maja D'Hondt 2005).

Fazit

Mit den in den Abschnitten 2.3, 2.4 und 2.5 vorgestellten Verfahren können merge-bedingte Fehler im Anschluss an die Anwendung des morphologischen Merge-Verfahrens automatisch erkannt und analysiert werden. Kern dieser Analyse ist die automatische Erkennung syntaktischer Fehler und weiterer Anomalien, um eine manuelle Auflösung inkonsistenter Änderungen weitestgehend maschinell zu unterstützen.

Dabei wird die Konformität des abstrakten Syntaxgraphen eines Modells zur Syntaxspezifikation (incl. Nebenbedingungen der sog. statischen Semantik) im Metamodell der jeweiligen Modellierungssprache nachgewiesen. Darüber hinaus können weitere Anomalien, wie die fehlende Abbildbarkeit von Modellelementen (Ebene M1) in der Realität (Ebene M0) durch Untersuchung der Konzepterfüllbarkeit, erkannt werden. Dazu bedienen sich die vorgestellten Ansätze der Repräsentation von Metamodell- und Modellinformationen als beschreibungslogische Wissensbasis bzw. entsprechender Anfragen. Diese Repräsentationsform erlaubt eine automatische Fehleranalyse durch Reasoning-Werkzeuge und Query-Engines.

Bei der kollaborativen Modellierung ist eine effiziente Erkennung und Analyse von merge-bedingten Fehlern entscheidend für eine praktikable asynchrone Arbeitsweise an Modellen. Durch Automatisierung der Integration paralleler Änderungen und die anschließende Fehleranalyse beschränkt sich der manuelle Aufwand für die Integration parallel erstellter Versionen ausschließlich auf Behebung der erkannten Fehler und auf die Erkennung nicht automatisch analysierbarer Modellierungsfehler. Bei der manuellen Analysearbeit muss dann aber die Konformität eines Modells zu dem formal spezifizierten Metamodell einer Modellierungssprache nicht mehr berücksichtigt werden.

Nachdem nun in diesem Kapitel alle theoretischen Ansätze zur Lösung der Merge-Problematik in der kollaborativen Modellierung erörtert wurden, soll im folgenden Kapitel die Implementierung einer Versionsverwaltung für Modelle vorgestellt werden, die auf Basis der eingeführten Ansätze realisiert wurde.

V Implementierung und Konzeptbewertung

Parallel zur Entwicklung der theoretischen Lösungsansätze zur Unterstützung der kollaborativen Softwaremodellierung, wurde ein optimistisches Versionsverwaltungssystem für Modelle entwickelt, das die erforschten Ansätze implementiert. Nach einer ersten prototypischen Implementierung (Bartelt, Molter und Schumann 2009) auf Basis des SCM-Datenmodells der Kollaborationsplattform Jazz stellte sich heraus, dass die Performance des Im- und Exports von XMI-Dateien großer Modelle in die MMR und der Merge unzureichend für einen praktikablen Einsatz ist.

Daraufhin wurde die Versionsverwaltungssoftware für Modelle, i. F. Team.Mode genannt, von Grund auf neu entwickelt. Dabei wurde insbesondere auf eine performante Implementierung der vorgestellten Lösungsansätze geachtet. Zur Beurteilung der praktikablen Einsetzbarkeit von Team.Mode wurde die Software im Rahmen unterschiedlicher Modellierungsszenarien evaluiert. Die dabei zugrunde gelegten Anwendungsfälle wurden aus publizierten Industrieanforderungen abgeleitet.

Team.Mode wurde wie die populären Versionsverwaltungswerkzeuge zur Codeverwaltung CVS und Subversion als Client-Server-System entworfen. Dabei entspricht die Client-Software einem Fat-Client, der als Eclipse-Plug-ins realisiert ist. Als Server dient ein Datenbankserver mit einer beliebigen relationalen Datenbank. Um auf den Erfahrungshintergrund von Softwareingenieuren einzugehen, spiegelt die Anwenderinteraktion in Team.Mode in wesentlichen Teilen die Interaktion mit Codeverwaltungssystemen wieder.

Die folgenden Abschnitte sollen eine kurze Einführung in die Software Team.Mode geben und damit ihre Anwendung in der kollaborativen Softwaremodellierung plastisch darstellen. Dazu wird im folgenden Abschnitt die Arbeit mit Team.Mode-Projekten in Eclipse vorgestellt. Die Anbindung an ein Team-weites Repository, die Erstellung neuer Versionen und Entwicklungszweige und der Merge von Modellversionen stehen hierbei im Mittelpunkt. In Abschnitt 2 wird anschließend die implementierte Fehleranalyse für Modellversionen vorgestellt, deren syntaktische Korrektheit möglicherweise durch das 3-Wege-Merge-Verfahren verletzt wurde.

1 Team.Mode – Automatisierung der Versionsverwaltung von Softwaremodellen

Die Team.Mode-Versionsverwaltung besteht aus einem Client-Server-System. Analog zu Clientanwendungen von Versionsverwaltungssystemen wie CVS und Subversion wurde für Team.Mode ein entsprechender Eclipse-Client entwickelt. Die Hintergründe dieser Entscheidung sind rein pragmatischer Natur.

Fast alle Modellierungswerkzeuge im Bereich der Softwaremodellierung basieren auf der Eclipse-Plattform bzw. bieten auch eine Eclipse-Werkzeugintegration an. Außerdem stehen alle externen Softwareeinheiten, die Team.Mode benötigt, wie bspw. das Eclipse-Modeling-Framework (EMF) oder die OWL-Unterstützung als Plug-ins für Eclipse zur Verfügung. Die Einbettung bietet somit vonseiten der Integration der Werkzeuglandschaft aus Anwendersicht als auch für das Packaging der Team.Mode-Software an sich viele Vorteile.

Für die folgende Vorstellung wurde Team.Mode in eine Softwareumgebung zusammen mit dem Modellierungswerkzeug Topcased in Eclipse integriert. Topcased diente dabei der Bearbeitung der zu verwaltenden Modelle, wie sie auch schon in den Beispielszenarien in den vorangegangenen Kapiteln vorgestellt wurden. Genau wie Topcased ließe sich aber auch jedes beliebige andere Modellierungswerkzeug, das den EMF/XMI-Standard unterstützt, in eine solche kollaborative Modellierungsumgebung einbetten.

Bevor in Abschnitt 1.2 auf die Verarbeitung von Modellen durch Team.Mode eingegangen wird, zeigt der folgende Abschnitt das Aufsetzen und Verwalten von sog. Team.Mode-Projekten.

1.1 Arbeiten mit Team.Mode-Projekten

Soll einem Team die Möglichkeit gegeben werden, asynchron Modelle zu bearbeiten, so muss mithilfe des Clients zunächst ein sog. Team.Mode-Projekt initiiert werden. Dazu nutzt der Initiator des Projektes den entsprechenden Eclipse-Dialog, wie er in Abbildung V.1 zu sehen ist. Anschließend wird ein neues Modellierungsprojekt in Eclipse erstellt.

Wie in der Zielsetzung und der Vision der Dissertation in Abschnitt I.3 und 4 beschrieben wurde, ist die Modellverwaltungssoftware unabhängig von bestimmten Modellierungssprachen für Modelle entworfen worden. Um die Verarbeitung von Modellen bestimmter Modellierungssprachen zu ermöglichen, müssen der Software die formalen Spezifikationen der verwendeten Metamodelle zur Verfügung gestellt werden. Dazu gibt der Initiator eines Modellierungsprojektes mithilfe des unten in Abbildung V.1 dargestellten Dialoges die XMI-Dateien an, in denen die relevanten Metamodellspezifikationen enthalten sind.

Diese müssen mit der Meta-Metamodellierungssprache ECore beschrieben worden sein. Des Weiteren muss der Projektersteller pro Metamodell eine Dateinamenserweiterung für die entsprechend zu verwaltenden Modelle des jeweiligen Metamodells angeben (in der Abbildung „re“ für das RE-Metamodell).

Nachdem ein Projekt erstellt wurde, kann ein Repository auf einem zentralen Datenbankserver erstellt werden. Über diesen Server läuft anschließend die gesamte inhaltliche Synchronisation der im Team erstellten Modelldaten. Der Dialog zur Erstellung eines neuen Repositorys auf dem Server ist in Abbildung V.2 oben zu sehen. Der Repository-Ersteller muss

anschließend den Datenbank-Connector den Host und den Namen der Datenbank angeben, wie es in Abbildung V.2 dargestellt ist.

In einem so erstellten Repository können nun alle Versionen, die durch ein Modelliererteam erstellt werden, gespeichert werden. Dabei wird zwischen Modellierer-spezifischen Versionen und Team-Versionen unterschieden. Modellierer-spezifische Versionen können nur durch den jeweiligen Modellierer weiterentwickelt werden; sie entsprechen einem Personen-bezogenem Entwicklungszweig. Team-Versionen von Modellen dienen als inhaltlicher Synchronisationspunkt für alle Softwareingenieure eines Modelliererteams. Sie sind für alle sichtbar und dienen zur Integration der eigenen Modellbearbeitungen eines Modellierers in den Team-weiten Informationsstand.

Ist schon ein Repository vorhanden, können sich weitere Teammitglieder mit einem gemeinsamen Repository verbinden. Dazu wählen sie wie bei CVS- und SVN-Projekten auch üblich über das Kontextmenu eines Projektes den Menüpunkt „Share Project“ aus. Danach können sie sich, wie in Abbildung V.3 oben dargestellt, für Team.Mode-Repository entscheiden und, wie in der Abbildung im unteren Bereich dargestellt, die entsprechende Datenbankverbindung angeben.

Nachdem nun mehrere Modellierer sich mit einem Team.Mode-Repository verbunden haben, kann die kollaborative Modellierungsarbeit beginnen. Die dementsprechende Verwendung der implementierten Software wird im nächsten Abschnitt genauer beleuchtet.

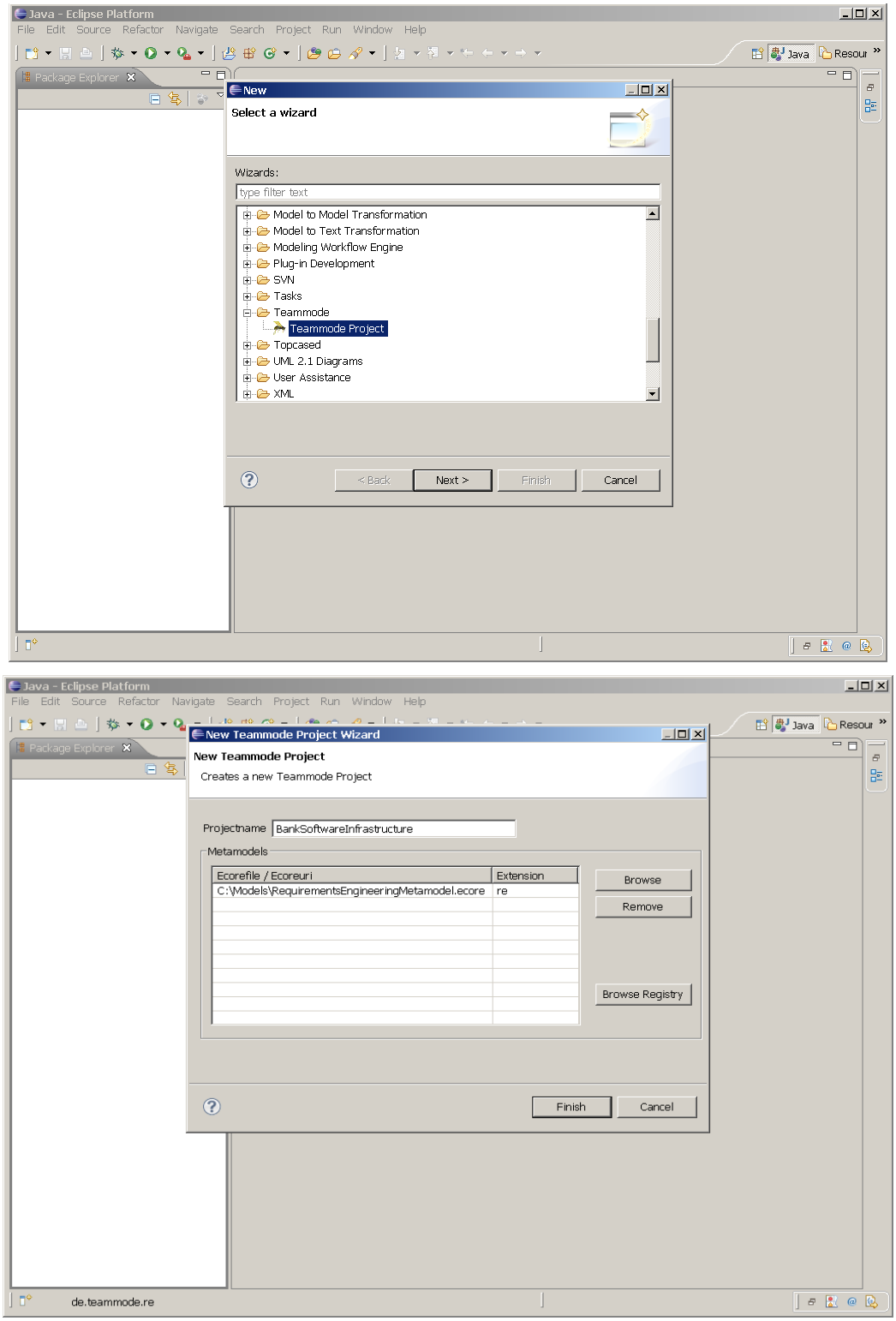


Abbildung V.1. Erstellung neuer Team.Mode-Projekte

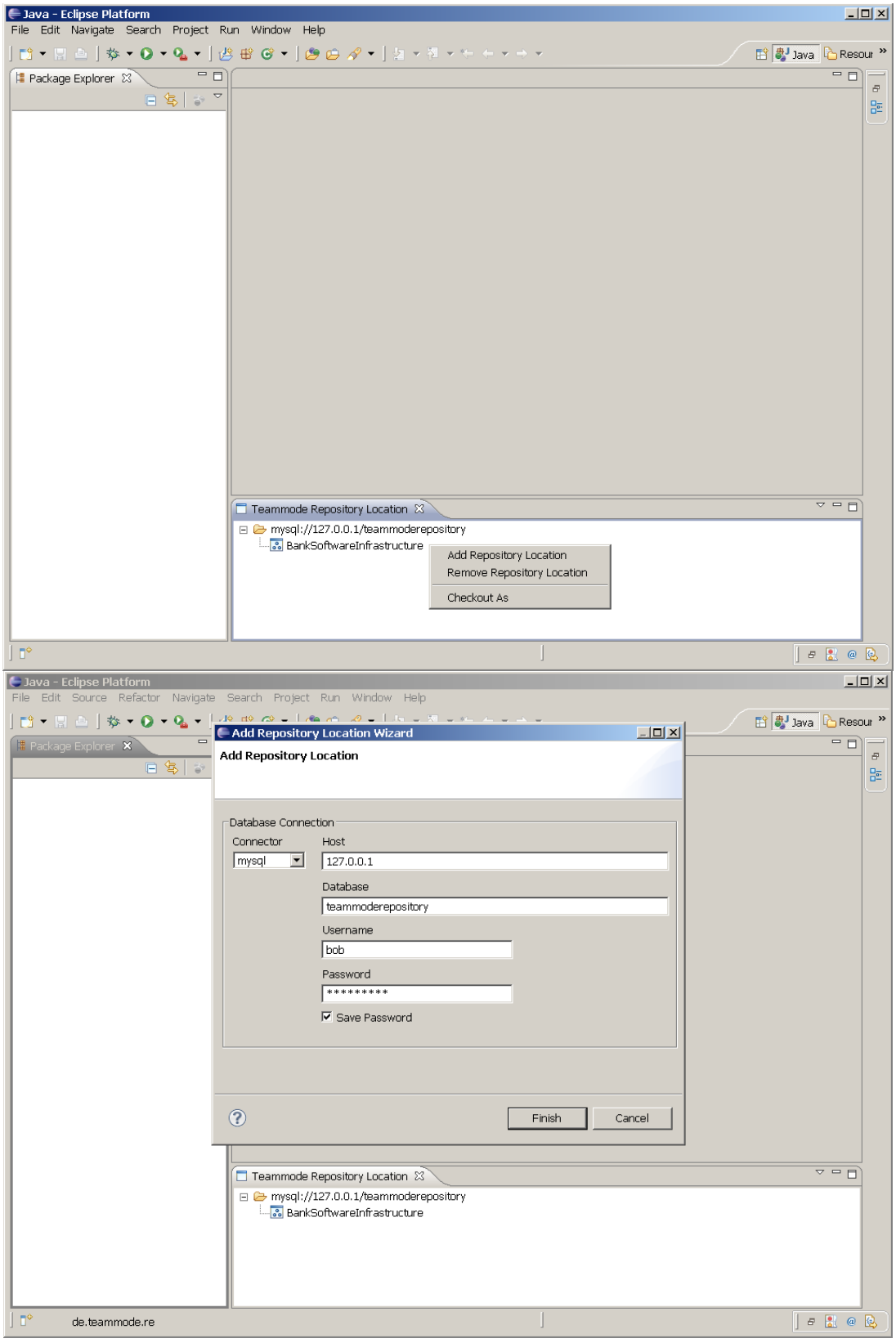


Abbildung V.2. Hinzufügen neuer Repositorys und Verbinden mit einem Repository

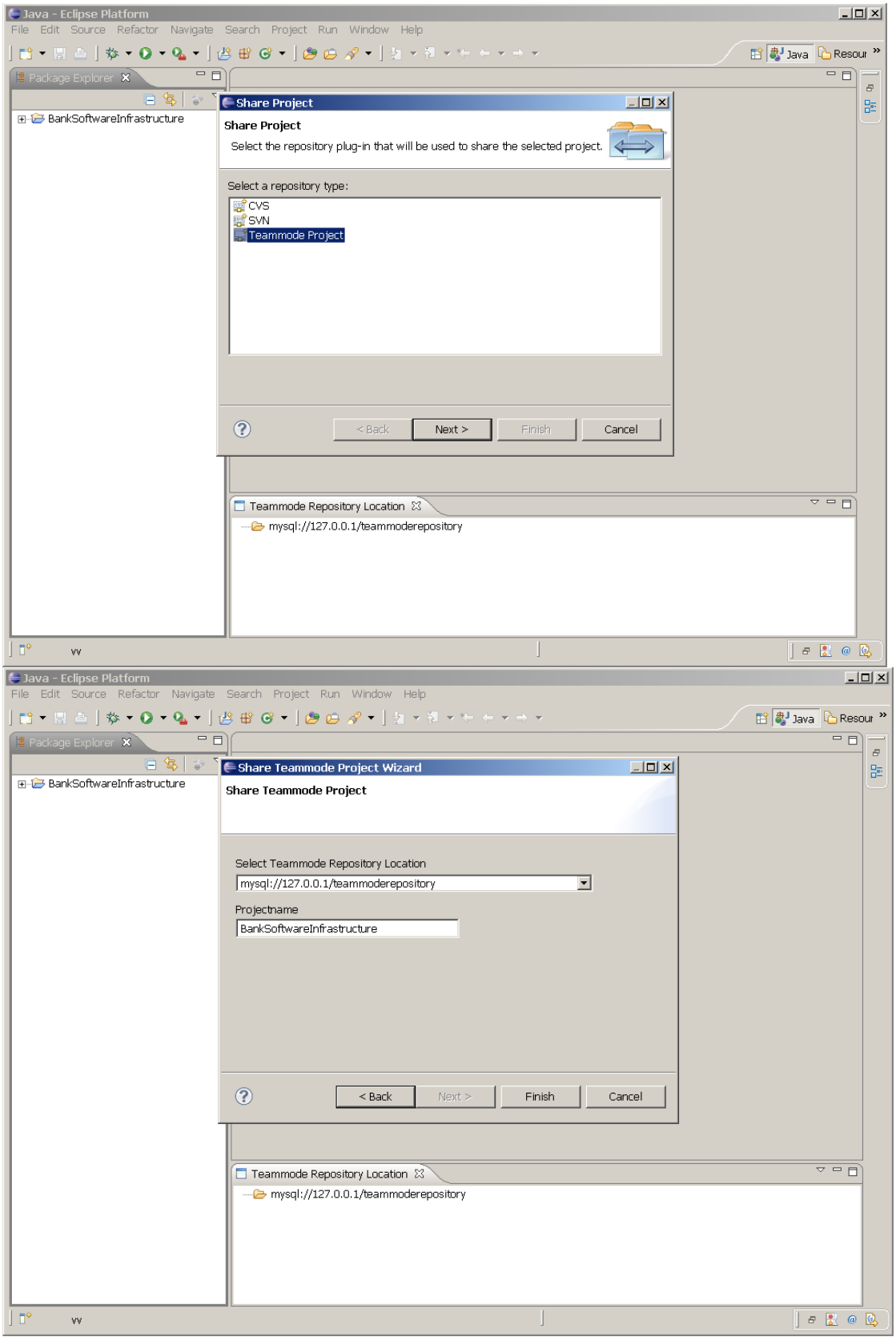


Abbildung V.3. Verbindung mit einem bestehenden Modell-Repository

1.2 Kollaboratives Modellieren mit Team.Mode

In einem Team.Mode-Projekt können in einer Verzeichnisstruktur Modelldateien verwaltet werden, die ggf. der gemeinsamen Bearbeitung zur Verfügung gestellt werden sollen. Wie in Abbildung V.4 zu sehen, sind die Daten des Team-weit verfügbaren Entwicklungszweiges im Verzeichnis „Team“ und die Daten der Modelliererin Alice im Verzeichnis „User (Alice)“ gespeichert. Wie in der Abbildung zu sehen, wurde im Team-weit verfügbaren Datenbestand auch das RE-Metamodell im Verzeichnis „Metamodell“ ECore/XMI-Datei abgelegt.

Über das Kontextmenü kann jeder Modellierer nun die üblichen Funktionen einer Versionsverwaltung ansteuern. Dabei kann er zwischen Team- und User-Funktionen wählen. Mit den Team-Funktionen (Team Update/Commit) wird der lokal auf dem Rechner des Benutzers gespeicherte Team-Datenbestand mit dem Stand des Team-Entwicklungszweiges auf dem Repository-Server synchronisiert. Mit der User-Funktion „User Commit“ kann der Modellierer lokale Modellversionen seiner Arbeit ins Repository übertragen, ohne sie für die Team-weite Änderung freizugeben.

Durch „User Update“ werden die lokalen Änderungen des Modellierers mit den Änderungen, die Teammitglieder im Team-weit verfügbaren Entwicklungszweig durchgeführt haben, integriert. Das Resultat wird zunächst lokal gespeichert, um ggf. merge-bedingte Fehler im Modell zu beheben; die Modellversion kann anschließend wieder mit „Team Commit“ allen zur Verfügung gestellt werden. Hierbei werden der Merge und die Validierung von Modellversionen, so wie sie in den letzten Kapiteln erörtert wurden, eingesetzt.

Neben den „Commit“ und „Update“-Funktionen können durch die Modellierer explizit neue Entwicklungszweige angelegt werden („Team/User Branch“), Modellversionen „gemerget“ werden („Team/User Merge“), lokale Versionen durch den Stand im Repository überschrieben werden („User Revert“) oder Versionen gelöscht werden („Team/User Delete“).

In dem in Abbildung V.4 dargestellten Beispiel werden zwei Dateien des bereits eingeführten BSI-Modells verwaltet. Dabei enthält die Datei „BankSoftwareInfrastructure.re“ die Modellinformationen nach dem RE-Metamodell im XMI-Format und die Datei „BankSoftwareInfrastructure.redi“ die Diagramminformationen zur konkreten Syntax des Modells (Position/Ausdehnung/Farbe der im Diagramm dargestellten Objekte).

Wie für die „Commit-Operation“ in Abbildung II.5 zu sehen, wird zunächst berechnet, welche lokal auf dem Rechner vorhandenen Versionen sich im Vergleich zum Stand im Repository verändert haben (in der Abbildung sowohl das Modell als auch die Diagramm-Datei). Nach der Transformation der Modellversionen in die MMR werden entsprechende Änderungen ins Repository übertragen. Da das Datenmodell des Repositorys nach dem Schema aus Abbildung IV.5 aufgebaut ist, wird dort dann eine neue Revision für das entsprechende Dokument angelegt.

Die Abbildung V.5 zeigt die jeweiligen Dialoge zum Erstellen eines neuen Entwicklungszweiges bzw. zum Merge zweier Versionen mit dem in der Dissertation entwickelten Merge-Verfahren. Dokumentversionen, die einzelnen Entwicklungszweigen zugeordnet sind, werden in dafür vorgesehenen Verzeichnissen verwaltet. Im Beispiel wurde das Unterverzeichnis „WithFrontend“ bspw. dafür verwendet.

Beim Merge zweier Versionen (vgl. Dialog in Abbildung V.5 unten) müssen die zu integrierenden Modellversionen explizit ausgewählt werden. Die Änderungen an der

Modellversion, die unter „source“ angegeben ist, werden dann in die Modellversion, die unter „Branch“ angegeben ist integriert.

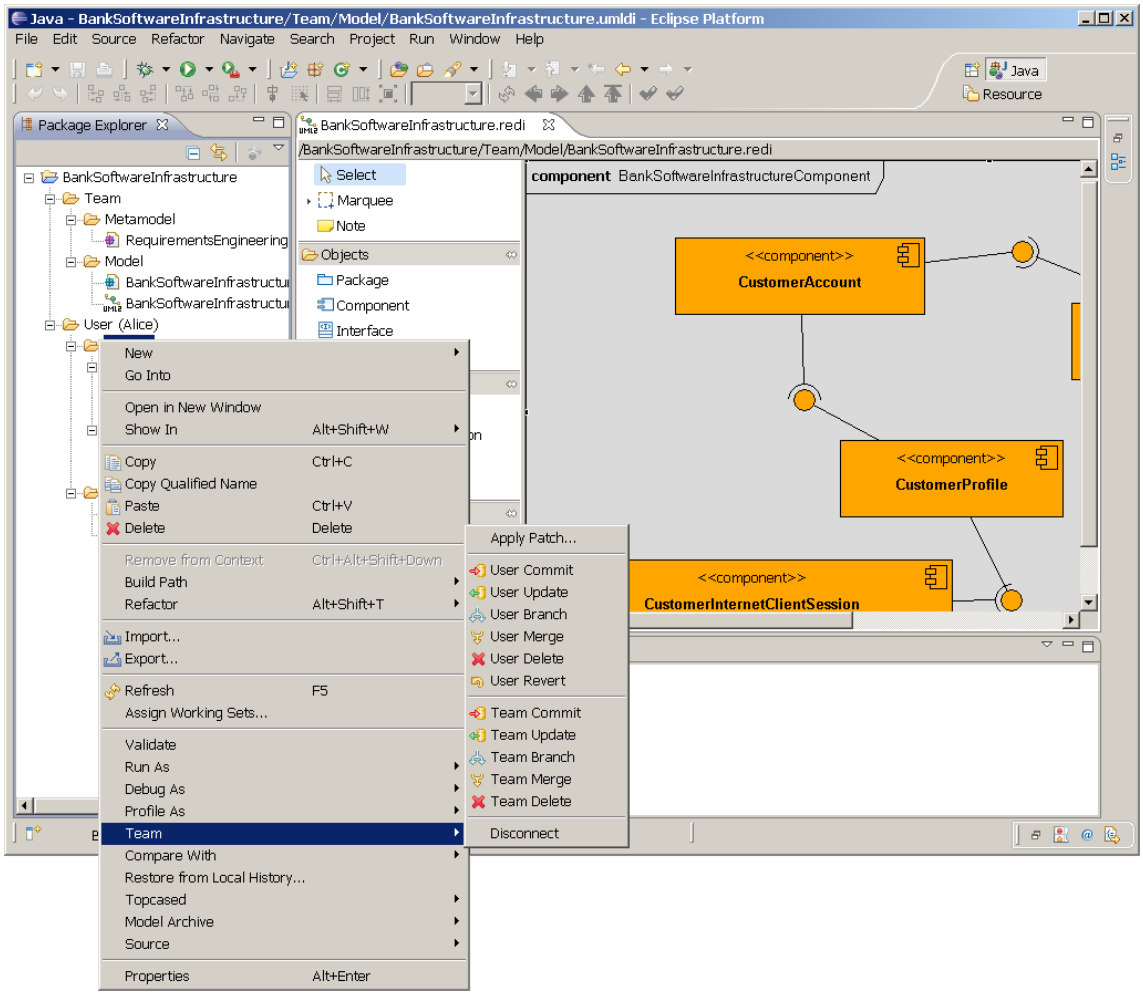


Abbildung V.4. Benutzerschnittstelle zur Synchronisation von Team.Mode-Modelldaten

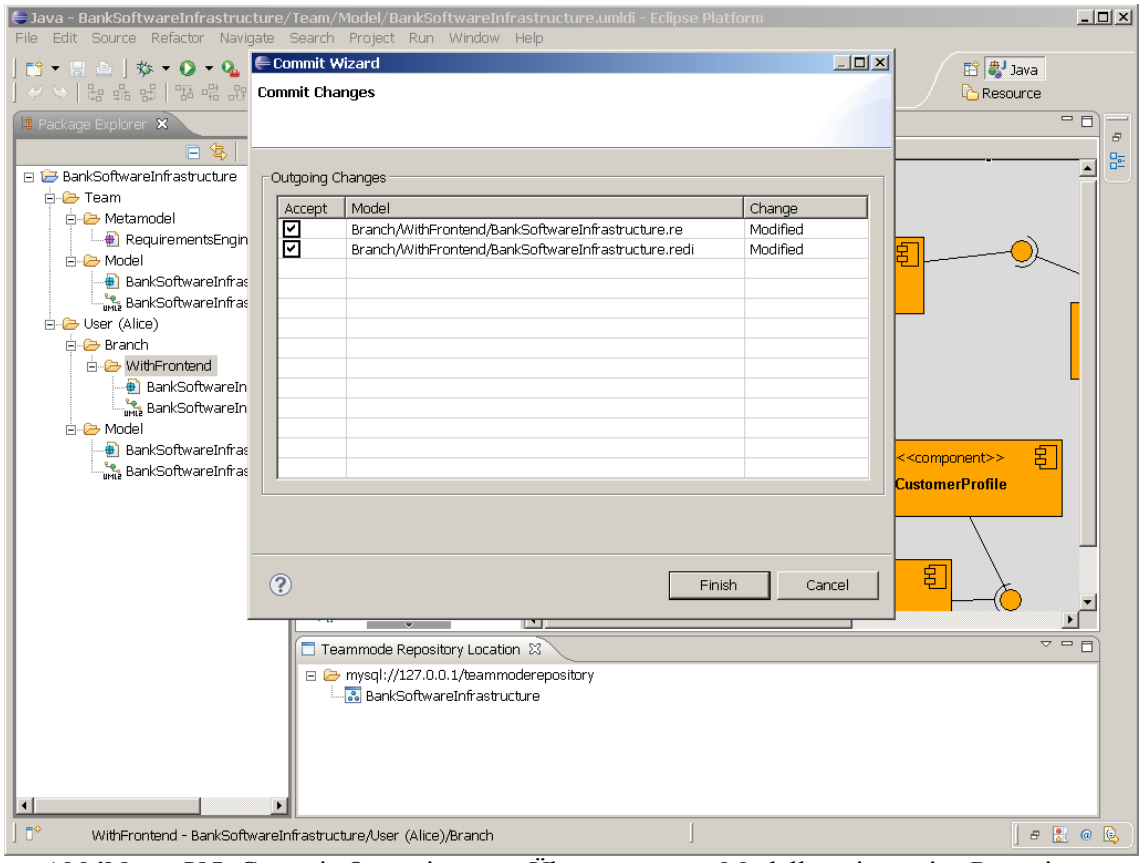


Abbildung V.5. Commit-Operation zum Übertragen von Modellversionen ins Repository

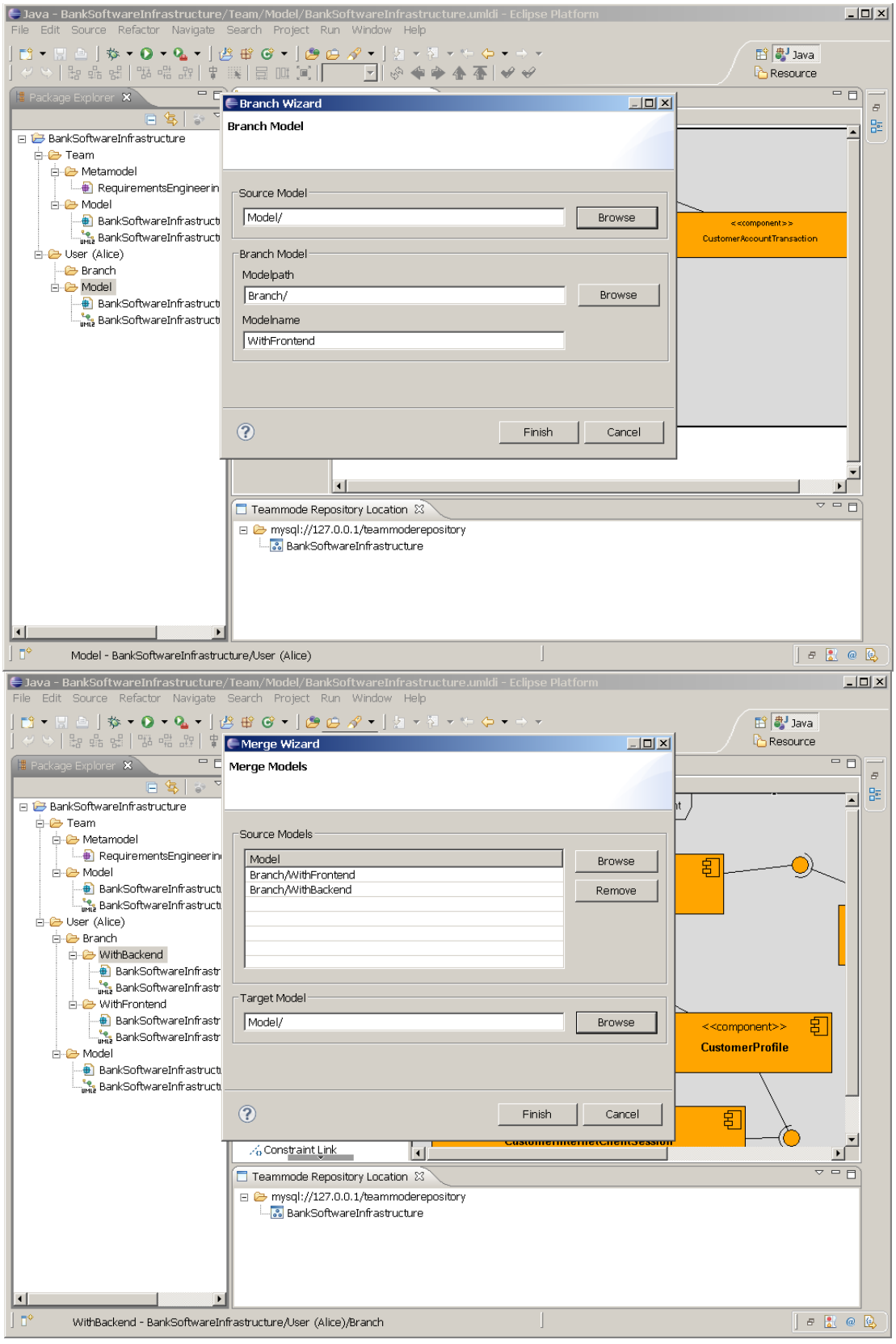


Abbildung V.6. Erstellen neuer Entwicklungszweige sog. Branches und Merge von Modellversionen

2 Werkzeuggestütze Analyse Merge-bedingter Integrationsprobleme

Im letzten Abschnitt wurde die Anwenderinteraktion mit Team.Mode zur Synchronisation eines Teams mithilfe eines zentralen Repositorys und insbesonders zur Integration paralleler Modelländerungen verdeutlicht. Im Abschnitt 2 soll nun die Leistungsfähigkeit von Team.Mode hinsichtlich der Analyse merge-bedingter Fehler in Modellversionen beleuchtet werden.

In Team.Mode wurde dazu der in Abschnitt IV.2 vorgestellte Ansatz implementiert. Dazu wurde ein Fehleranalyse-Editor entwickelt, der die beiden in Abbildung IV.11 dargestellten Phasen automatisiert. In Abbildung V.6 ist dieser dargestellt.

Um eine Fehleranalyse durchführen zu können, muss das Analysewerkzeug zunächst mit dem zu untersuchenden Modell und der Spezifikation der entsprechenden Modellierungssprache konfiguriert werden. Dazu wird, wie in der Mitte der Abbildung V.6 zu sehen, der „Open“-Dialog verwendet. In der ersten Zeile wird dazu das Metamodell der Modellierungssprache angegeben und in der zweiten das zu untersuchende Modell.

Als Bibliothek von SPARQL-Anfragen zur Fehlererkennung wird in der dritten Zeile ein Verzeichnis mit entsprechenden SPARQL-Dateien angeben. In der untersten Zeile kann eine zusätzliche OWL-Datei zur Anreicherung der TBox mit ergänzenden Bedingungen zur statischen Semantik der Modellierungssprache angegeben werden. Bspw. können das nach OWL manuell oder automatisch „übersetzte“ Bedingungen sein, die im Metamodell als OCL-Constraints ausgedrückt sind. Eine automatische Transformation solcher Bedingungen wurde allerdings im Rahmen dieser Dissertation nicht erforscht; sie ist aber prinzipiell möglich (vgl. (Milanović u. a. 2006)).

Nachdem die Software wie beschrieben konfiguriert wurde, lassen sich die für die zweite Phase (vgl. Abbildung V.6) vorgesehenen Analysealternativen durchführen. Dazu lässt sich die implementierte Generierung, die durch Interpretation IV.1 und Interpretation IV.2 beschrieben wurde, über den Button „Generate OWL2 DL TBox/ABox“ starten. Wie in Abbildung V.7 oben im zentralen Fenster des Editors zu sehen, wird danach das angegebene Metamodell und Modell gemeinsam in einer OWL-Wissensbasis dargestellt. Nach dem Start der Konsistenzanalyse des Reasoners durch „Run on generated OWL2 DL ABox“ wird im unteren Fenster des in Abbildung V.7 unten dargestellten Editors die Fehlerausgabe angezeigt. Erkannt wird der Fehler aus dem in den letzten Kapiteln verwendeten Modellierungsbeispiel hier durch den in Team.Mode integrierten Reasoner Pellet.

Auch der im letzten Kapitel diskutierte Ansatz zur Fehlererkennung mit SPARQL-Anfragen wurde in die Team.Mode-Validierung integriert. Dazu zeigt Abbildung V.9 den entsprechenden Tab „Extended Conjunctive Query Library“ der Analysesoftware. Im oberen Abschnitt der hier links zusehenden Navigationsleiste sind die möglichen, in der Anfrage-Bibliothek vorhandenen Anfragen zu sehen. Aus diesem Pool können einzelne oder alle Anfragen zur Analyse eines Modells ausgewählt werden.

In der Abbildung wurde u. a. auch die Anfrage „MultiplyContainment“ zur Analyse ausgewählt (vgl. Abschnitt IV.2.3.2). Diese ist im rechten Fenster dargestellt. Durch die Anwahl des Buttons „Run Extended Conjunctive Query Library“ werden die Anfragen automatisch an die Wissensbasis gestellt. Ein Ausschnitt aus dem Ergebnis der Analyse ist

unten im Ausgabefenster dargestellt. Neben dem Anwendungsfall *PrepareOffer* sind auch zwei Komponenten inkorrekterweise mit jeweils zwei Systemen assoziiert.

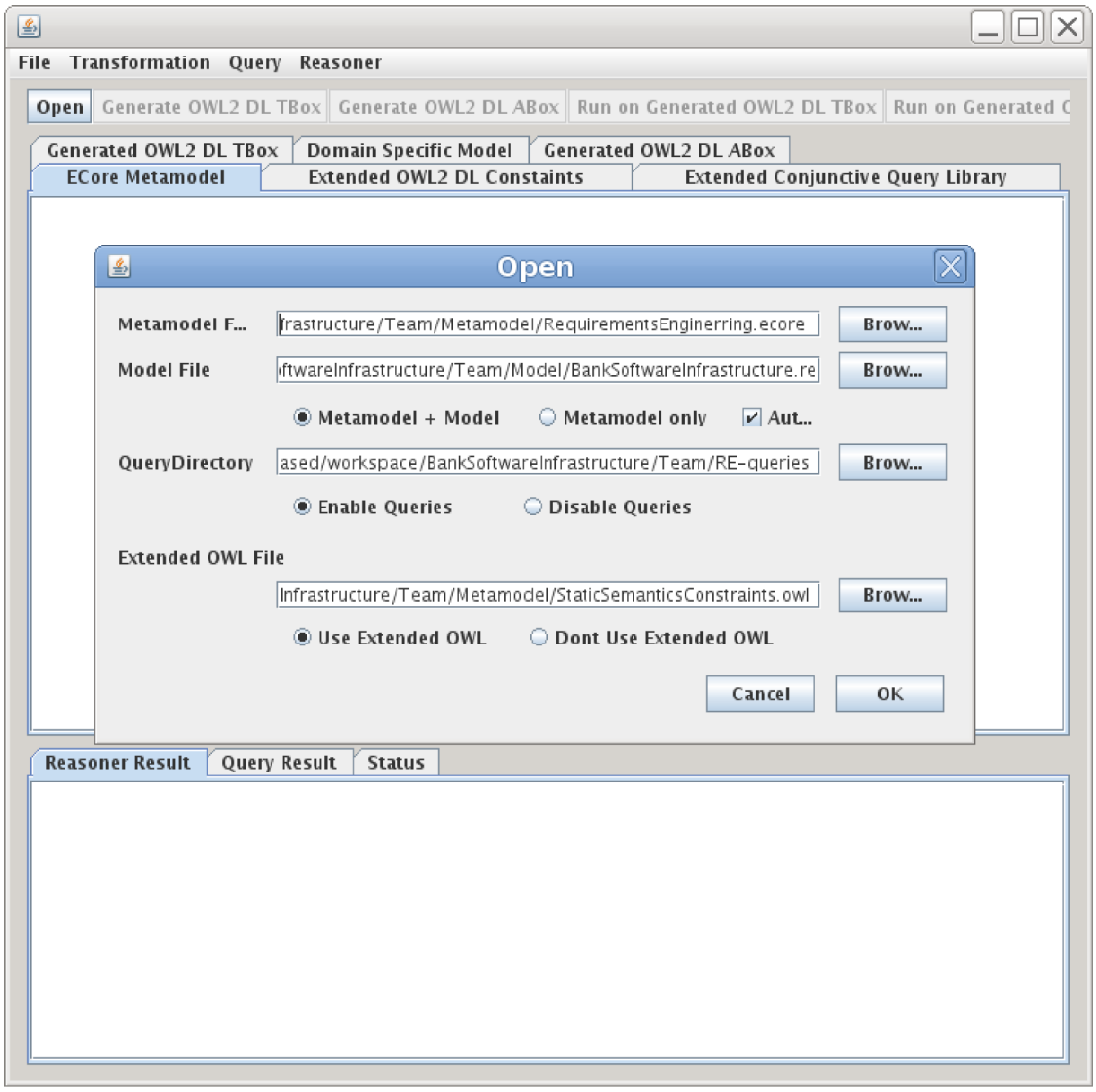


Abbildung V.7. Team.Mode Modellvalidierung – Dialog zur Zuweisung von Modell und Spezifikation der Modellierungssprache

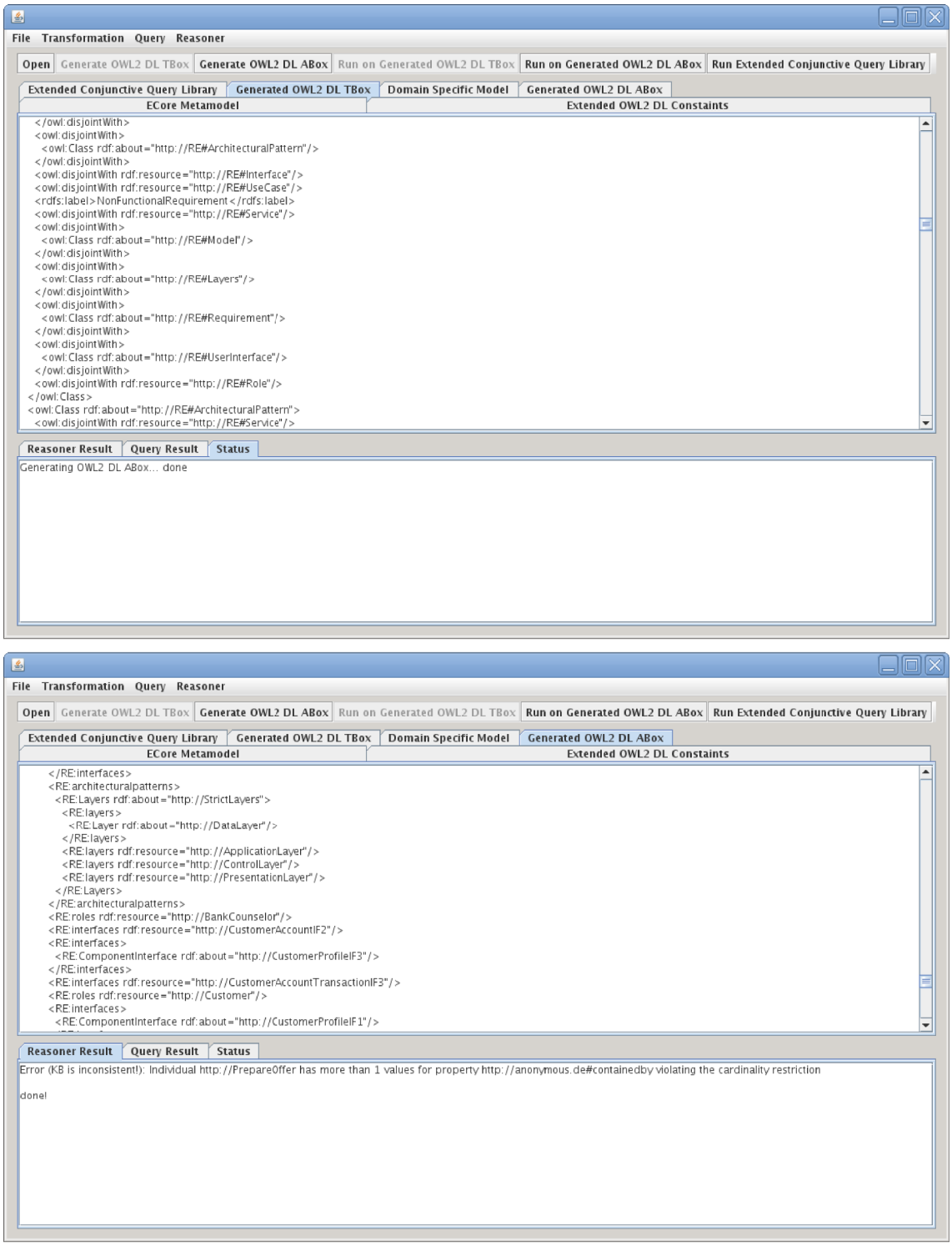


Abbildung V.8. Anzeige der generierten TBox und ABox aus Metamodell und Modell und anschließende Analyse der Wissensbasis mit dem integrierten Reasoner Pellet

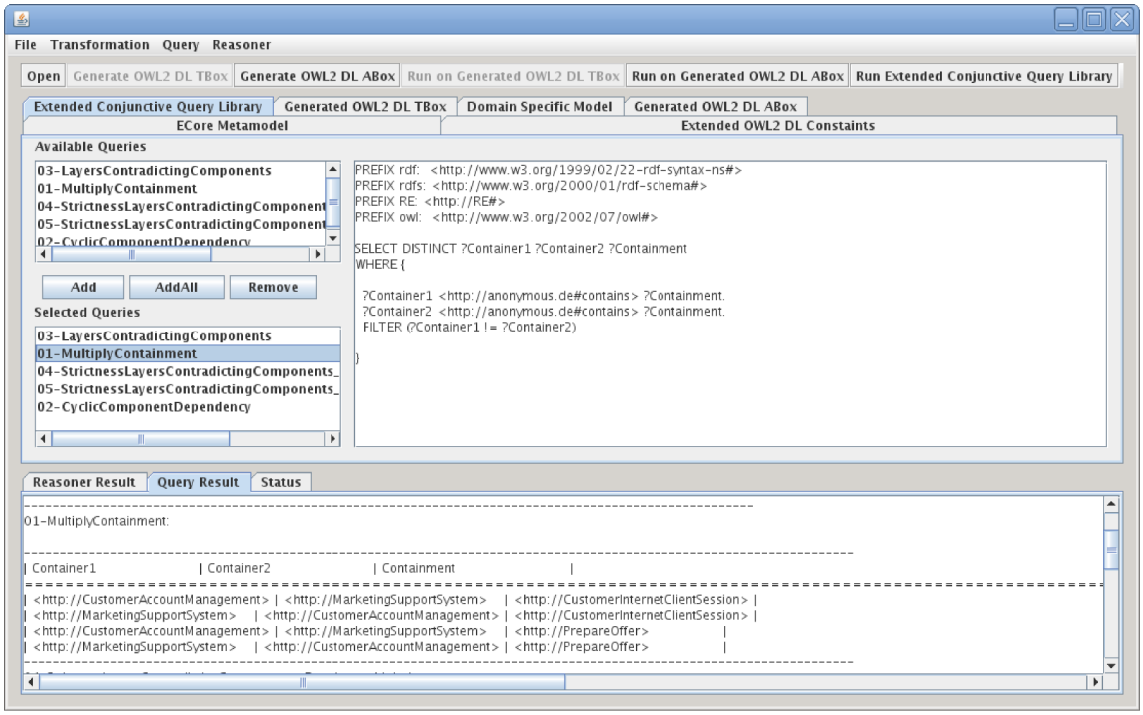


Abbildung V.9. Analyse der Wissensbasis durch SPARQL-Anfragen

Zur Durchführung der dritten in Abbildung IV.11 dargestellten Analysealternative per TBox-Reasoning zur Erkennung von Anomalien in Modellen muss die zu untersuchende Modellversion in eine beschreibungslogische TBox transformiert werden. Die dafür notwendige Transformation, so wie sie in Interpretation IV.3 dargestellt ist, muss dabei im Rahmen der Spezifikation der Modellierungssprache extern vorgegebenen sein. Nach dem das zu untersuchende Modell mit einer solchen Transformation in eine TBox umgewandelt wurde, lässt sich die Fehleranalyse, analog zu den Analysen oben, mit dem Button „Run on Generated OWL2 DL TBox“ starten.

Wie in den Darstellungen des Validierungseditors von Team.Mode oben zu sehen, werden die gefundenen Fehler zurzeit nur textuell den Anwendern angezeigt. Eine grafische Darstellung der Fehlersituation im grafischen Modell ist sicherlich wünschenswert und Gegenstand weiterer Arbeiten an Team.Mode.

3 Konzeptbewertung und Fazit

Die Entwicklung von Team.Mode orientierte sich maßgeblich an den publizierten Erfordernissen der modellbasierten Softwareentwicklung, in der die kollaborative Arbeit technisch bisher nicht hinreichend unterstützt wurde. Neben der Umsetzung der Anforderungen stand die Realisierung einer angemessenen Benutzungsschnittstelle, die an den Erfahrungshorizont von Softwareingenieuren anknüpft, im Vordergrund.

Da die Nutzer-Interaktion von Codeversionierungssystemen sich als sehr erfolgreich bewährt hat, implementiert Team.Mode eine weitgehend analoge Benutzungsschnittstelle. Ferner wurde darauf geachtet, dass schon die prototypische Implementierung von Team.Mode hinreichend große Softwaremodelle aus Anforderungsanalyse und Architekturentwurf performant verarbeiten kann.

Zur Bewertung der erforschten Ansätze wurde Team.Mode einer Reihe von Modellierungsszenarien unterzogen. In diesen wurden Modelle unterschiedlicher Größe und unterschiedlicher Modellierungssprachen (sowohl UML als auch DSLs) parallel bearbeitet. Dabei wurde Team.Mode zur rechnergestützten Verwaltung der entstandenen Modellversionen eingesetzt. Hierbei stand insbesondere die praktikable Nutzbarkeit bzgl. der Anwendungsfälle optimistischer Versionsverwaltungssysteme im Fokus. Zentrale Anwendungsfälle sind dabei die Integration asynchroner Modelländerungen durch die implementierten 3-Wege-Merge-Mechanismen und die Analyse merge-bedingter Verletzungen der syntaktischen Modellkorrektheit und weiterer Anomalien.

In prototypischen Modellierungsumgebungen wurde Team.Mode erfolgreich eingesetzt und hat bis heute alle gestellten Erwartungen erfüllt. Insbesondere stellt es eine Antwort auf die Anforderungen der Industrie (vgl. (Bendix und Emanuelsson 2009a; Bendix und Emanuelsson 2009b)) nach einer kollaborativen Modellierungsumgebung dar. Nach einer Erweiterung der Software um ein Modul zur bildlichen Darstellung der in der Modellvalidierung identifizierten Fehler und nach „Aushärtung“ der Implementierung steht dem produktiven Einsatz von Team.Mode in kollaborativen Modellierungsprojekten nichts mehr im Wege.

VI Zusammenfassung und Perspektiven

Welchen Beitrag leistet das beschriebene Verfahren zur Synchronisierung der Modellierungsarbeit eines Teams? Welche Alternativen zum verfolgten Lösungsansatz gibt es? Und wie wird der zukünftige Bedarf an automatischer Unterstützung in der kollaborativen Modellierung eingeschätzt? Um diese Fragen in Bezug zur vorgestellten Lösung zu setzen, werden in Abschnitt 1 zunächst die Ergebnisse der vorliegenden Arbeit zusammengefasst. Dabei wird insbesondere zu den eingangs erwähnten Zielen Bezug genommen.

Im Abschnitt 2 wird das Anwendungsgebiet beschrieben, in dem das konzipierte Verfahren unmittelbar eingesetzt werden kann und der entsprechende Nutzen für Modellierungsteams herausgestellt. Anschließend werden Lösungsalternativen und Erweiterungsmöglichkeiten zum vorgestellten Verfahren diskutiert, bevor abschließend die Perspektiven für eine weitreichende automatische Unterstützung der kollaborativen Modellierung erörtert werden.

1 Zusammenfassung der Ergebnisse

In der vorliegenden Dissertation wurde ein automatisches Verfahren zur Unterstützung der kollaborativ asynchronen Entwicklung von Softwaremodellen vorgestellt. Kern dieses Verfahrens ist die Unterstützung der inhaltlichen Synchronisierung von Modellversionen, die bei der parallel, asynchronen Entwicklungsarbeit verschiedener Softwareingenieure entstehen.

Als Lösung wird eine Versionsverwaltung für Modelle vorgeschlagen und konzipiert, die in Anlehnung an den Erfolg von Quellcodeverwaltungssystemen (engl. *Source Code Control System – SCCS*) eine optimistische Versionsverwaltung (Prudêncio, Murta und Werner 2009) realisiert. Diese nutzt ein zustandsbasiertes Merge-Verfahren zur Synchronisierung der Syntax von Modellversionen.

Dabei wird unabhängig von der verwendeten Modellierungssprache der abstrakte Syntaxgraph der zu verwaltenden Modellversionen in elementare syntaktische Einheiten zerlegt (sog. morphologische Dateneinheiten). Anhand dieser Zerlegung kann die Synchronisierung paralleler Modelländerungen mithilfe einer Realisierung des etablierten 3-Wege-Merge-Prinzips (Mens 2002) durchgeführt werden. Mit dem entwickelten Verfahren wird das eingangs

erörterte Ziel zur *automatischen Integration asynchroner Modellbearbeitungen auf Basis syntaktischer Modellbausteine* vollständig erfüllt.

Im Anschluss an die Integration paralleler Modelländerungen wird die Syntax der entstandenen Modellversion per statischer Analyse validiert. Ziel dieser Validierung ist die Erkennung inkonsistenter, parallel durchgeführter Modelländerungen. Bei der Prüfung wird die formale Syntaxspezifikation der jeweiligen Modellierungssprache zugrunde gelegt.

In Abschnitt IV.2 werden dabei Alternativen zur Validierung sowohl des abstrakten Syntaxgraphen auf Basis des durch die MOF spezifizierten Anteils des Metamodells als auch weiterer kontextsensitiver Korrektheitsbedingungen (sog. statische Semantik of OCL-Constraints) vorgestellt. Ferner können in diesem Rahmen mit statischen Analysen Anomalien im Modell wie die nicht gegebene Erfüllbarkeit/Instanziierbarkeit von Modellelementen (Verletzung Abbildungskriterium) erkannt werden.

Zur Validierung der Modellsyntax kommt dabei ein Verfahren zum Einsatz, dass sowohl das zu prüfende Modell als auch das zu Verfügung gestellte Metamodell (in XMI-Repräsentation) in beschreibungslogische Ausdrücke bzw. Anfragen (OWL/SPARQL-Repräsentation) umwandelt. Durch die vorgestellte statische Analyse der Modellsyntax können Fehler in Modellversionen erkannt werden, die auf die parallele Änderung der Modellsyntax zurückzuführen sind. Mit der Umsetzung dieses Ansatzes konnte das zweite Ziel zur weitgehenden Automatisierung der statischen Analyse von Modellen auf Basis ihrer formalen Syntax erreicht werden.

2 Anwendungsgebiet und Nutzen

In den meisten Softwareprojekten entwerfen insbesondere in den frühen Entwicklungsphasen mehrere Softwareingenieure im Team ein Modell des zu entwickelnden Systems. Hierbei kommen sowohl die UML als auch domänenspezifischen Modellierungssprachen zum Einsatz.

Es wechseln sich zudem synchrone Entwicklungsarbeiten, bei denen mehrere Modellierer zusammen am Whiteboard modellieren, mit asynchronen Entwicklungsarbeiten ab. In der asynchronen Entwicklungsarbeit ändern mehrere Modellierer parallel und unabhängig eine gemeinsame Modellversion. Um ihre Arbeiten zu integrieren, kann das erörterte Verfahren bzw. die Team.Mode-Software eingesetzt werden.

Voraussetzung dafür ist die Verwendung eines Modelleditors, der Modelle in der XMI-Repräsentation (Object Management Group (OMG) 2010a) speichert. Zusätzlich muss für die Durchführung der Korrektheitsanalyse das Metamodell der zu verwendenden Modellierungssprache mit der Meta-Metamodellierungssprache ECore spezifiziert worden sein, und ebenfalls als XMI-Datei vorliegen. Mit fast alle populären Modellierungswerkzeuge lassen sich heute diese Voraussetzungen erfüllen.

Da es sich um eine zentrale Versionsverwaltung für Modell handelt, benötigt das zu unterstützende Team noch einen zentralen Server auf dem die von Team.Mode genutzte Datenbank mit dem Versionsrepository zur Verfügung gestellt wird.

Ist die geforderte technische Infrastruktur eingerichtet, so kann jedes Teammitglied ein gemeinsames Modell lokal weiterentwickeln. Mithilfe des zentralen Repository-Servers können die parallel und asynchron erstellen Modellversionen inhaltlich auf Ebene einzelner

Modellelemente synchronisiert werden. Damit ermöglicht das Verfahren eine ähnlich kollaborative Zusammenarbeit in der Modellierung, wie sie in der Quellcodeverwaltung schon seit vielen Jahren geschätzt wird und erfüllt weitgehend die Anforderungen die diesbzgl. im wissenschaftlichen und industriellen Umfeld geäußert werden (Brunet u. a. 2006; Bendix und Emanuelsson 2009a).

3 Lösungsalternativen und Erweiterungspotenzial

Trotz der weitreichenden Unterstützung der kollaborativen Modellierungsarbeit ergeben sich zahlreiche Lösungsalternativen und Erweiterungsmöglichkeiten. Eine Auswahl davon wird in diesem Abschnitt überblicksartig beschrieben.

3.1 Lösungsalternativen

Wie schon in Abschnitt I.6 unter „Verwandte Arbeiten“ beschrieben wurde, werden im wissenschaftlichen Umfeld des Dissertationsthemas einige Alternativen zum vorgestellten Lösungsansatz erforscht. Diese sind dem hier vorgestellten Ansatz in einzelnen Aspekten überlegen, wirken sich aber in anderen Aspekten negativ aus. Die Auswahl des geeigneten Verfahrens erfordert somit eine Abwägung. Auf die Alternativen und ihre charakteristischen Eigenschaften soll nachfolgend kurz eingegangen werden. Eine umfassende, belastbare und vergleichende Evaluierung der Ansätze im industriellen Umfeld steht aufgrund des fehlenden Reifegrades entsprechender Implementierungen bisher noch aus.

3.1.1 Verstärkung synchroner Modellierungsarbeit

Alternativ zur asynchronen Entwicklungsarbeit, die durch die Ergebnisse der Dissertation unterstützt wird, kann auch eine verstärkte synchrone Modellierung favorisiert werden. Dieser Ansatz hat den Charme, dass ein komplexes Merge-Verfahren für Modelle entfallen kann, da bei synchronen Modellierungsaktivitäten alle Modellierer gleichzeitig am Modell arbeiten (teils auch geografisch verteilt bspw. über elektronische Whiteboards) und die Änderungen der anderen Teammitglieder „live“ mitbekommen. Sie integrieren somit durch jeden ihrer Bearbeitungsschritte ihre Modelländerungen in den teamweit verfügbaren Modellierungsstand. Damit ähnelt die Arbeitsweise der eines Wikis (Leuf und Cunningham 2001).

Der Vorteil, auf einen komplexen Modell-Merge verzichten zu können, wird allerdings mit einer wesentlichen Einschränkung erkauft: Alle Änderungen müssen immer unmittelbar in den teamweiten Modellierungsstand eingepflegt werden. Dies ist aber nicht immer erwünscht und sinnvoll. Insbesondere, wenn einzelne Modellierer erst komplexe (teils experimentelle) Modellbearbeitungen in mehreren Schritten ins Modell einpflegen wollen, ist es nicht sinnvoll, unvollständige Teilbearbeitungen am „globalen“ Stand des gesamten Teams zuzulassen. Um parallel isolierte Änderungen am Modell vornehmen zu können, benötigt man hingegen die gleichen Merge-Mechanismen, wie bei der asynchronen Modellierungskollaboration.

3.1.2 Operationsbasierte Synchronisation von Modellversionen

Operationsbasierte Verfahren zur Integration kollaborativ bearbeiteter Modellversionen erlauben eine detailliertere und ausdrucksstärkere Analyse inkonsistenter Änderungen, ein teils intelligenteres Merge-Verfahren für Modelle und ein höheres Potenzial die Korrektur inkonsistenter Änderungen automatisch zu unterstützen (Koegel, Hermannsdoerfer, Yang, Helming und David 2010; Koegel, Helming und Seyboth 2009). Die Typen von spezifischen

Änderungsoperationen sind je nach Modellierungssprache unterschiedlich und müssen durch das von den Modellierern verwendete Case-Tool unterstützt werden. Nur mit einer entsprechenden Aufzeichnung und Speicherung der spezifischen Änderungsoperationen durch das Case-Tool lassen sich die Stärken operationsbasierter Verfahren entsprechend ausnutzen.

Da es bisher noch keinen Standard zur Spezifikation modellierungssprachspezifischer Änderungsoperationen gibt, können nur modellierungswerkzeugspezifische Änderungsoperationen genutzt werden. Die Anwendbarkeit des Verfahrens hängt damit stark von den Fähigkeiten des Modellierungswerkzeugs ab, Änderungsoperationen während der Bearbeitung durch die Modellierer aufzuzeichnen und zu speichern. Sie ist damit auch stark gekoppelt an ein werkzeugspezifisches Format zur Speicherung von Änderungshistorien. Die Verwendung beliebiger XMI/ECore-unterstützender Modellierungswerkzeuge durch die Entwickler ist somit nicht möglich.

Wie in den Rahmenbedingungen in Abschnitt I4.1 erläutert, soll mit dieser Dissertation ein weitgehend werkzeugunabhängiger Ansatz zur Unterstützung der kollaborativen Modellierung vorgestellt werden. Aus diesem Grund wurde statt eines operationsbasierten Verfahrens ein zustandsbasiertes Merge-Verfahren favorisiert.

3.1.3 Einsatz von Sperrmechanismen

Eine dritte Alternative zur Synchronisation asynchroner Modellierungsarbeit ist der Einsatz von Sperrmechanismen im Sinne der pessimistischen Versionsverwaltung. Werkzeuge wie (MID GmbH 2010) gewähren einzelnen Modellierern ein exklusives Bearbeitungsrecht an bestimmten Teilen des teamweiten Modells. Für alle anderen Modellierer ist dann der entsprechende Teil des Modells zur Bearbeitung gesperrt. Durch diese Strategie werden inkonsistente Modelländerungen vermieden, die entstehen würden, wenn mehrere Softwareingenieure komplexe Bearbeitungen an denselben Modellteilen vornehmen würden.

Neben diesem Ansatz werden nach wie vor pessimistische Versionsverwaltungsstrategien zur Verwaltung von Daten eingesetzt, die sich nicht geeignet mit Text-Merge-Verfahren verarbeiten lassen.

Wegen den in Abschnitt II.2.2 ausführlich erörterten Nachteilen pessimistischer Strategien wurde in dieser Arbeit die Entwicklung eines optimistischen Versionierungsverfahrens angestrebt.

3.2 Erweiterungsmöglichkeiten

Die umfassende Unterstützung kollaborativer Modellierungsarbeit durch automatische Verfahren befindet sich zurzeit noch am Beginn ihrer Entwicklung. Die Ergebnisse dieser Dissertation bilden in diesem Kontext einen ersten grundlegenden Baustein zur Ergänzung weiterer Forschungs- und Entwicklungsansätze. Eine Auswahl interessanter und wesentlicher Erweiterungen wird anschließend skizziert:

3.2.1 Erweiterungspotenzial bei der Synchronisierung der visuellen Modelldarstellung

Das in der Dissertation vorgestellte Verfahren unterstützt die automatische Synchronisierung der abstrakten Syntaxgraphen von Modellversionen. Aktuell wird die Synchronisierung der konkreten Modellsyntax (bspw. Form und Position grafischer Symbole in der visuellen

Modelldarstellung) im Falle der parallelen Bearbeitung gleicher Modellelemente nur teilweise unterstützt.

Zur weiteren Steigerung der praktikablen Anwendbarkeit sollte die Implementierung von Team.Mode um eine erweiterte Merge-Unterstützung für Modelldarstellungen (konkrete Modellsyntax) erweitert werden.

3.2.2 Erweiterungspotenzial bei der visuellen Darstellung inkonsistenter Änderungen

Mit Team.Mode werden zurzeit Fehler, die durch die Modellvalidierung nach einem Merge auftreten, nur in einer textuellen Darstellung aufgelistet. Anhand dieser Darstellung kann man die inkonsistenten Änderungen in den betreffenden parallelen Entwicklungszweigen erkennen.

Zur weiteren Steigerung der Anwenderfreundlichkeit sollte Team.Mode um eine grafische Darstellung zur Illustration inkonsistenter Änderungen erweitert werden. Konzepte dafür wurden bereits entwickelt und prototypisch implementiert (Bartelt und Schindler 2010).

3.2.3 Erweiterungspotenzial bei der Konsistenzanalyse paralleler Änderungen

Die Konsistenzanalyse paralleler Modelländerung beruht nach dem vorgestellten Ansatz auf einer Validierung des abstrakten Syntaxgraphen eines gemergten Modells gegen die Metamodellspezifikation der jeweiligen Modellierungssprache. Dabei wird aktuell der Anteil des Metamodells, der per MOF-Modell ausgedrückt werden kann, in Ausdrücke einer beschreibungslogischen Wissensbasis und in Anfragen automatisch umgewandelt.

Weitere Nebenbedingungen der statischen Semantik, die in vielen Metamodellen durch OCL-Constraints ausgedrückt werden, müssen zurzeit manuell in beschreibungslogischen Ausdrücken bzw. Anfragen formuliert/umgewandelt werden. Zur Verbesserung des Validierungsverfahrens kann auch die Umwandlung der OCL-Constraints automatisiert werden. Forschungsansätze für eine derartige Abbildung existieren bereits (Milanović u. a. 2006; Kalibatiene und Vasilecas 2010).

3.2.4 Unterstützung der Co-Evolution von Metamodellen und Modellen

Die in Kapitel IV vorgestellten Lösungsansätze eignen sich, um nicht nur die kollaborative Entwicklung eines Modells bei einem über die Entwicklungszeit unveränderten Metamodell zu unterstützen sondern auch die Evolution des jeweiligen Metamodells mit einzubeziehen.

Dies ist umso wichtiger, wenn man berücksichtigt, dass auch Modellierungssprachen einem kontinuierlichen Entwicklungsprozess unterzogen sind. Bei einer entsprechenden Anwendung der vorgestellten Lösungsansätze kann dazu die Entwicklung des Metamodells als Veränderung eines Modells, dessen Metamodell durch die MOF bestimmt ist, angesehen werden.

Zusätzlich zur Unterstützung beliebiger Modellierungssprachen bei der Verwaltung von Modellen wäre demzufolge auch die Unterstützung der Co-Evolution von Modellen und ihren Metamodellen eine wünschenswerte Erweiterung der vorgestellten Arbeit.

3.2.5 Einbettung in ganzheitliche Entwicklungsumgebungen

Die optimistische Versionsverwaltung von Quellcode ist mehr und mehr Bestandteil ganzheitlicher Entwicklungsumgebungen im Software Engineering. Systeme wie (IBM 2010b;

Microsoft 2010) integrieren verstärkt Programmieraktivitäten mit Testen, Projekt- und Aufgabenplanung, Qualitätssicherung und Software Configuration Management (SCM). Wie in (Bartelt, Molter und Schumann 2009) beschrieben, ist die Einbettung einer werkzeuggestützten Kollaborationsunterstützung für die Modellierung ebenfalls eine sinnvolle Erweiterung zur Bereitstellung einer ganzheitlichen Entwicklungsumgebung, die den gesamten Softwareentwicklungsprozess unterstützt.

4 Perspektiven – Entwicklung von Modellen im Team

In (Bartelt u. a. 2009) wurde ausgeführt, dass die verteilte, hochgradig parallele Entwicklung in teils organisationsübergreifenden Softwareprojekten keine singuläre Erscheinung ist, sondern inzwischen die Regel. Es ist abzusehen, dass die Treiber dieses Trends mit steigender organisatorischer Vernetzung und wirtschaftlicher Globalisierung sich weiter verstärken werden.

Vor diesem Hintergrund wird der Erfolg von Softwareprojekten zunehmend auch davon abhängen, wie gut man in der Lage sein wird, parallele Entwicklungsarbeit inhaltlich korrekt zu synchronisieren. Wie die Verbreitung von Versionsverwaltungssystemen in der Quellcodeverwaltung zeigt, kann eine Werkzeugunterstützung die inhaltliche Synchronisierung von Softwaredokumenten entscheidend verbessern.

Da bislang die Entwicklung einer vergleichbaren Unterstützung für die Modellverwaltung noch am Anfang steht, ergeben sich weitreichende Möglichkeiten für Forschung und Praxis in diesem Bereich. Mit dieser Dissertation wurde dieser Entwicklung ein Baustein hinzugefügt, der eine Grundlage für die Werkzeugunterstützung in der kollaborativen Modellierung bildet.

Bibliografie

AIFB Karlsruhe. 2010. KAON2 - Ontology Management for the Semantic Web. http://kaon2.semanticweb.org/#literature, Stand 8.12.2010.

Alanen, Marcus und Ivan Porres. 2003. Difference and Union of Models. *In UML 2003 - The Unified Modeling Language*, 2863/2003:2-17. Lecture Notes in Computer Science. Turku, Finland: Springer-Verlag.

Altmanninger, Kerstin. 2008. Models Models in Conflict - Towards a Semantically Enhanced Version Control System for Models. *In Models in Software Engineering*, hg. v. Holger Giese. Bd. 5002. Berlin, Heidelberg: Springer Berlin Heidelberg.

Altmanninger, Kerstin, Martina Seidl und Manuel Wimmer. 2009. A survey on model versioning approaches. *International Journal of Web Information Systems* 5, no. 3: 271-304.

Baader, Franz, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi und Peter F. Patel-Schneider. 2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Baecker, Ronald M. 1992. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*. 1. Aufl. Morgan Kaufmann.

Balzert, Helmut. 1997. *Lehrbuch der Software-Technik, Software-Management, Software-Qualitätssicherung und Unternehmensmodellierung*. Bd. 2. Spektrum-Akademischer Verlag.

Bartelt, Christian. 2008a. An Optimistic Three-way Merge - Based on a Meta-Model Independent Modularization of Models to Support Concurrent Evolution. *In Proceedings of MODSE '08 - Workshop on Model-Driven Software Evolution at 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*. New York, NY, USA: IEEE.

Bartelt, Christian. 2008b. Consistence preserving model merge in collaborative development processes. *In Proceedings of International Workshop on Comparison and versioning of software models - CVSM '08 at International Conference on Software Engineering (ICSE '08)*, 13. Leipzig, Germany.

Bartelt, Christian, Manfred Broy, Christoph Herrmann, Eric Knauss, Marco Kuhrmann, Andreas Rausch, Bernhard Rumpe und Kurt Schneider. 2009. Orchestration of Global Software Engineering Projects - Position Paper, 332-337. IEEE Computer Society.

Bartelt, Christian, Georg Molter und Tim Schumann. 2009. A Model Repository for Collaborative Modeling with the Jazz Development Platform. *In Proceedings of Collaborative Modeling Track at 42th Hawaii International Conference on System Sciences*, 0:1-10. Los Alamitos, CA, USA: IEEE Computer Society.

Bartelt, Christian und Bjorn Schindler. 2010. Technology Support for Collaborative Inconsistency Management in Model Driven Engineering. *In Proceedings of Collaborative Modeling Track at 43rd Hawaii International Conference on System Sciences (HICSS-43)*, 0:1-7. Los Alamitos, CA, USA: IEEE Computer Society.

Bendix, Lars. 1995. Configuration Management and Version Control Revisited. Aalborg, Denmark: Ph.D. Dissertation, Aalborg University.

Bendix, Lars und Pär Emanuelsson. 2008. Diff and merge support for model based development. *In Proceedings of International workshop on Comparison and versioning of software models - CVSM '08*, 31-34. New York, NY, USA: ACM.

Bendix, Lars und Pär Emanuelsson. 2009a. Requirements for Practical Model Merge - an Industrial Perspective. *In Proceedings of 12th International Conference on Model Driven Engineering Languages and Systems - MODELS '09*, 5795:167 - 180. Lecture Notes In Computer Science. Berlin, Heidelberg: Springer-Verlag.

Bendix, Lars und Pär Emanuelsson. 2009b. Collaborative work with Software Models - Industrial experience and requirements. *In Proceedings of International Conference on Model-Based Systems Engineering - MBSE '09.*, 60-68. IEEE Press.

Bendix, Lars, Maximilian Koegel und Antonio Martini. 2010. The Case for Batch Merge of Models – Issues and Challenges. *In Proceedings of International Workshop on Models and Evolution - ME 2010*. Oslo, Norway.

Berardi, Daniela, Diego Calvanese und Guiseppe De Giacomo. 2001. Reasoning on UML Class Diagrams using Description Logic Based Systems. *In Proceedings of the KI'2001 Workshop on Applications of Description Logics* 44. CEUR Electronic Workshop Proceedings.

Berardi, Daniela, Diego Calvanese und Giuseppe De Giacomo. 2005a. Reasoning on UML class diagrams. *Artif. Intell.* 168, no. 1: 70-118.

Berardi, Daniela, Diego Calvanese und Giuseppe De Giacomo. 2005b. Reasoning on UML class diagrams. *Artifical Intelligence* 168, no. 1: 70-118.

BitMover Inc. 2010. BitKeeper - The Scalable Distributed Software Configuration Management System. http://www.bitkeeper.com/, Stand 21.11.2010.

Blanc, Xavier, Isabelle Mounier, Alix Mougenot und Tom Mens. 2008. Detecting model inconsistency through operation-based model construction. *In Proceedings of 13th international conference on Software engineering - ICSE '08*, 511. Leipzig, Germany.

Booch, Grady, James Rumbaugh und Ivar Jacobson. 1998. *Unified Modeling Language User Guide*. 1. Aufl. Pearson.

Borgida, Alexander, Ronald J. Brachman, Deborah L. McGuinness und Lori Alperin Resnick. 1989. CLASSIC: a structural data model for objects. *In Proceedings of ACM SIGMOD international conference on Management of data - SIGMOD '89*, 58-67. Portland, Oregon, United States.

Borland. 2010a. BorlandTogether. http://www.borland.com/de/products/together/, Stand 21.11.2010.

Borland. 2010b. Borland StarTeam. http://www.borland.com/de/products/starteam/, Stand 24.11.2010.

Brachman, Ronald J. und James G. Schmolze. 1985. An overview of the KL-ONE Knowledge Representation System. *Cognitive Science* 9, no. 2: 171-216.

Brosch, Petra. 2009. Improving conflict resolution in model versioning systems. *In Proceedings of 31st International Conference on Software Engineering - Companion Volume, 2009*, 355 - 358. Vancouver, BC.

Brunet, Greg, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu und Mehrdad Sabetzadeh. 2006. A manifesto for model merging, 5-12. New York, NY, USA.

Buffenbarger, Jim. 1995. Syntactic Software Merging. *In Proceedings of Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, 153-172.

Canonical Ltd. 2010. Bazaar. http://bazaar.canonical.com/en/, Stand 24.11.2010.

Chen, Peter Pin-shan. 1976. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems* 1: 9-36.

Christian Schneider, Albert Zündorf und Jörg Niere. 2004. CoObRA - a small step for development tools to collaborative environments. *In Proceedings of Workshop on Directions in Software Engineering Environments in 26th international conference on software engineering*. Edinburgh, Scotland, UK.

Clark, A., A. Evans und S. Kent. 2002. Engineering Modelling Languages: A Precise Metamodelling Approach. *In Proceedings of 5th International Conference on Fundamental Approaches to Software Engineering - FASE '02*, 2306:159-173. Fundamental Approaches to Software Engineering, LNCS. Berlin, Germany: Springer-Verlag.

Clatworthy, Ian. 2007. Distributed Version Control Systems - Why and How.

Collins-Sussman, Ben. 2002. The subversion project: building a better CVS. *Linux Journal*.

Conradi, Reidar und Bernhard Westfechtel. 1997. Towards a Uniform Version Model for Software Configuration Management. *In Proceedings of the 7th International Workshop on Software Configuration Management* 1235. Lecture Notes in Computer Science: 1-17.

Conradi, Reidar und Bernhard Westfechtel. 1998. Version models for software configuration management. *ACM Computing Surveys* 30, no. 2: 232-282.

Dantas, Cristine R., Leonardo G. P. Murta und Cláudia M. L. Werner. 2005. Consistent Evolution of UML Models by Automatic Detection of Change Traces. *In Proceedings of International Workshop on Principles of Software Evolution*, 0:14-147. Los Alamitos, CA, USA: IEEE Computer Society.

DASI-lab group - University of Rome und IBM Center for Advanced Studies (CAS). 2010. QuOnto. http://www.dis.uniroma1.it/~quonto/, Stand 27.9.2010.

Deiters, Constanze, Patrick Dohrmann, Sebastian Herold und Andreas Rausch. 2009. Rule-based architectural compliance checks for enterprise architecture management. *In Proceedings of 13th IEEE international conference on Enterprise Distributed Object Computing*, 158-167. Auckland, New Zealand: IEEE Press.

Demuth, Birgit. 2004. The Dresden OCL Toolkit and its Role in Information Systems Development. *In Proceedings of 13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice Conference, Advances in Theory, Practice and Education (ISD'2004)*. Vilnius, Lithuania.

Eclipse Foundation. 2010. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/, Stand 29.11.2010.

Elaasar, M. und L. Briand. 2004. *An Overview of UML Consistency Management*. Technical Report. Ottawa, Canada: , Carleton University, Department of Systems and Computer Engineering.

Espinazo-Pagán, Javier und Jesús García-Molina. 2010. A homogeneous repository for collaborative MDE. *In Proceedings of 1st International Workshop on Model Comparison in Practice - IWMCP '10*, 56. Malaga, Spain.

Estublier, Jacky. 2000. Software Configuration Management: A Roadmap. ICSE - Future of SE Track: 279-289.

Estublier, Jacky, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter F. Tichy und Darcy Wiborg-Weber. 2005. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology* 14, no. 4: 383-430.

Fikes, Richard und Ian Horrocks. 2003. *OWL-QL: A Language for Deductive Query Answering on the Semantic Web*. Stanford, CA: Knowledge Systems Laboratory, Stanford University.

Fleurey, Franck, Benoit Baudry, Robert France und Sudipto Ghosh. 2007. A Generic Approach For Automatic Model Composition. *In Proc. AOM at Models* 7: 2007.

Frühauf, Karol und Andreas Zeller. 1999. Software Configuration Management: State Of The Art, State Of The Practice. *Proceedings of the Dagstuhl Seminar* 254.

Fujaba development group. 2010. Fujaba Tool Suite. http://www.fujaba.de/, Stand 30.11.2010.

Gentleware. 2010. Poseidon for UML 8.0. http://www.gentleware.com/, Stand 21.11.2010.

Git Authors. 2010. Git - Fast Version Control System. http://git-scm.com/, Stand 24.11.2010.

GNU Project. 2010a. Revision Control System - GNU RCS. http://www.gnu.org/software/rcs/, Stand 28.11.2010.

GNU Project. 2010b. CVS - Open Source Version Control. http://www.nongnu.org/cvs/, Stand 21.11.2010.

Gries, Oliver. 2009. Generalized Closed World Reasoning in Description Logics with Extended Domain Closure. *In Proceedings of 22nd International Workshop on Description Logics - DL2009*. Bd. 477. Oxford, UK: CEUR.

Grimm, Stephan und Boris Motik. 2005. Closed world reasoning in the semantic web through epistemic operators. *OWL: Experiences and Directions*.

Grune, Dick. 1986. Concurrent Versions System, A Method for Independent Cooperation. *IR 113, VRIJE UNIVERSITEIT*.

Haarslev, Volker und Ralf Möller. 2001. *RACER system description*. Hamburg: University of Hamburg.

Haarslev, Volker, Ralf Möller und Michael Wessel. 2004. Querying the Semantic Web with Racer nRQL. *In Proceedings of KI-2004 International Workshop on Applications of Description Logics (ADL'04*.

Hammarström, Göran. 1966. *Linguistische Einheiten im Rahmen der modernen Sprachwissenschaft*. Springer.

Helming, Jonas, Maximilian Koegel, Helmut Naughton, Joern David und Aleksandar Shterev. 2009. Traceability-Based Change Awareness. *In Proceedings of 12th International Conference on Model Driven Engineering Languages and Systems*, 372-376. Denver, CO: Springer-Verlag.

Herrmannsdoerfer, Markus und Maximilian Koegel. 2010. Towards a generic operation recorder for model evolution. *In Proceedings of 1st International Workshop on Model Comparison in Practice (IWMCP '10)*, 76-81. Malaga, Spain: ACM.

Hitzler, Pascal, Markus Krötzsch, Sebastian Rudolph und York Sure. 2008. *Semantic Web: Grundlagen*. Berlin: Springer-Verlag.

Horrocks, Ian. 1998. Using an Expressive Description Logic: FaCT or Fiction? *In Proceedings of 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, 636-647.

Hunt, James W. und M. Douglas McIlroy. 1976. *An Algorithm for Differential File Comparison*. Computing Science Technical Report. Computing Science Technical Report.

Hunt, James W. und Walter F. Tichy. 2002. Extensible Language-Aware Merging. *In Proceedings of IEEE International Conference on Software Maintenance*, 0:0511. Los Alamitos, CA, USA: IEEE Computer Society.

IBM. 2010a. Rational Rose Modeler. http://www-142.ibm.com/software/products/de/de/rosemod, Stand 21.11.2010.

IBM. 2010b. Rational Team Concert. http://www-01.ibm.com/software/rational/products/rtc/, Stand 21.11.2010.

IBM. 2010c. Rational ClearCase. http://www-01.ibm.com/software/awdtools/clearcase/, Stand 24.11.2010.

IBM alphaWorks. 2007. IBM Integrated Ontology Development Toolkit. http://www.alphaworks.ibm.com/tech/semanticstk, Stand 26.9.2010.

Ignat, Claudia-Lavinia und Moira C. Norrie. 2004. Operation-based versus State-based Merging in Asynchronous Graphical Collaborative Editing. *In Proceedings of Sixth International Workshop on Collaborative Editing, Conference on Computer Supported Cooperative Work (CSCW'04)*. Chicago, USA.

Johnson, Stephen. 1977. *Lint, a C program checker. Computer Science Technical Report 65*. Computer Science Technical Report. Bell Laboratories.

Kalibatiene, Diana und Olegas Vasilecas. 2010. On OWL/SWRL mapping to UML/OCL. *In Proceedings of 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies - CompSysTech '10*, 58. Sofia, Bulgaria.

Kalyanpur, Aditya, Bijan Parsia, Evren Sirin, Bernardo Grau und James Hendler. 2006. Swoop: A Web Ontology Editing Browser. *Web Semantics: Science, Services and Agents on the World Wide Web* 4, no. 2: 144-153.

Karsai, Gabor, Janos Sztipanovits, Akos Ledeczi und Ted Babty. 2003. Model-integrated development of embedded software, 91:145-164. IEEE.

Kelter, Udo, Andreas Schürr und Bernhard Westfechtel. 2007. Vergleich und Versionierung von UML-Modellen. *In Proceedings of Software Engineering 2007, Fachtagung des GI-Fachbereichs Softwaretechnik*, 105:277-280. LNI. GI.

Kelter, Udo, Jürgen Wehren und Jörg Niere. 2005. A generic difference algorithm for UML models, hg. v. Peter Liggesmeyer, Klaus Pohl, und Michael Goedicke, 64:105-116. LNI. GI.

Kleppe, Anneke. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. 1. Aufl. Addison-Wesley Longman, Amsterdam.

Koegel, Maximilian und Jonas Helming. 2010. EMFStore. *In Proceedings of 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 307. Cape Town, South Africa.

Koegel, Maximilian, Jonas Helming und Stephan Seyboth. 2009. Operation-based conflict detection and resolution. *In Proceedings of ICSE Workshop on Comparison and Versioning of Software Models (CVSM)*, 43-48. Vancouver, BC, Canada.

Koegel, Maximilian, Markus Hermannsdoerfer, Otto von Wesendonk und Jonas Helming. 2010. Operation-based Conflict Detection on Models. *In Proceedings of International Workshop on Model Comparison in Practice, Intl. Conference on Model Transformation*. Malaga, Spain.

Koegel, Maximilian, Markus Hermannsdoerfer, Li Yang, Jonas Helming und Joern David. 2010. Comparing State- and Operation-based Change Tracking on Models. *In Proceedings of EDOC '10: 14th IEEE International EDOC Conference*.

Koegel, Maximilian, Helmut Naughton, Jonas Helming und Markus Herrmannsdoerfer. 2010. Collaborative model merging. *In Proceedings of ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*, 27. Reno/Tahoe, Nevada, USA.

Letkeman, Kim. 2005. Comparing and merging UML models in IBM Rational Software Architect: Part 3. CT316. http://www.ibm.com/developerworks/rational/library/05/802_comp3/, Stand 29.11.2010.

Leuf, Bo und Ward Cunningham. 2001. *The Wiki Way. Quick Collaboration on the Web.* Pap/Cdr. Addison-Wesley Longman, Amsterdam.

Lippe, Ernst und Norbert van Oosterom. 1992. Operation-based merging. *In Proceedings of Fifth ACM SIGSOFT symposium on Software development environments - SDE 5*, 78-87. Tyson's Corner, Virginia, United States.

Luoma, Janne, Steven Kelly und Juha-pekka Tolvanen. 2004. Defining Domain-Specific Modeling Languages: Collected Experiences. *4th OOPSLA Workshop on Domain-Specific Modeling (DSM04*.

MacGregor, Robert M. 1991. Inside the LOOM description classifier. *ACM SIGART Bulletin* 2, no. 3: 88-92.

MacKay, Stephen A. 1995. The State of the Art in Concurrent, Distributed Configuration Management.

Makarenkov, Victor, Pavel Jelnov, Azzam Maraee und Mira Balaban. 2009. Finite satisfiability of class diagrams. *In Proceedings of 6th International Workshop on Model-Driven Engineering, Verification and Validation - MoDeVVa '09*, 1-10. Denver, Colorado.

Meisinger, Michael, Andreas Rausch und Marc Sihling. 2006. 4everedit — Team-based Process Documentation Management. *Software Process: Improvement and Practice* 11, no. 6: 627-642.

Mens, Tom. 1999. A Formal Foundation for Object-Oriented Software Evolution. Brussel, Belgium: Ph.D. Dissertation, Department of Computer Science, Vrije Universiteit Brussel.

Mens, Tom. 2000. Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution. *In Proceedings of Applications of Graph Transformations with Industrial Relevance*, hg. v. Manfred Nagl, Andreas Schürr, und Manfred Münch. Bd. 1779. Berlin, Heidelberg: Springer Berlin Heidelberg.

Mens, Tom. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, no. 5: 449-462.

Mens, Tom und Theo D'Hondt. 2000. Automating Support for Software Evolution in UML. *Automated Software Engineering Journal* 7, no. 1: 39-59.

Mens, Tom, Ragnhild Van Der Straeten und Maja D'Hondt. 2005. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. *In Proceedings of Proc. les Jeunes sur l'Ingenierie Dirigee par les Modeles*, hg. v. Oscar Nierstrasz, Jon Whittle, David Harel, und Gianna Reggio, 4199:200-214. Lecture Notes in Computer Science. Springer.

Michel, Mirette M. und Galal H. Galal-Edeen. 2009. Detecting inconsistencies between software architecture views. *In Proceedings of International Conference on Computer Engineering & Systems*, 429-434. Cairo, Egypt.

Microsoft. 2003. Visual SourceSafe (VSS). http://msdn.microsoft.com/de-de/library/aa468128.aspx, Stand 1.12.2010.

Microsoft. 2010. Visual Studio Team Foundation Server 2010. http://www.microsoft.com/germany/visualstudio/products/team/visual-studio-team-foundation-server.aspx, Stand 21.11.2010.

MID GmbH. 2010. Modellierungsplattform Innovator. http://www.mid.de/produkte/modellierungsplattform-innovator.html, Stand 29.11.2010.

Milanović, Milan, Dragan Gašević, Adrian Giurca und Gerd Wagner. 2006. On Interchanging Between OWL/SWRL and UML/OCL. *In Proceedings of In OCL for (Meta-)Models in Multiple Application Domains*, 81-95.

moflon.org. 2010. MOFLON. http://www.moflon.org/, Stand 30.11.2010.

Murta, Leonardo, Hamilton Oliveira, Cristine R. Dantas, L. Lopes und Cláudia M. L. Werner. 2007. Odyssey-SCM: An integrated software configuration management infrastructure for UML models. *Science of Computer Programming* 65, no. 3: 249-274.

Network Working Group. 2010. RFC 4122 - A Universally Unique URN Namespace. http://tools.ietf.org/html/rfc4122, Stand 8.12.2010.

Nguyen, Tung Thanh, Hoan Anh Nguyen, Nam H. Pham und Tien N. Nguyen. 2010. Operation-Based, Fine-Grained Version Control Model for Tree-Based Representation. *In FASE*, hg. v. David S. Rosenblum und Gabriele Taentzer, 6013:74-90. Lecture Notes in Computer Science.

Niu, Nan, Steve Easterbrook und Mehrdad Sabetzadeh. 2005. A Category-theoretic Approach to Syntactic Software Merging. *In Proceedings of IEEE International Conference on Software Maintenance*, 0:197-206. Los Alamitos, CA, USA: IEEE Computer Society.

NoMagic Inc. 2010a. MagicDraw. http://www.magicdraw.com/, Stand 21.11.2010.

NoMagic Inc. 2010b. MagicDraw Teamwork Server. http://www.magicdraw.com/teamwork_server, Stand 29.11.2010.

OASIS. 2001. RELAX NG Specification. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html, Stand 30.11.2010.

Object Management Group (OMG). 2010a. XML Metadata Interchange (XMI). http://www.omg.org/spec/XMI/, Stand 30.11.2010.

Object Management Group (OMG). 2010b. Meta Object Facility (MOF). http://www.omg.org/spec/MOF/, Stand 30.11.2010.

Object Management Group (OMG). 2010c. Unified Modeling Language (UML). http://www.uml.org/, Stand 24.11.2010.

Ohst, Dirk, Michael Welle und Udo Kelter. 2003. Differences between versions of UML diagrams. *SIGSOFT Softw. Eng. Notes* 28, no. 5: 227-236.

Okubo, T., T. Matsutsuka, Y. Tanaka, H. Hara und S. Uehara. 1998. An optimistic method for updating information in distributed collaborative work. *In Proceedings of The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)*, 400-405. Vienna, Austria.

OMG. 2008a. Software and Systems Process Engineering
Metamodel specification (SPEM). http://www.omg.org/spec/SPEM/2.0/, Stand 30.11.2010.

OMG. 2008b. Query/View/Transformation (QVT) Specification. http://www.omg.org/spec/QVT/1.0/, Stand 30.11.2010.

OMG. 2010a. Model Driven Architecture. http://www.omg.org/mda/, Stand 30.11.2010.

OMG. 2010b. Object Contraint Language - OCL. http://www.omg.org/spec/OCL/2.2/, Stand 30.11.2010.

Parsia, Bijan und Evren Sirin. 2004. Pellet: An OWL DL Reasoner. *In Proceedings of the International Workshop on Decription Logics*.

Prudêncio, João Gustavo, Leonardo Murta und Cláudia M. L. Werner. 2009. On the Selection of Concurrency Control Policies for Configuration Management. *In Proceedings of XXIII Brazilian Symposium on Software Engineering (SBES)*, 155-164. Fortaleza, Ceara, Brazil.

Reidar Conradi und Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys (CSUR)* 30, no. 2: 232-282.

Rick Salay, Marsha Chechik, Steve Easterbrook, Zinovy Diskin, Pete McCormick, Shiva Nejati, Mehrdad Sabetzadeh und Petcharat Viriyakattiyaporn. 2007. An Eclipse-Based Tool Framework for Software Model
Management. *In Proceedings of OOPSLA workshop on eclipse technology eXchange*, 55-59. New York, NY, USA: ACM.

Rochkind, Marc J. 1975. The Source Code Control System. *IEEE Transactions on Software Engineering* 4, no. SE-1: 364-370.

Rosati, Riccardo. 2008. Finite model reasoning in DL-lite. *In Proceedings of 5th European semantic web conference on The semantic web: research and applications*, 215-229. Tenerife, Canary Islands, Spain: Springer-Verlag.

Rupp, Chris, Stefan Queins und Barbara Zengler. 2007. *UML 2 glasklar.* 1. Aufl. Hanser Fachbuchverlag.

Schematron Comitee. 2006. Information Technology - Document Schema Definition Language (DSDL). ISO/IEC.

Schmidt, Maik und Tilman Gloetzner. 2008. Constructing difference tools for models using the SiDiff framework. *In Proceedings of Companion of the 13th international conference on Software engineering - ICSE Companion '08*, 947. Leipzig, Germany.

Schmidt, Maik, Sven Wenzel, Timo Kehrer und Udo Kelter. 2009. History-based merging of models. *In Proceedings of ICSE Workshop on Comparison and Versioning of Software Models (CVSM)*, 13-18. Vancouver, BC, Canada.

Schneider, Christian und Albert Zündorf. 2007. Experiences in using Optimistic Locking in Fujaba. *Softwaretechnik Trends* 27.

Scott, John A. und David Nisse. 2001. *Software Configuration Management*. IEEE/ACM Software Engineering Body of Knowledge (SWEBOK) project. IEEE Computer Society,

Selic, Bran. 2007. A Systematic Approach to Domain-Specific Language Design Using UML. *In Proceedings of Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:2-9. Los Alamitos, CA, USA: IEEE Computer Society.

Spanoudakis, George und Andrea Zisman. 2001. Inconsistency Management in Software Engineering: Survey and Open Research Issues. *In Handbook of Software Engineering and Knowledge Engineering* 1: 329-380.

Stachowiak, Herbert. 1973. *Allgemeine Modelltheorie*. Springer-Verlag.

van der Straeten, Ragnhild. 2005. Inconsistency Management in Model-Driven Engineering. Dissertation, Brussel, Belgium: Ph.D. Dissertation, Vrije Universiteit Brussel.

Stuckenschmidt, Heiner. 2008. Debugging OWL Ontologies - A Reality Check. *In Proceedings of 6th International Workshop on Evaluation of Ontology-based Tools and the Semantic Web Service Challenge - EON-SWSC-2008*. Bd. 359. CEUR Workshop Proceedings. CEUR-WS.org.

Sun Microsystems. 2010. NetBeans - Metadata Repository (MDR). http://netbeans.org/, Stand 30.11.2010.

Ternité, Thomas. 2009. Process Lines: A Product Line Approach Designed for Process Model Development. *In Proceedings of 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 173-180. Patras, Greece.

Teufel, Stefanie, Christian Sauter und Thomas Mühlherr. 1995. *Computerunterstützung für die Gruppenarbeit*. Oldenbourg.

Tichy, Walter F. 1985. RCS - A System for Version Control. *Software — Practice and Experience* 15, no. 7: 637-654.

Tigris.org. 2010. Subversion. http://subversion.tigris.org/, Stand 21.11.2010.

Topcased. 2010. Topcased - The Open-Source Toolkit for Critical Systems. http://www.topcased.org/, Stand 23.11.2010.

Usman, Muhammad, Aamer Nadeem, Tai-hoon Kim und Eun-suk Cho. 2008. A Survey of Consistency Checking Techniques for UML Models. *In Proceedings of Advanced Software Engineering and Its Applications*, 0:57-62. Los Alamitos, CA, USA: IEEE Computer Society.

W3C. 2004. Web Ontology Language OWL. http://www.w3.org/2004/OWL/, Stand 8.12.2010.

W3C. 2009. OWL2 Web Ontology Language. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/, Stand 8.12.2010.

W3C. 2010. W3C XML Schema. http://www.w3.org/XML/Schema, Stand 30.11.2010.

Wahrig, Gerhard, Hrsg. 2005. Deutsches Wörterbuch. 7. Aufl. Gütersloh/München: Wissen Media Verlag.

Westfechtel, Bernhard. 2010. A formal approach to three-way merging of EMF models. *In Proceedings of 1st International Workshop on Model Comparison in Practice - IWMCP '10*, 31. Malaga, Spain.